GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates

Konstantinos Parasyris* Georgios Tziantzoulis† Christos Antonopoulos‡ Nikolaos Bellas§
*‡§Dept. of Electrical and Computer Eng. *‡§I.RE.TE.TH. †Computer Science Dept.
University Of Thessaly Centre for Research and Technology, Hellas Northwestern University
Volos, Greece Volos, Greece Chicago, U.S.A.
E-mail: *koparasy,‡cda,§nbellas@inf.uth.gr, †georgiostziantzioulis2011@u.nortwestern.edu

***Abstract*—Dependable computing on unreliable substrates is the next challenge the computing community needs to overcome due to both manufacturing limitations in low geometries and the necessity to aggressively minimize power consumption. System designers often need to analyze the way hardware faults manifest as errors at the architectural level and how these errors affect application correctness.**

This paper introduces GemFI, a fault injection tool based on the cycle accurate full system simulator Gem5. GemFI provides fault injection methods and is easily extensible to support future fault models. It also supports multiple processor models and ISAs and allows fault injection in both functional and cycle-accurate simulations. GemFI offers fast-forwarding of simulation campaigns via checkpointing. Moreover, it facilitates the parallel execution of campaign experiments on a network of workstations.

In order to validate and evaluate GemFI, we used it to apply fault injection on a series of real-world kernels and applications. The evaluation indicates that its overhead compared with Gem5 is minimal (up to 3.3%), whereas optimizations such as fast-forwarding via checkpointing and execution on NoWs can significantly reduce simulation time of a fault injection campaign.

***Keywords*-fault-injection; simulation; cycle accurate; full system**

I. Introduction

Fault injection is a fundamental experimental method for assessing the dependability and identifying weaknesses in the design of fault tolerant systems. Fault injection can also be used to simulate execution on unreliable systems, to study the behavior of applications under the presence of faults, or evaluate the coverage of a software fault tolerance mechanism.

Different fault injection techniques are appropriate for different phases of the design cycle of systems. Simulator-based fault injection can be used early in the design cycle, software and pin level fault injection require the availability of a system prototype. Most fault injection tools target a specific system using customized user interfaces. Extending such infrastructures is unrealistic, limiting the tool's application domain.

In this paper we introduce *GemFI*, a cycle accurate fault injection tool based on the Gem5 simulator [1]. A primary objective of the tool is to enable fault injection based on different fault models and on systems with various configurations. We target full system simulations to evaluate the impact of faults on the complete system stack, from the the architectural level up to applications. A variety of different system configurations and architectures can be supported without affecting the implementation of fault injection in GemFI.

GemFI supports the generic behavioral-level fault model for the register file within a processor [2]. The model describes the behavior of micro-architectural components of a generic processor under the presence of faults. The fault model is abstracting the low-level fault effects to the micro-architectural level. It can be used to simplify and accelerate fault injection campaigns – compared with injection on RTL models – without sacrificing accuracy of the obtained results.

GemFI can support any processor model and ISA available in the Gem5 simulator. For the purposes of this study we use the Alpha ISA[1]. To provide more accurate results, GemFI injects faults on the CPU while simulating both user- and kernel-level instructions and models a complete system including CPU, memory and the peripheral devices. The tool includes a number of performance enhancing features: checkpointing and restart functionality and the ability to launch fault injection simulation campaigns on a network of workstations (NoW).

To evaluate GemFI, we inject with faults a number of codes from different application domains with diverse characteristics. We focus on correlating the effects of faults in different architectural components with the particular characteristics of each application and its inherent error tolerance. Moreover, we study the relation of the timing of faults during the application life with the effect on the correctness of results.

The remainder of the paper is organized as follows. An overview of Gem5 simulator is given in Section II. Section III discusses the internal design and implementation of the tool and its usage. Section IV describes our experimental methodology and outlines the results of the experimental validation of the simulator. Section V focuses on the quantitative evaluation of GemFI performance. Section VI discusses previous work. Finally, Section VII concludes the paper.

II. The Gem5 Simulator

Gem5 is a popular open-source system simulator. It provides a modular platform for computer system-level architecture

[1]GemFI also supports Intel x86 ISA, however we discuss the Alpha ISA implementation due to the mature support of the Alpha architecture by GEM5.

research, encompassing system-level architecture as well as processor micro-architecture.

