On-the-Fly Load Data Value Tracing in Multicores

Mounika Ponugoti, Amrish K. Tewar, Aleksandar Milenković
Department of Electrical and Computer Engineering
The University of Alabama Huntsville
301 Sparkman Drive
Huntsville, AL, 35899 U.S.A.
{mp0046, akt0001, milenka}@uah.edu

ABSTRACT

Software testing and debugging of modern multicore-based embedded systems is a challenging proposition because of growing hardware and software complexity, increased integration, and tightening time-to-market. To find more bugs faster, software developers of real-time embedded systems increasingly rely on on-chip trace and debug resources, including hefty on-chip buffers and wide trace ports. However, these resources often offer limited visibility of the system, increase the system cost, and do not scale well with a growing number of cores. This paper introduces mlvCFiat, a hardware/software mechanism for capturing and filtering load data value traces in multicores. It relies on first-access tracking in data caches and equivalent modules in the software debugger to significantly reduce the number of trace events streamed out of the target platform. Our experimental evaluation explores the effectiveness of the proposed technique as a function of cache sizes, encoding mechanism, and the number of cores. The results show that mlvCFiat significantly reduces the total trace port bandwidth. The improvements relative to the existing Nexus-like load data value tracing range from 15 to 33 times for a single core and from 14 to 20 times for an octa core.

CCS Concepts

• Computer systems organization~Embedded hardware • Computer systems organization~Embedded software • Computer systems organization~Real-time system architecture *• Computer systems organization~Multicore architectures • Software and its engineering~Software testing and debugging*

Keywords

Real-time embedded systems; Multicores; Software testing and debugging; Program Tracing

1. INTRODUCTION

Growing complexity and sophistication of modern embedded systems and the shift toward multicores make software testing and debugging one of the most critical aspects of system development. Faster and cheaper processors with an increased level of integration have enabled new applications that were impossible just a decade ago. Users' expectations and their reliance on embedded systems have also gone up. 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 [2].

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 [4], [7], [5], [10]. Figure 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 ignoring mlvCFiat boxes). 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 from 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 [12] 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 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. To faithfully reconstruct a program execution in the software debugger, we need to capture and stream out load data values of memory and

CASES '16, October 01-07, 2016, Pittsburgh, PA, USA
© 2016 ACM. ISBN 978-1-4503-4482-1/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2968455.2968507

I/O reads, as well as exceptions on the target platform. However, these traces tend to be very large, in the order of 8-16 bits per instruction executed per processor core [9]. Capturing data traces in multicores is even more challenging because trace messages need to be ordered or time stamped. In addition, they need to include information about the origin of the trace message (core identification). Whereas a number of recent papers focus on capturing, compressing, and filtering control-flow traces [3], [5], [6], [11], [15], [17], relatively few studies look at on-the-fly data tracing [16]. Unfortunately, all these studies focus on single-core embedded platforms exclusively. In addition, the prior studies were based on functional simulation and did not address challenges of producing ordered or time-stamped trace messages coming from multiple cores. One interesting solution for debugging multicore SoCs called hidICE was proposed by Hochberger and Weiss [1]. It relies on a hardware emulator that replicates all master cores and memories from the target platform. The target platform reports only exceptions and data reads from peripherals that cannot be inferred by the emulator. However, hidICE is cost-prohibitive because it requires not only changes on the target platform to include a synchronization core and a new trace port, but also requires a sophisticated hardware emulator that replicates all the master modules and the RAM memory of the target. In addition, there has been no quantitative evaluation of hidICE. To the best of our knowledge, there have been no academic studies focusing on quantitative evaluation of data tracing requirements and development of cost-effective trace filtering mechanisms scalable to multicores.

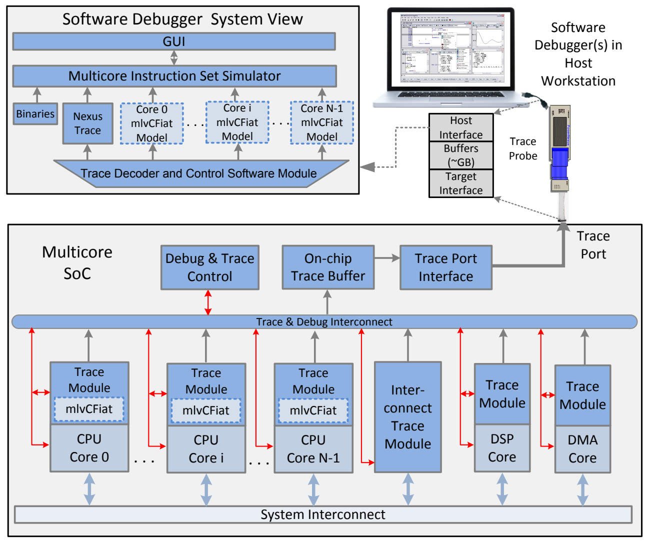

Figure 1. Multicore debugging and tracing infrastructure.

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 2). Next, we introduce mlvCFiat, a hardware/software framework for capturing and compressing load data values in multicores. mlvCFiat extends an existing method for capturing data traces in single-core platforms proposed by Uzelac and Milenkovic [16]. With mlvCFiat, data caches are augmented to include first-access tracking bits that help filter reads from memory, so that only first load accesses are traced out to the software debugger (Section 3). The first-access miss events are then encoded using effective and simple to implement encoding schemes (Section 3). Our experimental evaluation (Section 4) explores the effectiveness of mlvCFiat as a function of the number of cores, encoding mechanism, and data cache configurations. The results (Section 5) indicate that the mlvCFiat offers significant reduction in the required trace port bandwidth relative to the existing Nexus-like load data value tracing. The mlvCFiat with variable encoding reduces the trace port bandwidth in the range from 15 to 33 times for a single core with 16 KB and 32 KB data caches, respectively, and in the range from 14 to 20 times in a multicore with 8 processor cores, where each core has 16KB and 32KB private data caches, respectively.

The main contributions of this work are as follows:

- We characterize trace port bandwidth requirements in multicores for Nexus-like time stamped and untimed load 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 mlvCFiat for *multicore load value tracing using first-access tracking* to reduce the trace port bandwidth requirements.
- We perform a detailed experimental evaluation of the trace port bandwidth, while varying the number of cores, cache sizes, and encoding approaches.
- We analyze not only the average trace port bandwidth for each benchmark, but also consider variations of the trace port bandwidth during benchmark execution.

2. DATA TRACING IN MULTICORES

Exception traces and load data value traces captured on the target platform and streamed out to a software debugger are necessary to deterministically replay programs offline. Load data value traces are created by recording values read from memory and I/O devices. In addition to these traces, to faithfully replay the program offline the software debugger needs the following: (a) an instruction set simulator of the target platform, (b) access to the program's binary, and (c) the initial state of the general-purpose and special-purpose registers of individual cores. In multicores, the traces need to be either streamed in the order of occurrence (referred to as untimed traces) or they could be streamed out of order, but with global time stamps attached to each trace message (referred to as time-stamped traces). In our analysis we consider both alternatives.

