# SCRIBE: A Hardware Infrastructure Enabling Fine-Grained Software Layer Diagnosis

Majid Dadashi, Layali Rashid and Karthik Pattabiraman
Department of Electrical and Computer Engineering
The Univeristy of British Columbia
{mdadashi, lrashid, karthikp}@ece.ubc.ca

***Abstract*—Recent studies have shown that intermittent faults have gained increased prominence on being responsible for computer system failures. This category of faults is harder to diagnose in comparison with permanent faults. Full hardware diagnosis techniques incur significant power and area overheads. Software layer diagnosis techniques have zero area overhead but limited visibility into many micro-architectural structures and hence cannot diagnose faults in them. We propose SCRIBE, a simple hardware infrastructure to enable fine-grained software layer diagnosis. SCRIBE records the detailed micro-architectural resource usage of each instruction in the processor and exposes it to the software diagnosis layer. Our evaluation indicates that SCRIBE has an overhead of 12 to 23% depending on the processor type.**

## I. Introduction

The continued scaling of Silicon devices has exacerbated their unreliability, and made them more susceptible to different kinds of faults. The common kinds of faults are transient and permanent. However, a third category of faults, namely intermittent faults has gained increased prominence. A recent study of commodity hardware has found that intermittent faults were responsible for about 40% of computer system failures due to hardware errors [1]. Unlike transient faults, intermittent faults are not one-off events, and occur repeatedly at the same location. However, unlike permanent faults, they appear non-deterministically, and are hence hard to diagnose. While intermittent faults can occur anywhere in the processor, memory and I/O sub-systems, in this paper we focus on intermittent faults that occur within the processor.

Intermittent faults are caused by marginal or faulty micro-architectural components, and hence diagnosing such faults is important to isolate the faulty unit [2], [3], [4]. Because micro-architectural components cannot be replaced easily, it is important to prevent the recurrence of the fault by reconfiguring around the faulty unit. Therefore, the diagnosis should be fine-grained at the granularity of individual units in a microprocessor. In prior work [5], we have shown that such fine grained diagnosis is essential for achieving high-performance in a fault-prone multi core processor, even under modest rates of intermittent faults.

Diagnosis can be carried out in either hardware or software. Hardware-level diagnosis has the advantage that it can be done transparently to the software, and the hardware can be reconfigured around the fault. Unfortunately, performing diagnosis entirely in hardware incurs significant power and area overheads. On the other hand, software-based diagnosis techniques only incur power overheads during the diagnosis process, and have zero area overheads [6]. Unfortunately, they have limited visibility into many micro-architectural structures (e.g., the reorder buffer) and hence cannot diagnose faults in them. More importantly, software techniques are often unable to distinguish between faults in multiple replicas of redundant functional units, which are found in most modern superscalar processors. This severely limits their applicability.

In this paper, we propose a hardware-software integrated technique for diagnosing intermittent hardware errors at the granularity of micro-architectural units. As mentioned above, intermittent faults are non-deterministic and may not be easily reproduced through posteriori testing. Therefore, the hardware portion of our technique continuously records the execution of a program instruction in terms of the micro-architectural resources it uses as it moves through the processor's pipeline, and stores this information in a log that is exposed to the software layer. When the program fails (due to an intermittent fault), this log is used by the software portion of our technique to identify which unit(s) of the microprocessor were subject to the intermittent fault that caused the program to fail.

In prior work, we have presented the software-layer design and algorithms to facilitate the diagnosis [6]. In this paper, we present the detailed design and operation of the hardware portion of the diagnosis technique. Because the hardware layer continuously records each instruction's execution in the program, we call this layer SCRIBE [1]. The main contribution of this paper is therefore a detailed exposition of the design and implementation trade-offs we made in SCRIBE, and an evaluation of the impact these tradeoffs have on the processor's performance. Our preliminary evaluation using a timing accurate processor simulator running the SPEC2006 benchmark suite shows that the average performance overhead of SCRIBE varies from 12% to 23% depending on the type of processor used (i.e., narrow, medium or wide).