Object oriented design enhances the flexibility of Gem5. The ability to construct configurations from independent objects facilitates multicore and multi-system design. Moreover, Gem5 provides four different CPU models, each of them representing a different point in the speed vs simulation accuracy trade-off. *Atomic Simple* is a single IPC CPU model. *Timing Simple* is similar but also simulates the timing of memory references. *InOrder* is a pipelined in order CPU. Finally, *O3* is a pipelined out-of-order CPU model. Gem5 also supports two memory system models: *classic* and *ruby*. The classic is fast and easily configurable, while the ruby model provides a flexible infrastructure capable of accurately simulating a wide variety of cache coherence memory systems.

Gem5 operates in two modes: *System Call Emulation (SE)* and *Full System (FS)*. In SE mode applications execute on simulated "bare metal". Whenever the program executes a system call, Gem5 traps and emulates the call usually by passing it to the host OS. Currently there is no thread scheduler in SE mode. Therefore, threads are statically mapped to a core, hindering its use with multi-threaded applications. FS mode offers an environment for running an operating system (OS) on top of the simulator. There is support for interrupts, exceptions and I/O devices. Applications are executed under the control of the OS.

Gem5 supports a number of ISAs, including Alpha, MIPS, ARM, Power, SPARC and x86. The simulators modularity allows these different ISAs to be easily implemented on top of the generic CPU models and the memory system. Alpha is the most maturely supported ISA, with ARM and x86 following.

III. GemFI Design and Implementation

We extended Gem5 with fault injection capabilities, following the General Processor fault model described in [2]. The result, GemFI, is a configurable tool for studying the effect of faults in a processor.

GemFI was developed using C++ and Python. It fully supports the Alpha and Intel x86 ISAs. Supporting more instruction sets is rather straightforward, since the implementation of GemFI is fairly ISA-agnostic. GemFI supports full system simulation mode as well as the execution of multi-threaded applications. An architectural overview of GemFI is depicted in Fig. 1, whereas the following sections discuss its main features in more detail.

A. GemFI User Interface

GemFI provides an API consisted of two intrinsic functions.

- **void fi_activate_inst(int id)** is translated to a pseudo-assembly instruction. Its successive occurrences toggle (active/inactive) the manifestation of faults for the specific process/thread. The executing thread is assigned a numerical *id* which can be used as an identifier of the thread in fault injection configuration.
- **void fi_read_init_all()** checkpoints the simulation. Upon restoring from the checkpoint, it resets all the internal information of GemFI, allowing the same checkpoint to be used as a starting point for multiple experiments with potentially different fault injection configurations.

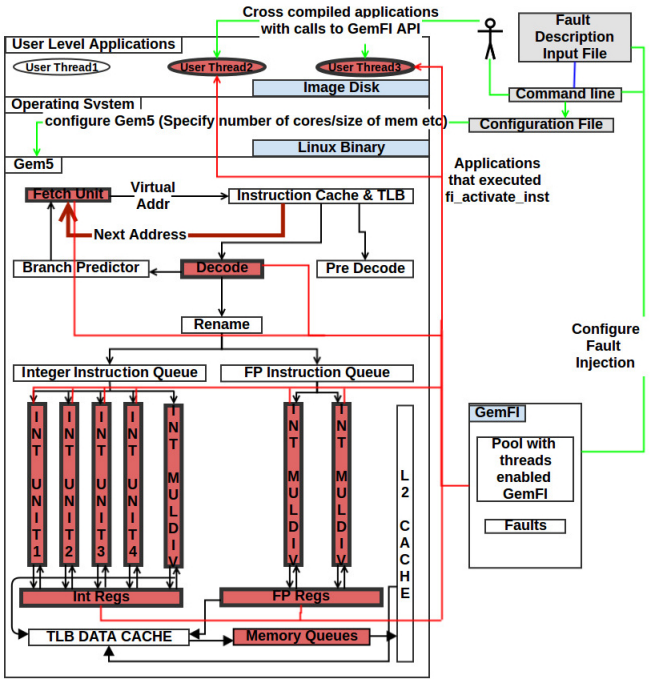

Fig. 1: An architectural overview of GemFI. The red components of the architecture demonstrate the possible locations where faults can be injected, whereas the red ovals represent applications which use the extended ISA.