To illustrate the tracing challenges in multicores, we consider a set of benchmarks and analyze the size of the load data value traces while varying the number of processor cores. As a metric we use the average trace port bandwidth (TPB) expressed in the number of bits per instruction executed (bpi) and the number of bits per processor clock cycle (bpc). The average TPB in bpi is calculated by dividing the total load data value trace size in bits with the number of instructions executed in a given benchmark. The average TPB in bpc is calculated by dividing the total trace size in bits with the benchmark execution time measured in processor clock cycles. The average TPB depends on the number of instructions executed, the frequency of instructions that read data from memory, and data types. The TPB in bpc also depends on the multicore model (pipeline, out-of-order execution, caches, and others), which can be characterized by the number of instructions committed per clock cycle (IPC).

Table 1 shows characteristics of interest for data tracing for the SPLASH-2 benchmarks [18] [8]. The benchmarks are compiled for the IA32 ISA and run on a cycle-accurate Multi2Sim [14] simulator that models multicores with N=1, 2, 4, and 8 cores. Table 1 shows (a) the number of instructions executed in billions (IC), (b) the IPC, and (c) the frequency of instructions that read

Table 1. Splash2 benchmark suite characterization

Benchmarks	*Instruction Count [IC] x10⁹*				*Instructions Per Cycle [IPC]*				*% Loads*			
No. of 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*
barnes	2.13	2.13	2.13	2.14	0.37	0.54	0.96	1.69	28.78	28.78	28.78	28.79
cholesky	1.27	1.43	1.95	3.07	0.19	0.41	0.92	2.12	27.78	29.54	30.32	31.30
fft	0.92	0.92	0.92	0.92	0.26	0.44	0.72	1.04	19.20	19.20	19.20	19.21
fmm	2.79	2.80	2.82	2.86	0.41	0.80	1.52	2.70	13.02	13.06	13.27	13.49
lu	0.45	0.45	0.45	0.45	0.39	0.74	1.27	1.95	20.20	20.22	20.25	20.31
radiosity	2.23	2.33	2.29	2.32	0.48	0.87	1.65	2.99	27.51	27.45	27.38	26.79
radix	1.59	1.59	1.59	1.60	0.23	0.36	0.54	0.65	35.09	35.09	35.09	35.09
raytrace	2.47	2.46	2.47	2.47	0.50	0.93	1.68	2.67	28.49	28.48	28.48	28.47
water-ns	0.74	0.74	0.74	0.75	0.61	1.17	2.22	3.90	16.31	16.33	16.36	16.42
water-sp	5.03	5.03	5.03	5.03	0.66	1.07	1.73	2.73	17.38	17.38	17.38	17.38
Total	19.61	19.87	20.39	21.60	0.40	0.69	1.21	1.95	22.77	22.96	23.21	23.67

data from memory. The IC remains constant or slightly increases with an increase in the number of cores, with an exception of *cholesky* where the IC increases significantly. The average IPC depends on the type of benchmarks, multicore models, 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. The frequency of instructions reading data from memory varies from 13% for *fmm* to 35% for *radix* and its total is ~23% for the entire benchmark suite. It increases slightly with an increase in the number of cores.

The Multi2Sim simulator is modified to capture load data values for committed instructions only. For untimed tracing we assume that trace messages coming from individual cores contain internal time stamps. These time stamps are used by the 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 field. Each trace message includes a (*Ti*, *LV*) pair, where *Ti* represents the core index (equivalent to the thread index) 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, Ti, LV)* triplet, where *dCC* represents the time in clock cycles measured from the beginning of the program execution or from the most recently streamed trace message at the given processor core (Figure 6a). This trace format complies with the Nexus format for single-cores, but it is extended to include information about the core id and the time stamp.

Figure 2a shows the average 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 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 *Ti* and (b) an increase in the frequency of memory reads. When *N*=8, the TPB ranges from 8.1 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 for all benchmarks divided by the sum of all instructions executed for all benchmarks. It ranges from 10.3 for *N*=1 to 11.0 bpi for *N*=8. For timed traces, the average TPB ranges from 8.8 for *fmm* to 15.4 bpi for *cholesky* when *N*=1, and from 9.3 for *fmm* to 16 bpi for *raytrace* when N=8. The total trace port bandwidth for the time-stamped traces ranges from 12.3 for *N*=1 to 13.2 bpi when *N*=8.

To further illustrate tracing challenges in multicores, we consider the TPB in bpc (Figure 2b). The required TPB for untimed 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., *water-sp*) place a lot of pressure on the trace port. The total average TPB ranges from 4.1 for *N*=1 to 21.5 bpc for *N*=8. In case of timed traces, the total average TPB increases even further to 4.9 when *N*=1 and to 25.6 bpc when *N*=8. Some benchmarks, e.g. *raytrace* and *water-ns*, require the average TPB of over 42 bpc. Whereas the results in Figure 2 indicate the average TPB for each benchmark, even higher peak bandwidths at the trace port are likely to occur during a benchmark execution. All these observations thus underscore a need for techniques that will reduce the volume of trace data that needs to be streamed out of the chip.

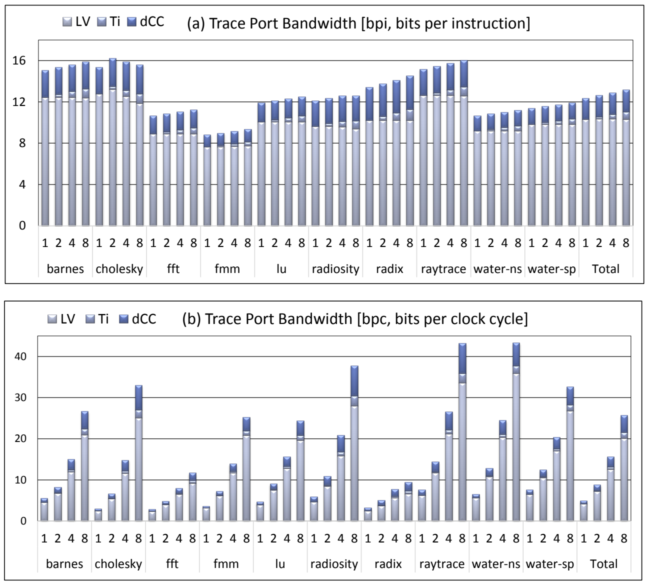

Figure 2. Trace port bandwidth for Nexus-like load data value traces.

3. mlvCFiat

mlvCFiat (*multicore load value cache first access tracking*) is a hardware-based mechanism that reduces load data value traces by capturing a minimal set of trace messages through the use of a cache first access mechanism. Figure 1 shows the block diagram of system debugging with light blue boxes representing additional mlvCFiat hardware and software modules. With mlvCFiat, each L1 data cache block in each processor core on the target platform is augmented with first access tracking bits (Figure 3). These bits keep track of sub-blocks that need to be reported to the software debugger. Let us assume an L1 data cache with 32-byte cache blocks. If a first-access tracking bit protects a 4-byte sub-block, each cache block needs to be augmented with an 8-bit first-access vector. The previously reported sub-blocks do not have to be reported again as they can be inferred by the software debugger. This way we exploit the temporal and spatial locality of data accesses to significantly reduce the number of trace events that needs to be reported. In addition to the first-access tracking bits, each trace module includes a local first-access counter (*Ti.fahCnt*) that counts the number of consecutive first-access hits.