## II. SCRIBE

In order to reveal the microarchitectural **R**esource **U**sage **I**nformation (RUI) to the software layer, a hardware logging mechanism is needed to store these information as the program

[1]In ancient ages, scribes were people who recorded all the transactions made in a city for bookkeeping purposes.

executes [7], [8], [3]. SCRIBE is used to provide this support (i.e. collecting RUI and revealing it to the software layer) in the processor. This section presents the functionality and microarchitecture of SCRIBE.

### A. RUI Format

We use the term RUI to denote the log corresponding to a single instruction's execution as it moves through the pipeline. The RUI records the units used by the instruction in each pipeline stage. Each field of the RUI corresponds to a single pipeline stage or functional unit. As an example, we consider an *add* instruction which is put in the entry 4 of Instruction Fetch Queue, entry 7 of the ROB, entry 24 of reservation station and also uses the second integer ALU. The RUI of this instruction is shown in figure 1.

The RUIs are stored in a circular buffer in a reserved address space in memory as the programs execute on the processor. We have found in a prior study that most intermittent faults cause programs to crash within a few thousand instructions from their start [9], and hence we choose the size of the buffer to be a few tens of thousands of instructions. Because the circular buffer is in a reserved area of memory, it does not interfere with the memory access patterns of the application.

| 000100 | 0000111 | 0011000 | 0001 | 111111 |
|---|---|---|---|---|
| **IFQ** | **ROB** | **RS** | **FU** | **LSQ** |

Fig. 1: The RUI entry corresponding to an *add* instruction

### B. Data Collection

We make use of the Reorder Buffer(ROB), a component in the superscalar architecuture, as a carrier for temporary RUI data. ROB is a hardware circular buffer implemented in the superscalar pipeline with one entry per dispatched yet not commited instruction [11]. We augment each ROB entry with an X bit field ($X \propto lg(\textit{Total number of resources})$) to keep the RUI of the instruction corresponding to that entry. This field is filled with valid RUI as the instruction traverses the pipeline and makes use of specific resources. Since there is a one to one correspondence between each instruction and an ROB entry, the complete RUI of the instruction can be known when it reaches the commit stage. The RUIs are sent to the memory hierarchy when their instructions are retired from ROB, and hence only correctly predicted instructions will be sent.

To gather information about which resources have been used by the corresponding instruction of that ROB entry, we need to connect the various stages of the pipeline with the ROB entry and send the information to the ROB at the appropriate time. The locations and the time for sending the information to the RUI field of an ROB entry is shown in table I.

Figure 2 shows the whole processor with our mechanism added. The processor chosen to show the added components is a *Superscalar DLX* processor ,(based on [10]) with some modifications to make the figure more understandable. The units related to SCRIBE in the figure are the *logging* and *priority handling* units which are added to the commit stage of the pipeline. The logging unit is in charge of compressing the RUI entries and sending them to the priority handling unit. The priority handling unit in each cycle chooses a regular store or a logging store to send to memory. These units are explained in detail in sections II-C and II-D. The interconnects transferring the resource usage to the ROB are shown using dashed lines.