On GemFI invocation the user also provides – at command line – an input file specifying the faults to be injected in the upcoming simulation. Each line of the input file describes the attributes of a single fault. Faults are characterized by four attributes: *Location*, *Thread*, *Time* and *Behavior*.

1) Location: Fault location specifies the micro-architectural modules to be targeted for fault injection. The user specifies the core, the module within the core and finally the specific bit location to be corrupted. Supported locations include registers (integer, floating point, special purpose), the fetched instruction, the selection of read/write registers during the decoding stage, the result of an instruction at the execution stage, the PC address and finally memory transactions (load/stores).

2) Thread: The thread attribute allows to selectively inject faults to specific threads, using the id assigned to the thread upon execution of *fi_activate_inst(id)* as an identifier.

3) Time: Another important aspect of the fault injection configuration is its timing. Timing is relative to a simulation milestone, marked by the execution of the *fi_activate_inst*. Faults are scheduled relatively to the number of instructions already executed, or to the number of elapsed simulation ticks of the targeted thread.

4) Behavior: The values of the specified faulty location can be corrupted in following ways:

- by assigning an immediate value provided by the user to the location.

```
"RegisterInjectedFault Inst:2457 Flip:21
    Threadid:0 system.cpu1 occ:1 int 1"
```

Listing 1: A sample input file to GemFI

```
#include <m5op.h>
int main(int argc, char *argv[]){
  int id = 0;
  initialize_input_data();
  fi_read_init_all();
  fi_activate_inst(id);
  foo();
  fi_activate_inst(id);
}
```

Listing 2: Modified source code of an application for fault injection.

- by XORing the running value at this location with a user-specified constant.
- by flipping the running value at bit locations. Multiple bit flips are supported by injecting multiple faults on the same module.
- by setting all bits of the location to a value of 0 or 1.

To emulate the behavior of transient and permanent faults, the user can define how long the fault is active in terms of the number of simulation ticks or number of instructions. For example, a fault injected in the execution stage of the processor can be injected continuously for the next N instructions (or for the next N simulation cycles) if so instructed by the user.

B. Simple Example

Listing 1 outlines a user-provided fault configuration example. The fault is injected in the 21^{st} bit of register R_1 of the CPU (location), when the application fetches the 2457^{th} instruction after the initiation of fault injection for this thread (*fi_activate_inst*). The fault is activated for a single instruction (occ:1) and only for the thread with id equal to 0.

The end user compiles (or cross-compiles) the application to be tested (Listing 2). Target applications must, at least, contain one call to initialize fault injection. Afterwards, the user moves the binaries into the disk image serving as the virtual disk of GemFI. Using the command line, the user provides a configuration file (Listing 1) describing all the faults to be injected in the simulation. After *fi_activate_inst(id)* is called, the thread identifier is stored in the internal data structures of GemFI. Simulation continues normally, until it is time for a fault to be injected. At that time, GemFI alters the state of the target hardware structure according to the fault specification in the configuration file.

C. GemFI Internals and Implementation

Fig. 2 demonstrates the main abstract steps executed by GemFI on each simulated served instruction.

Threads that have enabled fault injection are internally represented as instances of a class (*ThreadEnabledFault*),

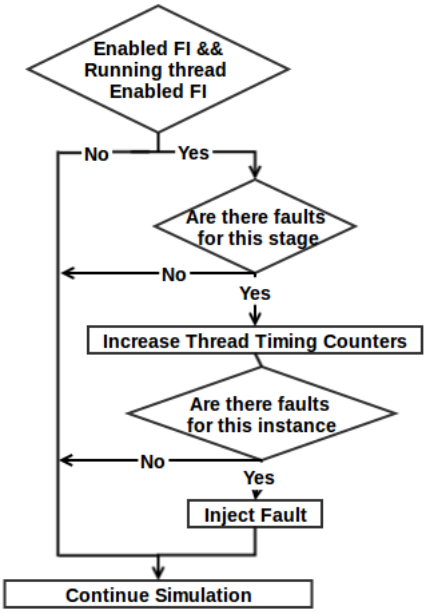

Fig. 2: GemFI functionality on each simulated instruction.