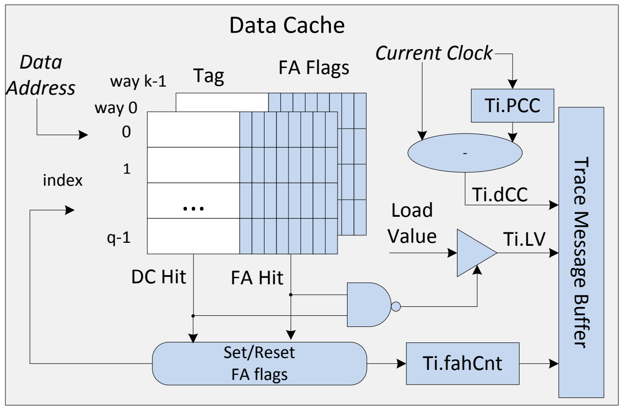

Figure 3. mlvCFiat structures for core *i*.

Figure 4 describes operation of the mlvCFiat mechanism on core *i* carried out for memory reads (lines 2-12), memory writes (lines 15-16), and external invalidate requests (line 19). Each memory read causes an L1 data cache lookup; if the requested data item is found in the data cache (a cache hit event) and the corresponding first-access bit(s) is set (an FA hit event), the data value does not need to be reported to the software debugger and *Ti.fahCnt* is incremented (line 3, Figure 4). In case of an FA miss event, a trace message is streamed out of the chip. The message includes the time stamp (*Ti.dCC*), the core id (*Ti*), the current value of *Ti.fahCnt*, and the load data value (*Ti.LV*) that is being reported for the first time (line 5). In addition, the corresponding FA bit(s) is set and the counter *Ti.fahCnt* is cleared (lines 6 and 7). In case of a data cache miss event, the newly fetched block's FA tracking bits are cleared and then the steps 5-7 are carried out. Similarly, external cache block invalidation or update requests invalidate the cache block and clear the corresponding FA bits (line 19). Finally, each memory write operation includes acquiring the exclusive ownership of the block and setting the corresponding FA tracking bit(s) (lines 15-16). Please note that we assume that the first-access tracking bits are tied to L1 data caches. By capturing trace events at the L1 level, cache coherence protocols are transparent to mlvCFiat. Thus, a write request to a shared block is treated as a miss in mlvCFiat. By capturing trace messages at each core, mlvCFiat complies with modular and scalable design methodologies.

Figure 5 describes steps carried out by the software debugger in response to memory reads (lines 2-10), memory writes (lines 13-15), and external invalidate requests (line 18). The debugger maintains software copies of the data caches and the *Ti.fahCnt* counters; these are updated during program replay using the same policies employed on the target platform. The program replay starts by reading and decoding the trace messages received from the target for each core separately. The format of the trace messages and the lengths of the individual fields are known to the software debugger. The debugger replays the instructions for each core using the corresponding instruction set simulator. For each memory read operation, the software copy of the counter *Ti.fahCnt* is decremented (line 2). If *Ti.fahCnt*>0, the debugger retrieves the load data value from the software copy of the data cache and moves to the next instruction (lines 4 and 5). If *Ti.fahCnt*=0, we have a first read miss event: the load data value is retrieved from the current trace message, the software copy of the data cache is updated, a new trace message for a given core is read from the trace probe, and the software copy of the *Ti.fahCnt* counter is loaded with a new value from the trace message (lines 7-9). For memory writes, if the block is shared in the cache, the current core acquires and exclusive ownership by invalidating copies of the block in other caches (line 13). The software copy of the cache block is updated and the corresponding FA bits are set (lines 14-15). In case of an invalidate request, the specified cache block is invalidated and all FA bits attached to that cache block are cleared.

```
1.  // For each read operation core i
2.  if (CacheHit) {
3.    if (corresponding FA bits are set) Ti.fahCnt++;
4.    else {
5.      Generate message (Ti.dCC, Ti, Ti.fahCnt, Ti.LV);
6.      Set corresponding FA bits;
7.      Ti.fahCnt = 0;
8.    }
9.  } else { // cache miss event
10.   Clear all FA bits for newly fetched cache block;
11.   Perform steps 5-7;
12. }
13.
14. // For each retired write operation
15. If (Shared) Acquire exclusive ownership;
16. Set the corresponding FA bits;
17.
18. // For external invalidation/update request
19. Invalidate the block and clear all FA bits;
```

Figure 4. mlvCFiat operation on the target core *i*.

```
1.  // For each read operation on core i
2.  Ti.fahCnt --;
3.  if (Ti.fahCnt > 0) {
4.    Perform lookup in the SW data cache;
5.    Retrieve data value from SW cache;
6.  } else { // FA miss event
7.    Read n bytes from trace record;
8.    Update SW cache;
9.    Get new message (Ti.dCC, Ti, Ti.fahCnt, Ti.LV);
10. }
11.
12. // For each store that writes n bytes
13. If (Shared) Acquire exclusive ownership;
14. Update SW cache;
15. Set the corresponding n SW cache FA bits;
16.
17. // For external block invalidate/update request
18. Invalidate the block and clear FA bits;
```

Figure 5. mlvCFiat operation in the software debugger for core *i*.

3.1 Hardware Implementation

mlvCFiat requires hardware extensions to support first-access tracking in L1 data caches for all processor cores. The majority of hardware overhead is due to the first-access tracking bits. The overhead depends on first-access tracking bits granularity and location, data cache size, and block size. The first-access tracking can be attached to the data cache blocks and control logic is added to maintain them (Figure 3). For example, if we assume cores with 32 KB data cache, 32-byte cache blocks, and first-access bit granularity of 4 bytes, the overhead is 1/32nd of the data cache capacity, or 1 KB of additional storage. Using complexity estimation based on Cacti tools [13], the total overhead is less than 3% of the regular L1 data cache area [16]. With finer granularity when each byte is protected with a first-access bit, we can possibly reduce the size of trace messages when byte sized memory reads dominate. However, the area overhead increases. With a coarse-grain granularity, every first-access miss event results in reporting the entire sub-block, regardless of the size of the memory read. Interestingly, coarse-grain granularity may have negative effects on the total size of trace messages in cases with poor spatial locality. However, it can also contribute to reducing the number of trace messages in cases when short operands are accessed sequentially (strong spatial locality).

Alternatively, the first-access flags can be implemented outside of processor cores in trace modules and connected to processor cores through a well-defined interface. In this case, mlvCFiat would need to include cache tags and address decoding, which introduces the additional hardware overhead. However, this approach may offer higher modularity and flexibility because the geometries of data caches in trace modules do not have to mirror actual processor data caches. However, duplicating cache tags results in an increased overhead that is slightly over 13% of the total L1 data cache area. In our analysis, we assume that the first-access bits are tied to the L1 data cache.

3.2 Encoding of Trace Messages

Trace messages streamed out through the trace port should be encoded in such a way to minimize the total number of bits. Figure 6 shows formats of trace messages for the Nexus-like load value trace (NX_b), mlvCFiat base encoding (CF_b), and mlvCFiat variable encoding (CF_e). With NX_b, each load data value, *LV*, is streamed out through the trace port together with a core index on which the read operation is carried out, *Ti*, and a differentially encoded time stamp, *dCC*. The length of the *Ti* 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_b, 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. In mlvCFiat, the length of the *LV* field also depends on the granularity size, GS, and can be calculated as follows: $8\cdot GS\cdot\lceil \texttt{sizeof(type)/GS} \rceil$ bits. For example, if the operand size is one byte and the granularity size is 4 bytes, the length of the *LV* field is 4 bytes or 32 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*, *Ti.dCC* = *Ti.CC* – *Ti.P.CC*, *Ti.P.CC* = *Ti.CC*. 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_b and CF_b 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).