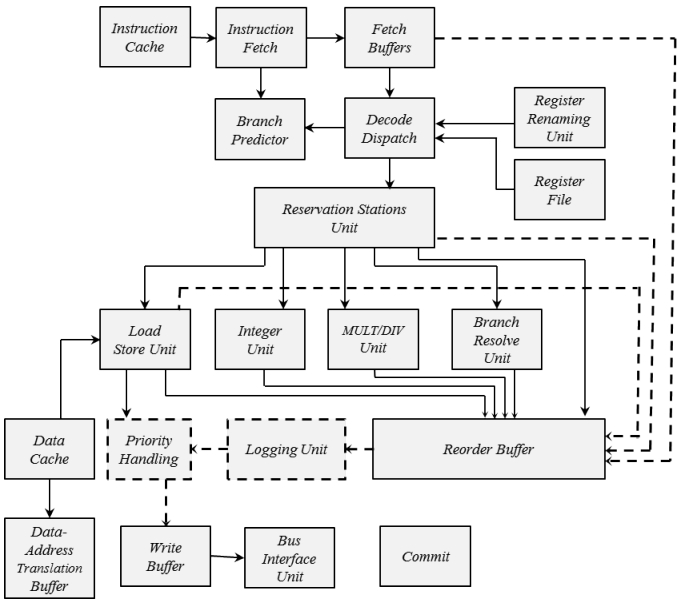


Fig. 2: Organization of SCRIBE in the context of a processor derived from SuperScalar DLX processor [10]. Dashed lines show the added units and interconnects to implement SCRIBE.

| Resource | Where | When |
|---|---|---|
| IFQ | IFQ Head Pointer | Instruction Reading In Dispatch Stage |
| ROB | ROB Tail Pointer | ROB entry allocation In Dispatch Stage |
| LSQ | LSQ Tail Pointer | LSQ entry allocation in Dispatch Stage |
| RS | *Select* port of RS entry MUX | RS entry allocation In Dispatch Stage |
| Functional Unit | *Select* port of FU selection MUX | Issue Stage |

TABLE I: Details of the additional connections that must be introduced in the pipeline and when the information should be sent to the RUI

### C. Logging Mechanism

As shown in figure 2, the ROB is connected to the logging unit which is in charge of sending the RUI to memory hierarchy for long time storage. The logging unit is expanded and shown in figure 3 along with the units communicating with it (*Priority handling*, LSQ and ROB). We explain the design of the logging mechanism in more detail below.

When an instruction is retired from the ROB buffer, the RUI field of its ROB entry will be inserted into one of the

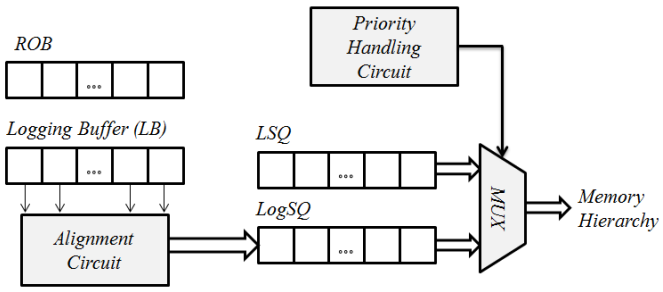


Fig. 3: The *Logging Unit* includes the *Logging Buffer*, *Alignment Circuit* and *LogSQ*

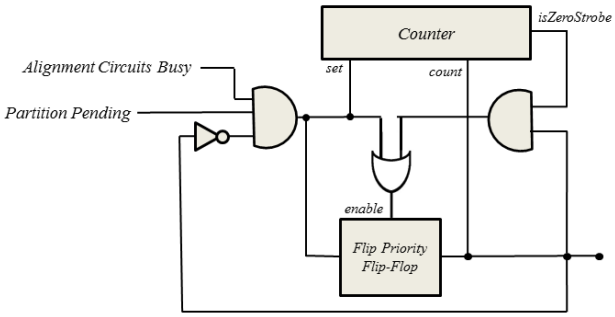


Fig. 4: The priority handling circuit schematic

partitions in the *Logging Buffer*. Logging Buffer (LB) is a dual partitioned queue and is in charge of keeping the RUI of the retired instructions. Each of the partitions of the *LB* get filled separately. The role of the *alignment circuit* is to compress the RUI data and send them to the memory. When one of the partitions is full, its data is processed by *Alignment Circuit* and the other partition starts getting filled and vice versa. Thus, *data processing* and *filling* mode alternate with each other in each partition of the logging unit.

When a partition of the logging buffer gets full, the alignment circuits start concatenating RUI fields of multiple log buffer entries to form *quadwords* (64 bit words). These quadwords are then stored in *Logging Store Queue (LogSQ)* where they compete with the regular loads and stores of the program to be sent to memory hierarchy. This process is explained in section II-D. If the LogSQ is full, the alignment circuits have to be stalled until a free entry in the logSQ becomes available. We discuss the implications of this stalling in Section III-C.