containing all per thread information necessary for fault injection, such as the number of instructions the thread has executed on each core. Each simulated core has a pointer to a *ThreadEnabledFault* object. If the thread executing on the core has not activated fault injection, the pointer is NULL. When a thread executes **fi_activate_inst()**, GemFI looks in a hash table to identify whether the specific thread has already activated fault injection. Threads are identified at the hardware/simulator level by their unique Process Control Block (PCB) address. If the thread is not found in the hash table, a new *ThreadEnabledFault* object is created and the running core is set to point to that object. On the other hand, if there was already an entry in the hash table, the invocation of **fi_activate_inst()** deactivates fault injection for the specific thread. The thread is removed from the hash table, the corresponding *ThreadEnabledFault* object is destroyed and the core's pointer is set to NULL. During context switches, which are identified by the change of the PCB address, GemFI checks whether the switched-in thread has activated fault injection, in order to properly set the core's pointer to the thread's *ThreadEnabledFault* object. Monitoring context switches allows GemFI to eliminate the overhead of checking the fault injection status of the executing thread in the hash table on each simulated clock tick.

Faults are described in the input file provided by the user at GemFI command line. The file is parsed at startup and each fault is inserted to one of five internal queues. Each queue corresponds to a different pipeline stage.

On each simulation tick, GemFI checks if fault injection has been enabled for the running thread. In such a case, it prefetches the corresponding *ThreadEnabledFault* objects. Then and for each instruction served at a pipeline stage, GemFI updates the thread's data and scans the corresponding queue for faults targeting the executing thread at the specific simulation point. Queue entries are sorted according to the

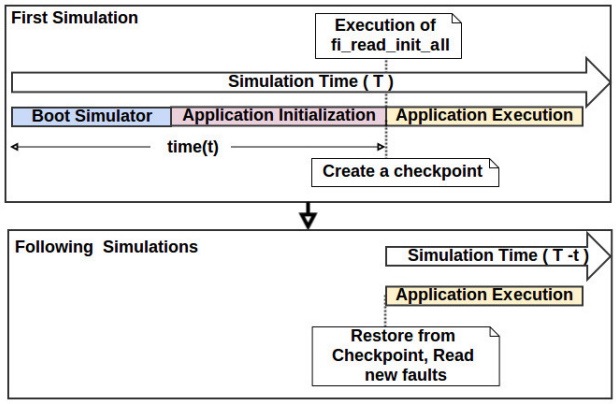

Fig. 3: Simple checkpoint-restore mechanism to speedup simulation campaigns.

timing of each fault. If such a fault is found, the value of the targeted location is corrupted according to fault's behavior.

D. Simulation Checkpointing

Checkpointing allows saving the state of a process or a system at a specific time snapshot and reverting to that later, to restart the execution from that point if needed. Checkpointing is necessary in order to avoid losing simulations in case of unexpected failures. It is particularly useful when simulation campaigns are executed to non-dedicated networks of workstations, a feature supported by GemFI.

Gem5 provides checkpointing, however with limitations. One method is to switch the simulation from O3 to atomic simple mode, create the checkpoint, and revert back to O3 mode to continue the simulation. This requires a pipeline flush, presenting a potential realism loss hazard. The second method requires simulating the MOESI hammer cache coherency protocol, which however dramatically increases simulation time.

We used DMTCP (Distributed MultiThreaded Checkpointing) [3] to checkpoint the state of the Linux process running the simulator, instead of checkpointing the internal state of the simulator. A feature of DMTCP is its ability to take checkpoints either by programmatically invoking checkpointing from within the process to be checkpointed, or asynchronously, by setting environment variables. The ability to invoke DMTCP from within the simulator allows us to exploit the front-end checkpointing mechanism of Gem5, while altering the checkpointing back-end to use the DMTCP API.

Apart from protecting against unexpected problems in simulation campaigns, checkpointing can be used to speed-up simulations. Before starting simulation campaigns, the user executes one simulation up to the point when fault injection is activated (including booting of the operating system and application initialization). Using GemFI's API the user can checkpoint the simulation at this point. The saved state is then used as a starting point for all experiments in the campaign (Fig. 3). Upon restoring a checkpoint GemFI parses again the faults configuration file. Therefore, this strategy allows fast-forwarding of the execution to the checkpoint and spawning of multiple experiments, with different fault injection configurations from that point on. As a result, the cumulative execution time of the simulation campaign is significantly reduced, as we demonstrate in Sec. V.