With mlvCFiat, trace messages consist of the following fields: *dCC, Ti, fahCnt,* and *LV*. The *fahCnt* field contains the value of the counter *Ti.fahCnt* (the number of consecutive first-access hit events on core *i*). The number of bits needed to encode *fahCnt* varies as a function of FA miss rate. With CF_b we use at least 8 bits to encode the *fahCnt*. The connect bit (C) determines whether another 8-bit chunk is needed to fully encode the *fahCnt* value (C=1) or not (C=0). With CF_e, we allow chunks for encoding *fahCnt* (*i0*, *i1*, *i2*, …) and chunks for encoding *dCC* (*h0, h1, h2*, …) to be variable in size as shown in Figure 6c. We evaluate different encoding arrangements to select good values that minimize the number of bits needed to encode these fields.

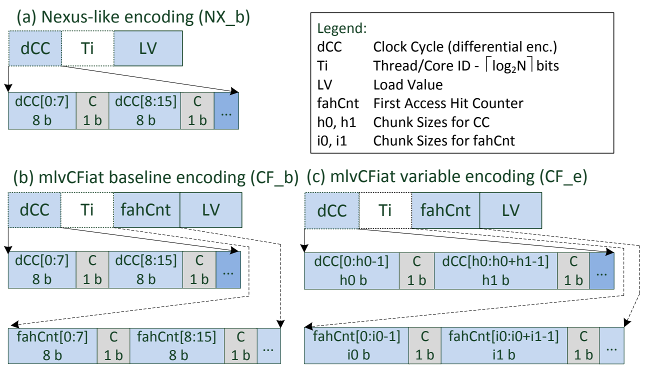

Figure 6. Formats of trace messages.

4. EXPERIMENTAL ENVIRONMENT

The goal of the experimental evaluation is to determine the effectiveness of the proposed mlvCFiat as a function of the number of cores. In addition, the goal is to quantitatively assess the impact of mlvCFiat configuration parameters (cache sizes, granularity sizes), and encoding parameters (baseline and variable) on its performance. As a measure of effectiveness, we use the average trace port bandwidth requirements expressed in bits per instruction and bits per clock cycle. Whereas the average trace port bandwidth 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 trace port bandwidth as a function of time during benchmark execution.

Figure 7 shows the experimental flow used to create hardware traces and evaluate the trace port bandwidth. The timed traces are collected using the Multi2Sim simulator executing IA32 ISA. The Multi2Sim simulator is extended with a custom TmTrace module that captures full 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 traces are read by the mlvCFiat simulator that simulates the behavior of data caches and mlvCFiat modules and generates compressed load data value traces (mlvCFiat trace). The output traces are then processed by trace filtering and encoding tools that determine trace port bandwidth and generate minimal hardware traces, namely the Nexus like trace, NX_b, and the compressed traces, CF_b and CF_e.

As the workload we use Splash2 [18] [8] benchmarks run with N=1, 2, 4, and 8 cores. Whereas the Splash2 benchmark suite may not be an ideal representative of cyber-physical systems targeted by this research, it includes well-understood standard parallel applications and application kernels that can work with a cycle-accurate simulator such as Multi2Sim.

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 Figure 8. Each core has its private level 1 instruction (L1I) and data (L1D) caches with hit latency of 4 clock cycles. To evaluate effectiveness of mlvCFiat 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 mlvCFiat tracking bits are added to L1 data cache tags. The L1 data caches are 4-way set-associative with the least-recently used replacement policy, cache block sizes are set to 32 bytes. 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.

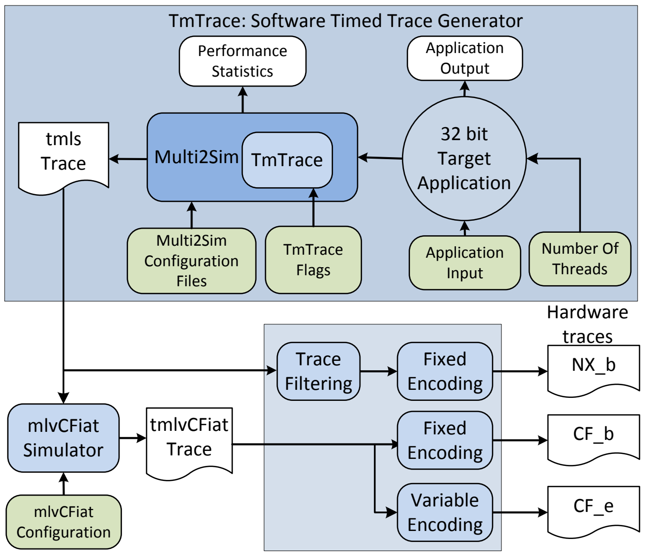

Figure 7. Experimental environment.

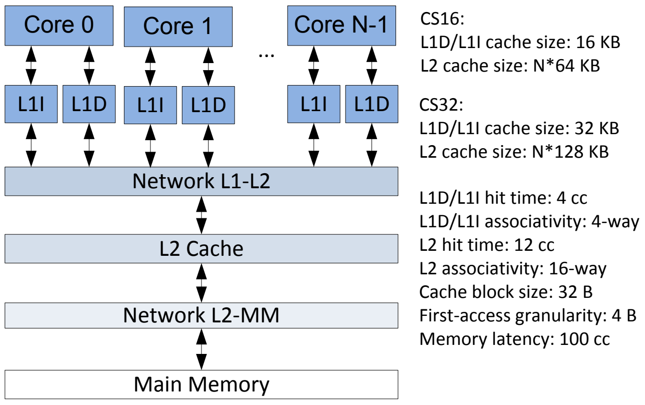

Figure 8. Multicore model.

An important part of experimental evaluation is determining a good granularity size. Figure 9 shows the normalized TPB for two representative benchmarks as a function of the number of cores, the cache configuration, and the granularity size. We consider granularity sizes of 1-byte, GS(1), 8-bytes, GS(8), 16-bytes, GS(16), and 32-bytes, GS(32). The TPB is normalized to the granularity size of 4 bytes, GS(4). The results show GS(1) offers limited (less than 4% for *barnes*) or no benefit at all (for *water-ns*). In general, we observed just a few benchmarks that read memory operands shorter than 4 bytes. By increasing the first-access granularity from 4 to 8 bytes, some benchmarks see an increase in the TPB (e.g., *barnes*), but some benchmarks do not (*water-ns*). Coarse-grained granularity of 16 bytes increases the TPB in the range from 40% to 70% relative to GS(4), depending on the benchmark, cache size, and the number of processor cores. When we use a single first-access bit per each cache block, GS(32), the TPB increases in the range from 60% to 145%. Consequently, our choice to use granularity of 4 bytes strikes a good balance between the method effectiveness and the hardware overhead.

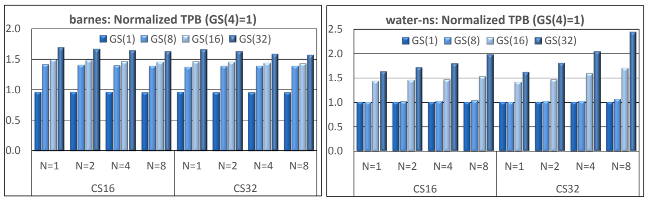

Figure 9. Normalized TPB as a function of FA granularity.