### D. Priority handling

The goal of the priority handling unit is to mediate accesses to main memory between the stores performed by the logging mechanism and the regular stores performed by the processor. The priority handling unit includes the priority handling circuit and a multiplexer to select between the regular store instructions and the *logging store*s. The schematic of the simple priority handling circuit is depicted in figure 4. The priority handling unit takes inputs from the (regular) *load and store unit* and the *logging unit* and has its output connected to the *write buffer*.

In the commit stage, when both a regular load/store instruction and a *logging store instruction* from *logSQ* are ready, one of them has to be chosen to be sent to memory hierarchy. If logging store instructions are not sent to memory on time, the logSQ becomes full and the alignment circuits are stalled. As described in section II-C, the retiring process will be stalled in this situation which incurs a performance overhead for the processor. The main challenge is to reduce the probability of this situation occuring and hence reduce the performance overhead.

One way to avoid the logSQ from becoming full is to always prioritize the logging stores over regular ones. This will prevent the regular stores from blocking the draining process of logSQ. However, this will lead to stalling the regular non-store instructions which do not need the store port before a blocked regular store. This is because instructions are retired from the ROB in a FIFO (First-In-First-Out) manner. If logging store instructions are always prioritized, a regular memory instruction at the head of the ROB would not be retired until the LogSQ is drained. This in turn will lead to stalling of the instruction commit mechanism in the processor, preventing the other partition of the logSQ from filling up with instructions. Our experiments show that this leads to severe performance slowdowns in the processor.

The other alternative is to prioritize regular stores over logging store instructions. However, this can lead to the processor being deadlocked due to the following sequence of events in the processor: ***(I)*** LogSQ becomes full and therefore one partition becomes full before processing of the other partition is finished. ***(II)*** As a result, the instruction retiring from the ROB will be stalled so that the logging mechanism can catch up. If there is a load/store effective address generation instruction at the head of ROB, it will also be held because the retiring process is stalled. ***(III)*** Since the held load/store has priority over logging store instructions, it will not let them be sent and so the logSQ will not drain. ***(IV)*** Hence, the alignment circuits will be kept stalled. This leads to a deadlock and the processor hangs.

Our solution is to use a hybrid approach where we switch the priorities between the logging stores and the regular stores based on the size of the LogSQ. We prioritize regular load/store instructions by default, until the logging mechanism starts stalling the commit stage (because of one partition becoming full before processing of the other one is finished). At this point, the logging store instructions gain priority over regular load/stores for approximately the number of cycles needed for logSQ to be drained. This value is computed at the time of the stall as: $\frac{|LogSQ|}{\#ofmemoryPorts}$. During all other periods, the regular stores continue to get priority over the logging stores, and hence do not hold up the other instructions in pipeline.

## III. Evaluation

In this section, we first describe the methodology used for evaluating SCRIBE. We then present the results of the evaluation and a detailed breakdown of the results.

### A. Methodology

We implemented SCRIBE in *sim-mase*, a cycle-accurate mciroarchitectural simulator, a part of the SimpleScalar package [12]. The common configurations between the baseline processors we simulated are shown in table II. We use three

| Parameter | Value |
|---|---|
| Level 1 Data Cache | 32K, 4-way, LRU, 1-cycle latency |
| Level 1 Instruction Cache | 32K, 4-way, LRU, 1-cycle latency |
| Level 2 combined data & instruction cache | 512K, 4-way, LRU, 8-cycle latency |
| Branch Predictor | Bi-modal, 2-level |
| Instruction TLB | 64K, 4-way, LRU |
| Data TLB | 128K, 4-way, LRU |
| Memory Access Latency | 200 CPU Cycles |
| Simulation Length | $10^9$ instructions |
| Simulation Warm Up | $10 \times 10^6$ instructions |