E. Simulation Campaigns on a Network Of Workstations

GemFI is accompanied by a set of shell scripts which facilitate launching simulation campaigns on a network of workstations (NoW). The workstations need to share a network file-system, in order to store the fault description files of the experiments, the simulation checkpoints and the output of each simulation. The main steps for parallel execution of simulation campaigns on a NoW are the following:

1) The configuration files for all experiments are stored on a network share.
2) A simulation is executed up to the point fault injection is activated and the simulator process is checkpointed. The checkpoint is stored to the share.
3) Each workstation gets a local copy of the checkpoint.
4) Each workstation checks the share for experiments to be executed. It selects one of the remaining experiments and executes it locally, starting from the checkpointed state.
5) Simulation results are moved from the workstation back to the network share.
6) Steps 4-6 are repeated until there are no experiments left.

IV. Validation

In order to validate the functional correctness of GemFI, we conducted an experimental study using a set of benchmark applications. Our simulator system was set to simulate a single core ALPHA CPU coupled with a tournament branch predictor, a L1 instruction cache and a L1 data cache and as a L2 cache we used a unified L2 cache.

DCT, is a kernel of JPEG image compression and decompression [4]. We applied each kernel on a gray-scale 512X512 image. *Jacobi* is applied on a diagonally dominant 64X64 matrix. *Monte Carlo PI* estimates the value of PI by randomly selecting 10^5 points within a unit square and evaluating whether they fall into the inscribed into a circle with radius one. *Knapsack* is a solution of the zero one knapsack combinational problem using a genetic algorithm. We use an input of 24 items and a weight limit of 500. The *Deblocking* filter is a kernel of the AVS video decoding process [5]. We apply it on a $720X240$ pixel image. *Canneal* is a benchmark of the PARSEC Benchmark Suite [6]. *Canneal* employs an annealing (SA) algorithm to minimize the routing cost of a chip design by randomly swapping netlist elements. It was applied on 100 nets, allowing up to 100 swaps in each step.

The number of executions of each application for every experiment varied from 2501 to 2504 and has been calculated using the method presented in [7], setting 99% as a target confidence level and 1% as the error margin.

A. Experimental Validation in the Absense of Faults

The execution of each application was simulated both with our tool and the original Gem5 simulator. When simulating using GemFI we did not inject any faults. We then compared

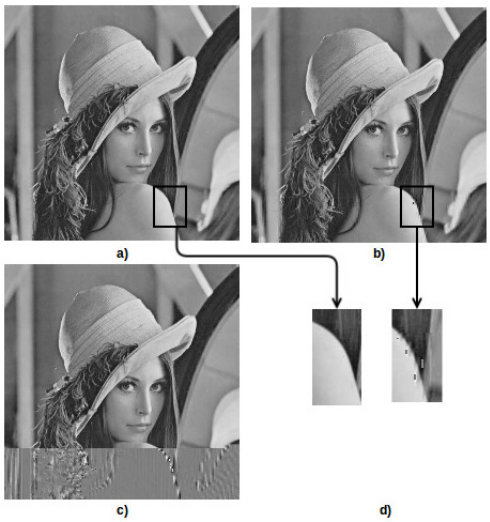

Fig. 4: Different categories of results for the DCT benchmark. a) A strict correct result b) Relaxed correct result c) SDC d) The difference between (a),(b) (loss of quality)

the application output from the two experiments, as well as the statistical results provided by the simulator. For all benchmarks the results were identical. This indicates that GemFI does not corrupt the simulation process.

B. Experimental Validation in the Presence of Faults

1) Methodology: As a next step, we launched simulation campaigns in which applications are injected with faults. We use a single event upset fault model. Each experiment injects a flip-bit fault, using a uniform distribution for the *Location*, *Time* and *Behavior*. Although this methodology does not necessarily represent the way faults affect systems, it is ample for the evaluation of the simulator. As mentioned earlier, GemFI can support any user-provided realistic fault model.

We initially checkpoint after the system boot-up and the initialization phase of the application under investigation. For each experiment in a campaign, we restore from the checkpoint, start simulating in O3 mode and inject the fault. The simulation continues until the affected instruction commits or squashes (for example, due to a branch misprediction). At that point we switch to atomic simulation and after application termination (normal or crash) we evaluate the quality of the end-result. When injecting a fault we print information on the affected assembly instruction. This information is used *postmortem* to correlate, either analytically or statistically, the fault with the simulation result.