A part of the experimental evaluation is finding good chunk sizes for the variable encoded *dCC* and *fahCnt* fields used in CF_e. The number of bits needed to encode *dCC* and *fahCnt* depends on the frequency and distribution of first-access misses, which in turn depends on the number of cores and cache configurations. Whereas chunk sizes can be tailored for each combination of benchmarks and the mlvCFiat configurations, we seek chunk sizes that perform well across all benchmarks and configurations. We limit the search space by setting *i1=i2=…=ik*, and *h1=h2=…=hk*. Figure 10 shows the average bit length of the *fahCnt* field (left) and *dCC* field (right) when all benchmarks are considered together for different chunk sizes. The results show that chunk sizes (*i0, i1*)=(2, 2) and (*h0, h1*)=(4, 2) perform well for the *fahCnt* and *dCC* fields, respectively. Somewhat surprisingly we find that these chunk sizes exhibit good performance regardless of the number of cores. The findings hold for the CS32 configuration.

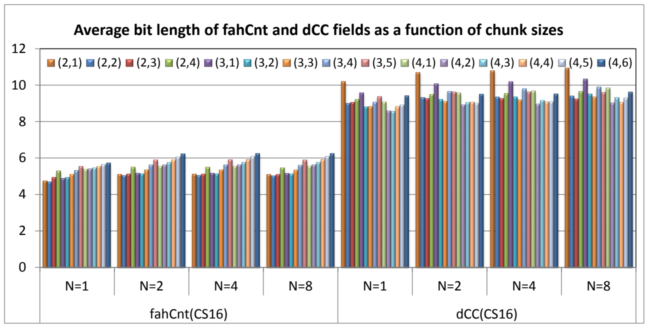

Figure 10. Average bit length of the fahCnt and dCC fields as a function of chunk sizes.

5. RESULTS

The effectiveness of mlvCFiat directly depends on (a) benchmark characteristics – namely, the type, frequency, and distribution of memory read operations, (b) data cache miss rates and first-access flag miss rates, and (c) encoding parameters. The first-access miss rate is a good indicator of the mlvCFiat effectiveness – the lower it is, the fewer trace messages need to be streamed out through the trace port. Figure 11 shows the total read L1 data cache miss rate

Table 2. Trace port bandwidth in bpi

# Cores	N=1			N=2			N=4			N=8		
Mechanism	*NX_b*	*CF_e*	*CF_e*	*NX_b*	*CF_e*	*CF_e*	*NX_b*	*CF_e*	*CF_e*	*NX_b*	*CF_e*	*CF_e*
Config.	-	*CS16*	*CS32*	-	*CS16*	*CS32*	-	*CS16*	*CS32*	-	*CS16*	*CS32*
barnes	15.03	2.21	0.79	15.31	2.30	1.16	15.59	2.39	1.47	15.86	2.44	1.77
cholesky	15.35	1.86	0.75	16.21	1.30	0.79	15.87	0.95	0.62	15.59	0.61	0.44
fft	10.65	2.57	1.48	10.84	2.62	1.50	11.02	2.65	1.52	11.19	2.67	1.54
fmm	8.82	0.36	0.23	8.96	0.37	0.24	9.14	0.38	0.25	9.33	0.38	0.26
lu	11.88	0.58	0.57	12.07	0.61	0.57	12.27	0.62	0.58	12.47	0.65	0.45
radiosity	12.11	0.25	0.09	12.36	0.55	0.45	12.59	0.56	0.44	12.58	0.63	0.54
radix	13.41	0.75	0.54	13.75	1.64	1.44	14.09	1.73	1.52	14.54	1.79	1.57
raytrace	15.17	1.06	0.34	15.45	1.28	0.61	15.73	1.38	0.71	16.01	1.54	0.90
water-ns	10.64	0.49	0.22	10.81	0.52	0.25	10.98	0.56	0.39	11.15	0.56	0.42
water-sp	11.38	0.07	0.05	11.55	0.07	0.06	11.73	0.08	0.07	11.90	0.09	0.08
Total	12.34	0.80	0.37	12.63	0.92	0.57	12.89	0.93	0.61	13.17	0.92	0.66

and the total first-access 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 read data cache and first-access 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 FA miss rate is calculated as the total number of first-access misses divided by the total number of data reads when all benchmarks are considered together. For the CS16 configuration, the total read L1 data cache miss rate is ~2% regardless of the number of cores, with the maximum of 4.66%. For the CS32 configuration, the total read L1 data cache miss rate is below 1.5% with the maximum of 3.18%. The total FA miss rate ranges between 5.67% (*N*=1) and 6.32% (*N*=2) for the CS16 configuration, and from 2.61% (*N*=1) and 4.24% (*N*=8) for the CS32 configuration. However, the maximum FA miss rate reaches as high as 17.99% with CS16 and 10.17% for CS32 (*fft* benchmark). Overall, the results confirm our expectations that mlvCFiat can indeed significantly reduce the number of trace messages that needs to be streamed out through the trace port. We find that the total FA miss rate does not increase significantly as we increase the number of cores. With an increase in the number of cores we may decrease the number of conflict-induced misses, but we may also increase the number of misses caused by invalidations.

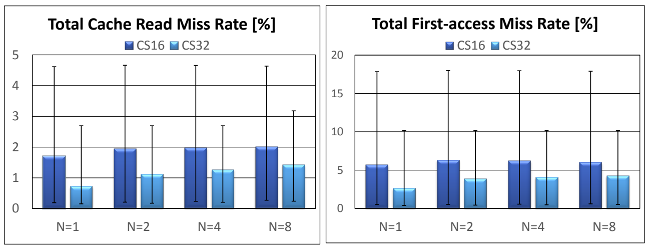

Figure 11. Data cache read miss rate & first access miss rate.

5.1 Trace Port Bandwidth in bpi

Figure 12 shows the total trace port bandwidth with the min-max ranges in bpi for the timed Nexus-like load data value traces (NX_b), the mlvCFiat with base encoding (CF_b), and the mlvCFiat with variable encoding (CF_e) as a function of the number of cores (*N*=1, 2, 4, 8). Table 2 shows the average trace port bandwidth in bpi for individual Splash2 benchmarks as well as the total average trace port bandwidth (row Total) for NX_b and CF_e with CS16 and CS32 configurations as a function of the number of cores. The results show that NX_b requires from 12.34 bpi when *N*=1 (ranging from 8.82 bpi for *fmm* to 15.35 bpi for *cholesky*) to 13.17 bpi when N=8 (from 9.33 bpi for *fmm* to 16.01 bpi for *raytrace*). An increase in the required TPB with an increase in the number of cores can be explained by an increased size of the *Ti* field in trace messages and an increase in the number of memory reads due to synchronization operations.

The mlvCFiat mechanism dramatically reduces the total trace port bandwidth requirements relative to NX_b. The total average TPB for CF_b with CS16 is 0.88 bpi when *N*=1 (from 0.07 for *water-sp* to 2.84 for *fft*) and 1.0 bpi when *N*=8 (from 0.09 for *water-sp* to 2.93 for *fft*). Compared to NX_b, CF_b(CS16) thus reduces the bandwidth 14.0 times for *N*=1 and 13.1 times for *N*=8. Expectedly, increasing the size of data caches leads to even lower trace port bandwidths. Thus, CF_b with CS32 requires only 0.40 bpi when *N*=1 (from 0.06 for *water-sp* to 1.63 for *fft*), and 0.71 bpi when *N*=8 (from 0.08 for *water-sp* to 1.89 for *barnes*). Compared to NX_b, CF_b(CS32) reduces the TPB 30.6 times when *N*=1 and 18.6 times when *N*=8.