TABLE II: Default machine configurations

different processor configurations (Narrow, Medium and Wide pipelines) for our experiments. These configurations represent different families of processors, from standard or simple to higher end commercial processors. These configurations are similar to the ones used in Timor et al. [13]. The configurations are listed in table III. The width of the processor affects the RUI size and the utilization of resources which in turn impacts the performance overhead of SCRIBE. We choose the LogSQ and Logging Buffer to be 32 and 64 entries respectively as our experiments have shown increasing their sizes beyond these numbers has no significant effect on performance.

| Topic | Parameter | Machine Width | | |
|---|---|---|---|---|
| | | *Nar.* | *Med.* | *Wide* |
| Pipeline Width | Fetch | 2 | 4 | 8 |
| | Decode | 2 | 4 | 8 |
| | Issue | 2 | 4 | 8 |
| | Commit | 2 | 4 | 8 |
| Array Sizes | ROB Size | 64 | 128 | 256 |
| | LSQ Size | 32 | 32 | 32 |
| Number of ALUs | Integer Adder | 2 | 4 | 8 |
| | Integer Multiplier | 1 | 1 | 1 |
| | FP Adder | 1 | 1 | 2 |
| | FP Multiplier | 1 | 1 | 1 |

TABLE III: Different Machine Widths

To evaluate SCRIBE, we use seven benchmarks from the SPEC 2006 integer benchmarks set and two (*soplex* and *dealII*) from the SPEC2006 floating-point benchmarks set. We choose these benchmarks as they were compatible with our infrastructure. We did not cherry-pick them based on the results. All the benchmarks have been fast forwarded for $10^7$ instructions and then simulated in detail for $10^9$ instructions. The goal of the experiments is to measure the performance overhead of SCRIBE. This is measured as the percentage of extra cycles incurred by the simulator to run the benchmark programs. We do not evaluate fault coverage in these experiments.

### B. Results

Figure 5 shows the performance overhead incurred by SCRIBE for the seven benchmarks and across three configurations. The *geometric mean* of the overheads across all configurations is 14.7 %. In most of the cases (all except *soplex-pds*), the *wide* configuration ($GeoMean = 23.21\%$) incurs higher overhead than the *medium* ($GeoMean = 11.88\%$) and *narrow* ($GeoMean = 11.53\%$) configurations. The *Medium* and *narrow* configurations are comparable in terms of overhead. This is because the *wide* processor is utilizing the resources in a better way leaving fewer free slots to be used by SCRIBE and therefore is stalled more often, leading to higher performance overheads.

Figure 6 shows correlation between the performance overhead for each benchmark and the baseline IPC of the benchmark. We find that there is a positive correlation between the IPC and the overhead (i.e. overhead increases as the baseline IPC increases). For example, the baseline IPC values for *perl* and *libquantum* are 1.35 and 3 and their overheads are 6.18% and 28.05% respectively. This result is intuitive, as SCRIBE uses underutilized resources in the processor to perform its logging, and a higher IPC indicates that fewer resources are under-utilized and are hence available to SCRIBE.

### C. Discussion

To better understand the performance overheads incurred by SCRIBE, we break down the overhead into four parts. Figure 7 shows the overhead breakdown of SCRIBE for the medium width processor. The overhead components are:

- **Memory Ports Pressure**: Since multiple *Logging Store*s are being sent to the memory hierarchy by SCRIBE, the memory ports might get busy in some cycles when needed by regular loads/stores. This would make the regular loads and stores stall. This component is responsible for $\sim 41\%$ of the total SCRIBE overhead.
- **Stalling Regular Stores**: In the cycles in which the logging mechanism has priority, regular stores at the head of ROB will be stalled and hence the commit stage becomes stalled. This will continue until the regular stores gain priority over *logging store*s again. This component is responsible for $\sim 32\%$ of the total SCRIBE overhead.
- **Reducing Commit Bandwidth**: The commit stage has a limited bandwidth. SCRIBE consumes part of this bandwidth in each cycle by attempting to send logging stores to the memory hierarchy. This part of the overhead is responsible for $\sim 23\%$ of the overhead.
- **Stalling the Commit Stage**: This happens when one partition of the *Log Buffer* is not still done being compressed while the other partition becomes full. In this case, the commit stage will be stalled. This component is responsible for only $\sim 4\%$ of the total SCRIBE overhead.