The outcome of each experiment can be classified in the following categories: *crashed*, *non propagated*, *strictly correct* result, *correct* result and *SDC (Silent Data Corruption)*. *Crashed* are experiments which fail to successfully terminate. *Non propagated* are experiments in which faults did not manifest as errors (for example they were inserted in registers, however the corrupted register was either not used during the execution of the application or overwritten before the erroneous value was used). *Strictly correct* experiments produce results which are bit-wise identical to those produced by the corresponding

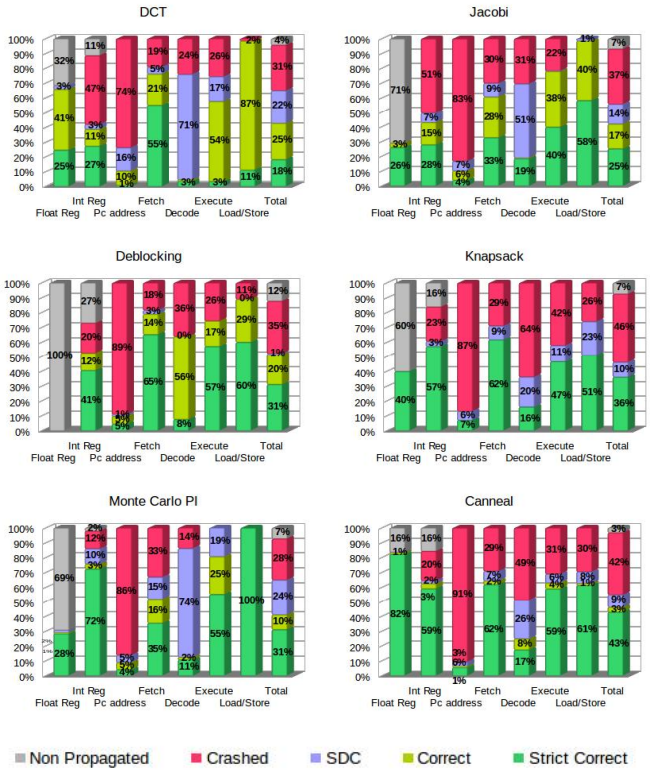

Fig. 5: Application behavior when fault injecting different architectural components.

error-less execution. *Correct* experiments produce results that are within acceptable quality margins, although not bit-wise identical to those of the error-less execution. The degree of tolerance is application dependent. For *DCT* we compare the produced compressed image with the uncompressed one used as input. Images with PSNR higher than 30 are regarded as correct, since typical PSNR values in lossy image and video compression range between 30 and 50 dB [8]. For the deblocking filter, outputs with PSNR higher than 80 dB, when compared with the error-free execution, are characterized as correct [8]. For *PI* estimation we accept experiments that have computed the first two decimal points correctly, since this the accuracy expected by the error-free execution for the 10^5 test points. Since the tolerance on *Jacobi* is highly dependent on the application domain, we characterize as correct solutions that result to the same (bit-exact) output as the golden model, converging after a potentially different number of iterations. Correct *Canneal* executions are those that reduce the total cost of routing and produce a correct chip. Finally, *SDC*s are executions that terminate normally, yet they produce results outside the acceptable range compared to the results of the error-free execution. Fig. 4 depicts an example of the different classes of results.

2) Experimental Results: Fig. 5 depicts the results of the fault injection campaigns, correlating the *Location* of the fault with application behavior. The last column of each chart summarizes the results for the specific application.

All applications demonstrate their highest resiliency to faults

targeting floating point registers. Most applications use a small subset of these registers, hence there is a low probability for a fault to affect a live register. Moreover, floating point registers are typically used to store data and not system state information or control flow information. *Deblocking*, a benchmark with no floating point operations, behaves exactly as expected, demonstrating 100% strict correctness.