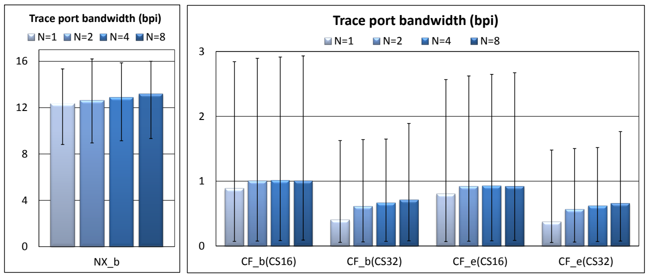

Figure 12. Total trace port bandwidth in bpi.

The variable encoding further reduces the total trace port bandwidth from 8% to 9% compared to CF_b for all configurations. CF_e with CS16 thus requires 0.80 bpi for *N*=1 (from 0.07 for *water-sp* to 2.57 bpi for *fft*) and 0.92 bpi for *N*=8 (from 0.08 for *water-sp* to 2.67 for *fft*). Compared to NX_b, CF_e with CS16 reduces the average trace port bandwidth 15.3 times when *N*=1 and 14.3 times when *N*=8. CF_e with CS32 requires merely 0.37 bpi when *N*=1 (from 0.05 bpi for *water-sp* to 1.48 bpi for *fft*) and 0.66 bpi when *N*=8 (from 0.08 bpi for *water-sp* to 1.77 bpi for *barnes*). Relative to the NX_b, CF_e with CS32 reduces the trace port bandwidth by 33.4 times when *N*=1 and 20.1 times when N=8. Table 3 shows the speedup (or compression ratio) achieved by CF_e with CS32 relative to NX_b for all individual benchmarks. The speedups range from as low as 4.2 times for *fft* to 210.3 times for *water-sp* when *N*=1 and to 156.2 times when *N*=8.

Table 3. Speedup TPB(NX_b)/TPB(CF_e)

# Cores	*N=1*		*N=2*		*N=4*		*N=8*	
Config.	*CS16*	*CS32*	*CS16*	*CS32*	*CS16*	*CS32*	*CS16*	*CS32*
barnes	6.8	19.1	6.7	13.1	6.5	10.6	6.5	9.0
cholesky	8.2	20.4	12.5	20.4	16.7	25.5	25.7	35.1
fft	4.2	7.2	4.1	7.2	4.2	7.3	4.2	7.3
fmm	24.2	37.8	24.1	37.2	24.2	36.7	24.4	36.6
lu	20.5	20.7	19.8	21.3	19.8	21.2	19.2	27.6
radiosity	47.7	128.8	22.3	27.6	22.6	28.4	19.8	23.4
radix	17.9	24.8	8.4	9.6	8.1	9.3	8.1	9.3
raytrace	14.3	44.2	12.1	25.4	11.4	22.1	10.4	17.9
water-ns	21.7	47.4	20.8	42.8	19.7	28.0	20.0	26.6
water-sp	168.1	210.3	158.5	189.4	147.3	170.9	136.9	156.2
Total	15.3	33.4	13.7	22.3	13.9	21.0	14.3	20.1

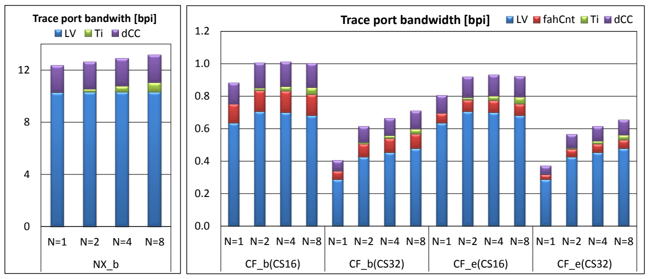

Figure 13. Total trace port bandwidth of individual trace fields in bpi.

Figure 13 shows the total trace port bandwidth in bpi broken down into individual fields of trace messages: *LV, Ti, fahCnt, dCC*. Expectedly, the majority of trace port bandwidth is consumed by streaming out the load values (*LV*). For NX_b, the *LV* portion ranges from 83% for *N*=1 to 78% for *N*=8. The time field is responsible for ~17% of the bandwidth regardless of the number of cores. Thus, if we order trace messages coming from different cores in the trace buffer and stream them out without the time field, the trace port bandwidth requirements will be lower for ~17%. However, this would require hardware support for buffering and sorting trace messages coming from individual processor cores before they are streamed out through the trace port. For CF_b, the *LV* field accounts for 67%-72% of the trace port bandwidth depending on the number of cores, the *fahCnt* field for ~13%, and the *dCC* field for ~16%. In CF_e, the *LV* field accounts for 73%-79% of the total bandwidth, the *fahCnt* for ~9%, and the *dCC* field accounts for ~15%. Thus, if further trace reduction is desired, reducing the size of the *LV* field (e.g., using dictionaries or predictors) would be the most beneficial approach.

5.2 Trace Port Bandwidth in bpc

Figure 14 shows the total trace port bandwidth with the min-max ranges in bpc for the timed NX_b, CF_b, and CF_e traces as a function of the number of cores and L1 data cache sizes. Table 4 shows the average trace port bandwidth for individual Splash2 benchmarks and the total bandwidth for NX_b and CF_e with the CS16 and CS32 configurations, as a function of the number of cores. Please note that we have trace port bandwidth in bpc reported for NX_b for both configurations CS16 and CS32 because the benchmark execution time depends on the data cache size. The results show that the total trace port bandwidth for NX_b scales linearly with the number of cores. With CS16, the TPB is from 4.92 bpc for *N*=1 (ranging from 2.79 for *fft* to 7.53 for *raytrace*) to 25.64 bpc for *N*=8 (ranging from 9.41 for *radix* to 43.51 for *water-ns*). With CS32, the total trace port bandwidth increases because larger data caches reduce the program execution time. The TPB is from 5.31 bpc for *N*=1 (ranging from 3.07 for *fft* to 8.62 bpc for *raytrace*) to 25.99 bpc for *N*=8 (ranging from 9.47 for *radix* to 45.82 bpc for *raytrace*). These results underscore the challenges in on-the-fly data tracing. The average trace port bandwidth of over 40 bpc would require a very wide trace port of over 100 pins dedicated to tracing because trace ports typically cannot run at processor clock speeds. In addition, it should be noted that more aggressive processor models with a higher IPC or multicores with a larger number of cores would both require even higher trace port bandwidth.

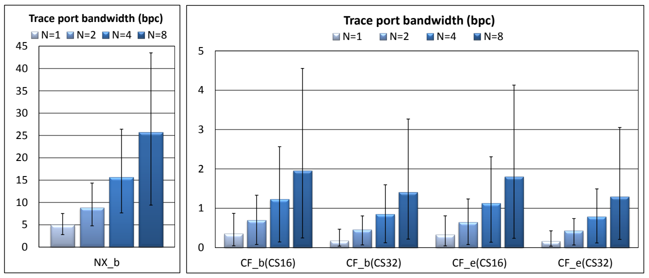

Figure 14. Trace port bandwidth in bpc.