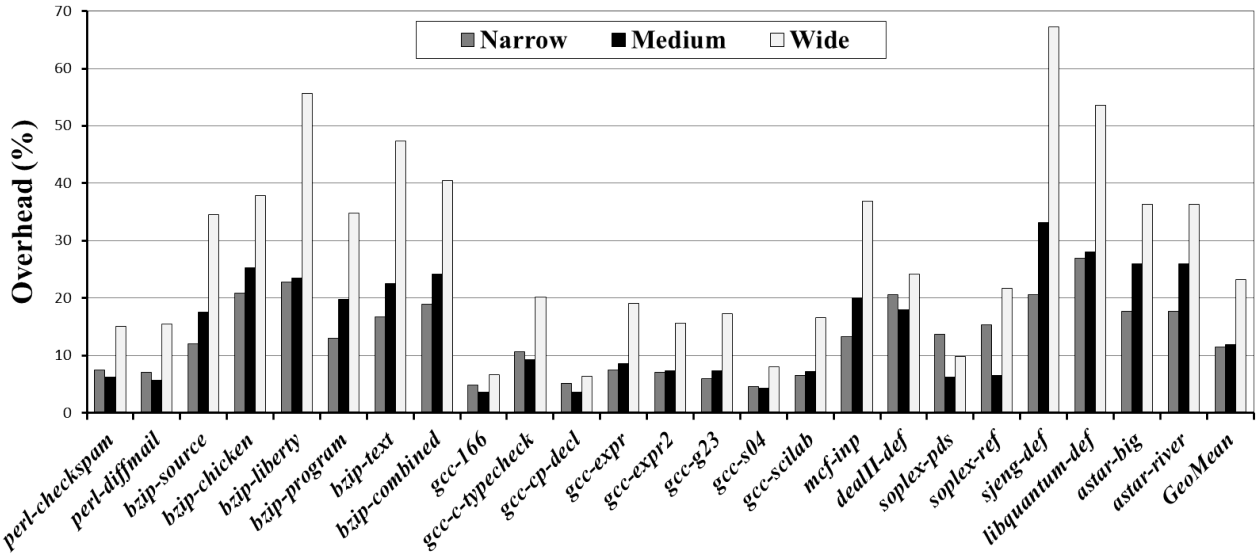


Fig. 5: The performance overhead of SCRIBE applied to three configurations: *Narrow*, *Medium* and *Wide*

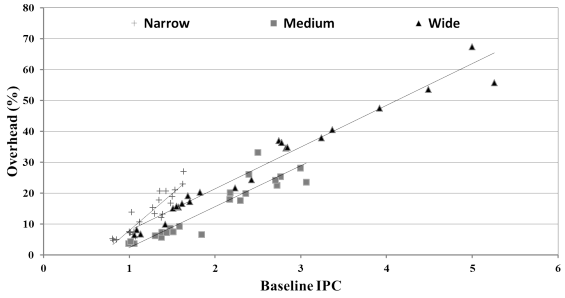


Fig. 6: Correlation between baseline IPC and Performance Overhead. The trendline for each dataset is also shown.

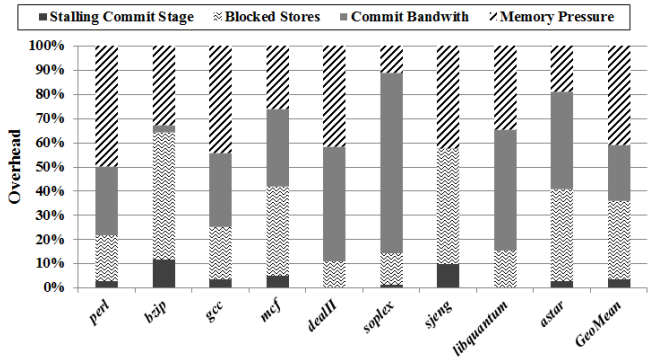


Fig. 7: The breakdown of the overhead into different sources for medium width processor (Normalized to overhead in each benchmark)

Thus we see that the overhead of SCRIBE is dominated by the memory pressure it introduces to the processor, as well as the stalling of regular stores in the processor.

### *D. Sensitivity Analysis*

In this section, we perform a sensitivity analysis of the performance overhead of SCRIBE to the sizes of Logging Buffer and LogSQ. Recall that these are structures added by SCRIBE to the processor to store the log data. In the earlier part of the evaluation, we had fixed the sizes of these buffers to be 32 and 64 entries, respectively. We explain the reason for these choices below.

We use the medium width processor from table III in this part of the evaluation. Further, for conciseness we use three of the nine benchmarks used earlier. These benchmarks are representative of the general trend we observed across all nine benchmark programs.

Figures 8 and 9 show the variation of the overhead with the sizes of the Logging Buffer and LogSQ. As can be seen in the figures, the overhead initially decreases as we increase the sizes of the buffers. However, increasing the sizes of Logging Buffer and LogSQ beyond 32 and 64 entries respectively does not provide any benefit in terms of decreasing the overhead. This is because the overhead of SCRIBE is dominated by the memory pressure and stalled loads and stores (section III-C) both of which are independent of the sizes of the Logging Buffer and LogSQ. However, the component *Stalling Commit Stage* of the overhead does depend on these two sizes, but this only accounts for 4% of the overhead of SCRIBE. Note that at small buffer sizes, this overhead dominates, which is why the overhead of SCRIBE is high for smaller values of the sizes of the Logging Buffer and LogSQ.

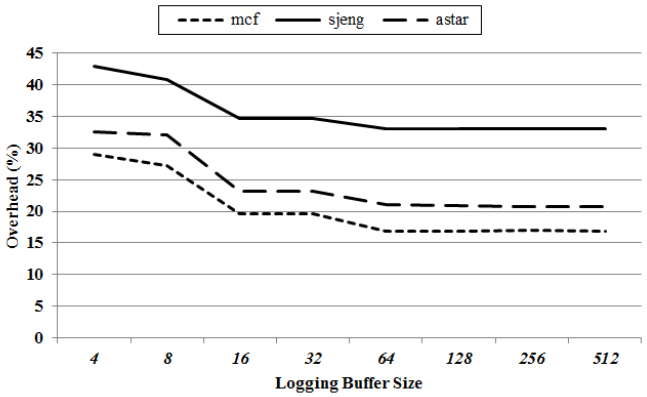


Fig. 8: The overhead with respect to logging buffer size for medium width processor

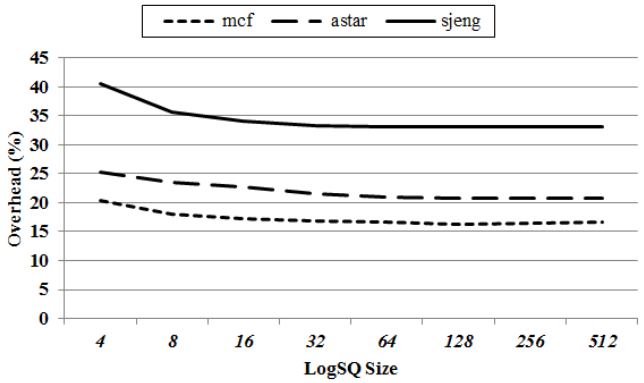


Fig. 9: The overhead with respect to logSQ size for medium width processor

## IV. Related Work

Bower et al. [3] propose a *hardware* layer diagnosis mechanism for *permanent* faults. They use a similar mechanism as ours for collection of resource usage information; however, they do not expose this information to the software layer. To our knowledge, there is no prior work on online gathering of microarchitectural resource usage and exposing it to the software layer.