Both CF_b and CF_e provide significant reductions in the trace port bandwidth. CF_e with CS16 requires from 0.32 bpc for *N*=1 (ranging from 0.04 for *water-sp* to 0.81 for *barnes*) to 1.79 bpc for *N*=8 (ranging from 0.24 for *water-sp* to 4.13 for *barnes*). CF_e with CS32 requires from 0.16 bpc for *N*=1 (from 0.04 for *water-sp* to 0.43 for *fft*) to 1.29 bpc for *N*=8 (from 0.21 for *water-sp* to 3.05 for *barnes*). Thus, we can say that CF_e with CS32 reduces

Table 4. Trace port bandwidth in bpc

# Cores	*N=1*				*N=2*				*N=4*				*N=8*			
Mechanism	*NX_b*	*NX_b*	*CF_e*	*CF_e*	*NX_b*	*NX_b*	*CF_e*	*CF_e*	*NX_b*	*NX_b*	*CF_e*	*CF_e*	*NX_b*	*NX_b*	*CF_e*	*CF_e*
Config	*CS16*	*CS32*	*CS16*	*CS32*	*CS16*	*CS32*	*CS16*	*CS32*	*CS16*	*CS32*	*CS16*	*CS32*	*CS16*	*CS32*	*CS16*	*CS32*
barnes	5.50	6.20	0.81	0.33	8.24	8.92	1.24	0.68	14.96	15.82	2.29	1.49	26.79	27.43	4.13	3.05
cholesky	2.93	3.30	0.36	0.16	6.58	7.44	0.53	0.36	14.67	16.75	0.88	0.66	32.99	32.91	1.28	0.94
fft	2.79	3.07	0.67	0.43	4.78	5.32	1.16	0.74	7.93	8.77	1.91	1.21	11.59	11.83	2.77	1.62
fmm	3.59	3.70	0.15	0.10	7.17	7.37	0.30	0.20	13.92	14.12	0.58	0.38	25.16	25.36	1.03	0.69
lu	4.67	5.16	0.23	0.25	8.97	9.68	0.45	0.46	15.56	16.68	0.78	0.79	24.38	24.64	1.27	0.89
radiosity	5.87	6.11	0.12	0.05	10.79	11.26	0.48	0.41	20.81	21.85	0.92	0.77	37.60	38.37	1.89	1.64
radix	3.14	3.39	0.18	0.14	5.01	5.20	0.60	0.55	7.67	7.87	0.94	0.85	9.41	9.47	1.16	1.02
raytrace	7.53	8.62	0.53	0.20	14.35	16.01	1.19	0.63	26.40	29.03	2.31	1.32	42.70	45.82	4.10	2.57
water-ns	6.49	7.00	0.30	0.15	12.68	13.50	0.61	0.32	24.34	24.71	1.24	0.88	43.51	43.29	2.17	1.63
water-sp	7.50	7.63	0.04	0.04	12.40	12.57	0.08	0.07	20.29	20.42	0.14	0.12	32.48	32.64	0.24	0.21
Total	*4.92*	5.31	0.32	0.16	8.76	9.32	0.64	0.42	15.61	16.44	1.13	0.78	25.64	25.99	1.79	1.29

the pressure on the trace port relative to NX_b between 33 times when N=1 to 20 times when N=8. In addition, we should note that the worst average TPB is reduced from 43.51 for NX_b to 2.17 for CF_e in the CS16 configuration. Similarly, the TPB is reduced from 45.82 for NX_b to 2.57 for CF_e in the configuration with CS32 (see Table 4).

5.3 mlvCFiat vs. Software Trace Compression

To underscore the effectiveness of the proposed mechanism we will compare it to a software trace compression. We consider the NX_b trace as an input trace and use gzip compression utility with compression level -1 to determine the compression ratio and thus the trace port bandwidth that can be achieved if such a mechanism is employed. The level -1 is used because of its relatively modest memory requirements, offering compression ratios that closer match those that could be achieved in hardware compressors. The trace messages from all cores are streamed into the compressor as they appear in the original NX_b trace. Please note that implementing a full general-purpose compressor dedicated to compressing load data value traces would require significant hardware resources.

Unfortunately, the software compression yields relatively modest compression ratios because the original NX_b trace is bit-packed to minimize the number of bits required on the trace port. Thus, possible redundancy in data traces cannot be exploited because gzip is a byte oriented compressor. Columns marked with *Unif* in Table 5 show the compression ratios for individual Splash2 benchmarks achieved by gzip with -1 as a function of the number of processors. The total compression ratios vary from 1.38 for N=4 to 1.54 times for N=1. These results show that the software compression will not significantly reduce the trace port bandwidth under given conditions.

Table 5. Speedup TPB(NX_b)/TPB(compressed NX_b)

# Cores	*N=1*		*N=2*		*N=4*		*N=8*	
Config.	*Split*	*Unif.*	*Split*	*Unif.*	*Split*	*Unif.*	*Split*	*Unif.*
barnes	2.10	1.38	1.78	1.30	1.66	1.24	1.58	1.27
cholesky	6.73	1.74	3.85	1.67	3.53	1.85	4.13	2.50
fft	1.93	1.39	1.79	1.36	1.68	1.30	1.66	1.37
fmm	4.95	1.95	3.73	1.85	3.06	1.58	2.75	1.59
lu	5.93	1.56	3.58	1.54	3.14	1.42	3.06	1.77
radiosity	3.86	1.63	2.50	1.54	2.10	1.37	1.95	1.48
radix	4.23	2.02	3.05	1.81	2.08	1.46	1.96	1.42
raytrace	3.88	1.51	2.61	1.47	2.26	1.32	2.08	1.38
water-ns	2.69	1.41	2.08	1.38	1.94	1.27	1.87	1.35
water-sp	3.03	1.37	2.40	1.36	2.11	1.26	2.02	1.39
Total	*3.33*	*1.54*	*2.52*	*1.49*	*2.21*	*1.38*	*2.18*	*1.52*

To achieve a higher compression ratio, we split the input NX_b trace into two streams, one with the (dCC, Ti) fields and the other with the load data values (LV). These two streams are fed into separate software compressors that use gzip utility. By splitting the input trace, we can better exploit redundancy present in individual trace fields. However, even with this modification, the total compression ratio remains fairly limited. Columns marked with *Split* in Table 5 show the compression ratios for individual Splash2 benchmarks achieved by gzip -1 as a function of the number of processors. The total compression ratios vary from 3.33 times when N=1 to 2.18 when N=8. By comparing these compression ratios to the ones presented in Table 3, we can conclude that CF_e significantly outperforms even software trace compression.

5.4 Dynamic Trace Port Bandwidth Analysis

Whereas the average trace port bandwidth allows us to quantify the effectiveness of mlvCFiat, 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 first-access misses, the trace port bandwidth at a given moment in a program execution may exceed the average trace port bandwidth discussed above.

Figure 15 and Figure 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 the time-stamped NX_b and CF_e traces with both configurations, CS16 and CS32. The benchmarks *raytrace* and *water-ns* are selected because they require the highest average total trace port bandwidth for the 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 TPB for NX_b with CS16 is 42.7 bpc. However, the peak bandwidth reaches ~61.5 bpc, further underscoring the challenges in load data value tracing. CF_e with CS16 requires the average TPB of 4.10 bpc with the peak values of 7.3 bpc, which is almost an order of magnitude smaller trace port bandwidth than for NX_b. CF_e with CS32 requires the average TPB of 2.57 bpc with the peak value of 4.9 bpc. These results indicate that CF_e not only reduces the average TPB, but also reduces the requirements for on-chip trace buffers. Similar observations stand for *water-ns*. The average TPB for NX_b with CS16 is 43.5 bpc and the peak TPB reaches 56.4 bpc. CF_e requires the average TPB of 2.17 bpc with the peak of 6.3 bpc with CS16 and 1.63 bpc with the peak of 6.0 bpc with CS32.