Li et al. [8] propose a *software* diagnosis algorithm for *permanent* faults. They introduce a hardware component called *Instruction Trace Buffer* which only records information of the the retired instructions in the diagnosis mode (replay mode after crash). However, this mechanism relies on deterministic scheduling of instructions on micro-architectural resources, which may not be the case in complex processors. Further, it cannot be applied to *intermittent* faults which may not reoccur while replaying the execution.

IFRA [14], is a post-silicon diagnosis method, which records the footprint of every instruction as it executes in the processor to find faulty components. SCRIBE differs from IFRA in two ways. First, IFRA records the instruction information within the processor, and this information has to be scanned out after the failure. On the other hand, SCRIBE includes a mechanism to write out the information to memory which is essential for online diagnosis. Second, IFRA assumes the presence of hardware-based fault detectors to limit the error propagation. However, it is unclear how these detectors are derived, and also how much overhead they incur. In contrast, SCRIBE does not require any additional detectors in the hardware or software.

Finally, Carratero et al. [7] performs integrated hardware software diagnosis for the Load-Store Unit (LSU). Our work is similar to theirs in some respects. However, SCRIBE covers faults in the entire pipeline, and not only the LSU. Also, their focus is on bug localization and not in-field fault diagnosis.

## V. Conclusion

This paper introduced SCRIBE, a simple hardware infrastructure to enable fine-grained software layer diagnosis. SCRIBE records the detailed micro-architectural resource usage of each instruction in the processor and exposes it to the software diagnosis layer. Our preliminary evaluation indicates that SCRIBE has an overhead of 12 to 23% depending on the processor type.

As future work, we plan to evaluate the end-to-end performance of SCRIBE and its fault coverage, in conjunction with various software diagnosis algorithms. We also plan to explore micro-architectural techniques to further optimize the overhead of SCRIBE.

**Acknowledgements:** This work was funded in part by an Discovery Grant and an Engage Grant, both from the National Science and Engineering Research Council of Canada (NSERC). We thank the anonymous reviewers of SELSE for helping us make this paper better.

## References

[1] E. Nightingale *et al.*, "Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer pcs," in *Proc. EuroSys*, 2011.
[2] P. Wells *et al.*, "Adapting to intermittent faults in multicore systems," *Proc. of 13th ASPLOS*, 2008.
[3] F. Bower *et al.*, "A mechanism for online diagnosis of hard faults in microprocessors," in *MICRO*, 2005.
[4] S. Gupta *et al.*, "Stagenet: A reconfigurable fabric for constructing dependable cmps," *IEEE Trans. Computers*.
[5] L. Rashid *et al.*, "Intermittent hardware errors recovery: Modeling and evaluation."
[6] L. Rashid *et al.*, "Dieba: Diagnosing intermittent errors by backtracing application failures," *SELSE*, 2012.
[7] J. Carretero *et al.*, "Hardware/software-based diagnosis of load-store queues using expandable activity logs," in *HPCA*, 2011.
[8] M.-L. Li *et al.*, "Trace-based microarchitecture-level diagnosis of permanent hardware faults," in *DSN*, 2008.
[9] L. Rashid *et al.*, "Modeling the propagation of intermittent hardware faults in programs," in *PRDC*, 2010.
[10] H. Eveking, "Superscalar dlx documentation," http://www.rs.tu-darmstadt.de/downloads/docu/dlxdocu/DlxPdf.zip.
[11] J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, 1995.
[12] E. Larson *et al.*, "Mase: a novel infrastructure for detailed microarchitectural modeling," in *ISPASS. 2001*.
[13] A. Timor *et al.*, "Using underutilized cpu resources to enhance its reliability," *IEEE Trans. Dependable Secur. Comput.*, 2010.
[14] S. Park and S. Mitra, "Ifra: instruction footprint recording and analysis for post-silicon bug localization in processors," in *DAC*, 2008.