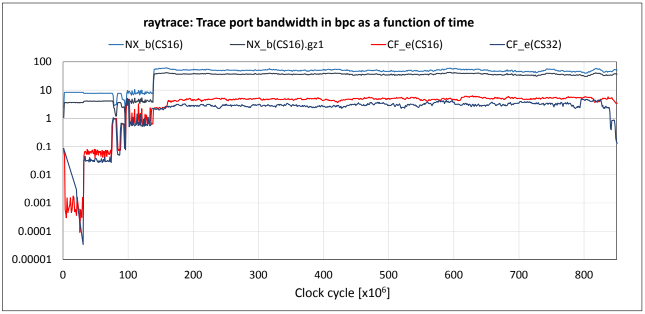

Figure 15. Dynamic trace port bandwidth in bpc during execution of *raytrace* for N=8.

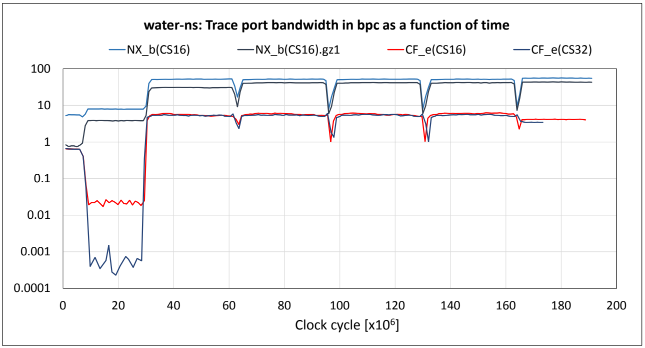

Figure 16. Dynamic trace port bandwidth in bpc during execution of *water-ns* for N=8.

6. 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 mlvCFiat, a technique for on-the-fly capturing and filtering load data value traces in multicore systems. mlvCFiat requires extensions of data caches to include first-access tracking bits, as well as software copies of data caches maintained by the software debugger. The first-access tracking bits, updated by memory read and write operations, determine which memory read operations need to be streamed out to the software debugger. This way we reduce the number of trace messages needed to replay the programs in the software debugger.

Our simulation-based experimental evaluation explores the effectiveness of mlvCFiat as a function of data cache sizes (16 and 32 KB), the encoding mechanism, and the number of processor cores (N=1-8). 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. mlvCFiat compression ratio relative to the Nexus-like load data value traces ranges from 15 to 33 times when N=1 and from 14 to 20 times when N=8. The effectiveness of mlvCFiat increases with an increase of private data cache size. The variable encoding scheme for time stamps and first-access counter fields of trace messages improves the effectiveness of mlvCFiat.

The future research efforts may focus on exploiting the first-access tracking bits with cache coherent protocol states, so that load data values are not reported on first-access misses if they are available in data caches of other processor cores. Another promising avenue is to explore whether a simple hardware scheme can be used to exploit redundancy in load data value traces.

7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable suggestions. 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.

8. REFERENCES

[1] Hochberger, C. and Weiss, A. 2008. Acquiring an exhaustive, continuous and real-time trace from SoCs. *IEEE International Conference on Computer Design, 2008. ICCD 2008* (Lake Tahoe, CA, Oct. 2008), 356–362. DOI= http://doi.org/10.1109/ICCD.2008.4751885.

[2] International Technology Roadmap for Semiconductors 2007 Edition: *https://goo.gl/TdZY52*. Accessed: 2016-04-08.

[3] Kao, C.-F., Huang, S.-M. and Huang, I.-J. 2007. A Hardware Approach to Real-Time Program Trace Compression for Embedded Processors. *IEEE Trans. Circuits Syst.* 54, 3 (Mar. 2007), 530–543. DOI= http://doi.org/10.1109/TCSI.2006.887613.

[4] MCDS - Multi-Core Debug Solution - Infineon Technologies: 2011. *https://www.ip-extreme.com/IP/mcds.shtml*. Accessed: 2016-04-01.

[5] Mihajlović, B., Žilić, Ž. and Gross, W.J. 2015. Architecture-Aware Real-Time Compression of Execution Traces. *ACM Trans Embed Comput Syst*. 14, 4 (Sep. 2015), 75:1–75:24. DOI= http://doi.org/10.1145/2766449.

[6] Milenković, A., Uzelac, V., Milenković, M. and Burtscher, B. 2011. Caches and Predictors for Real-Time, Unobtrusive, and Cost-Effective Program Tracing in Embedded Systems. *IEEE Trans. Comput.* 60, 7 (Jul. 2011), 992–1005. DOI= http://doi.org/10.1109/TC.2010.146.

[7] MIPS PDtrace Specification: 2009. *http://goo.gl/UwIYGv*. Accessed: 2016-04-01.

[8] Multi2Sim/m2s-bench-splash2: *https://goo.gl/5kbE8r*. Accessed: 2016-04-01.

[9] Orme, W. 2008. Debug and Trace for Multicore SoCs. *http://goo.gl/Wrc7Hk*. Accessed: 2016-03-28.

[10] Stollon, N. and Collins, R. 2006. Nexus Based Multi-Core Debug. *Proceedings of the Design Conference International Engineering Consortium* (Santa Clara, CA, USA, 2006), 805–822. http://goo.gl/VHn6vv.

[11] Tewar, A., Myers, A. and Milenković, A. 2015. mcfTRaptor: Toward unobtrusive on-the-fly control-flow tracing in multicores. *J. Syst. Archit.* 61, 10 (Nov. 2015), 601–614. DOI= http://doi.org/10.1016/j.sysarc.2015.07.005.

[12] The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface: 2003. *http://goo.gl/RZPYXU*. Accessed: 2016-03-28.

[13] Thoziyoor, S., Muralimanohar, N., Ahn, J.H. and Jouppi, N.P. 2008. *CACTI 5.1*. Technical Report #HPL-2008-20. HP Laboratories.

[14] Ubal, R., Jang, B., Mistry, P., Schaa, D. and Kaeli, D. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2012), 335–344. DOI= http://doi.org/10.1145/2370816.2370865.

[15] Uzelac, V. and Milenkovic, A. 2009. A Real-Time Program Trace Compressor Utilizing Double Move-to-Front Method. (San Francisco, CA, Jul. 2009), 738–743. DOI= http://doi.org/10.1145/1629911.1630102.

[16] Uzelac, V. and Milenković, A. 2013. Hardware-Based Load Value Trace Filtering for On-the-Fly Debugging. *ACM Trans. Embed. Comput. Syst.* 12, 2s (May 2013), 1–18. DOI= http://doi.org/10.1145/2465787.2465799.

[17] Uzelac, V., Milenković, A., Milenković, M. and Burtscher, M. 2014. Using Branch Predictors and Variable Encoding for On-the-Fly Program Tracing. *IEEE Trans. Comput.* 63, 4 (Apr. 2014), 1008–1020. DOI= http://doi.org/10.1109/TC.2012.267.

[18] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy, 1995), 24–36. DOI= http://doi.org/10.1109/ISCA.1995.524546.