On the other hand, faults on the integer register file result to higher crash rates. The compiler uses integer registers for storing important information (global pointer, stack pointer, frame pointer, return address register). Moreover compiler uses integer registers for control flow information (loop iterators, base addresses for memory translation). The integrity of these registers is crucial. Integer resisters tend to be live during large spans of the application life. Therefore, any fault affecting them has a high probability to cause a crash. For example *DCT* and *Jacobi* which are characterized by many memory accesses and use multi-level loop nests exhibit almost twice the crash rate compared with other applications.

In order to validate fault injection at the fetch stage, we correlated the affected bit location and the instruction type with the end result of the application. The analysis is ISA dependent; Table I summarizes Alpha instruction format. Experiments affecting unused bits always resulted into strict correct results. Faults affecting branch instructions were validated by checking the simulation statistical information. For example when inserting a fault into the *displacement* bits of the instruction and the branch is not taken the simulation statistics were the same and the end-result was categorized as strict correct. Faults affecting the *Ra* field may cause no error, should the result of the branch remain the same. Whenever faults altered the *displacement* field of memory instructions the application would crash with a high probability. The same was observed when the error altered the *Ra* value of a memory instruction, since the base address was read by another register. Finally we observed that, exactly as expected, when faults were injected into the *opcode* or the *function* and the resulting opcode/function is not implemented the benchmarks always terminated their execution due to illegal instruction.

A similar analysis was applied for faults inserted in the selection of registers during the decoding stage. Errors which affect the selection of the base of load/store instructions would usually cause a segmentation fault. An interesting observation is that faults inserted in the decoding stage of the *PI* algorithm result to crashes almost at half the probability compared with the remaining applications, because *PI* performs almost no data accesses from memory. Errors in the decoding stage usually lead to SDCs. This is expected, since operations are executed with different inputs. Correct results may be produced only by faults which alter a squashed instruction, or due to inherent, algorithmic application resiliency.

Faults introduced in the execution stage, which alter memory access instructions tended to result to crashes, because at this stage the virtual address of the memory transfer is being calculated. Faults altering the resulting address usually result to segmentation violations. The variation between the percentage of crashes among different applications is consistent with the variation of the percentage of memory operations in the instruction mix. In *Knapsack*, which makes heavy use of arrays and pointers 42% of faults in the execution stage result to crashes. On the other hand, *PI* evaluation, with almost no data accesses from memory, suffers almost no crashes. Correct and strictly correct results when fault injecting in the execution stage were found to be due to faults that have been masked during the remaining execution of the application, or faults that affected the less significant bits of data computations.

Faults altering the result of data loads/stores rarely resulted to crashes, and when they did it was because the error affected a store/load of an address. For example, altering the stored or loaded value of the return address usually led to crash. In total, errors affecting data store/load operations exhibit high resiliency, resulting to correct results in 78% of the cases.

Finally faults altering the value of the PC address were almost always fatal for the affected applications. Correct results were obtained in the few cases when the corrupted PC address was close to the correct one. This, in practice, corresponds to a small forward or backward jump.

Another interesting aspect of the experimental validation is the correlation of the timing of fault injection to the effects on the application. Fig. 6 depicts the results from three fault injections campaigns with interesting trends. The horizontal axis corresponds to the timing of fault injection normalized to the application execution time and the vertical axis corresponds to the fraction of experiments that resulted to each of the classes of outcomes. *Acceptable* represents the union of correct and strictly correct results.

For *Monte Carlo PI* estimation the time when fault injection took place appears to be uncorrelated with application behavior. This is reasonable, since the application iteratively produces random numbers, which are used to compute the final result. All iterations affect the final result similarly, therefore we did not expect different behavior with respect to the timing of the faults. On the other hand, *Knapsack* demonstrates a different behavior. The later the faults are injected, the more likely the results are acceptable. Faults corrupting data in a manner that does not result to values which converge towards the solution will be discarded on the following iteration, after applying the fitness function. This effect becomes more intense on each consecutive iteration of the algorithm. In *Jacobi*, faults inserted at the beginning of the execution tend to result to strict correctness. The later the faults are injected, the more the correct results at the expense of strictly correct. Given that the input matrix is diagonally dominant, errors which do not alter important variables of the application (etc. iterators) but alter input or intermediate data, will have no significant effect to the results, since the algorithm is bound to converge. However, more iterations may be needed to achieve convergence.

V. GemFi Performance Evaluation

In order to evaluate the overhead of GemFI we compare the execution time for simulated runs of all the aforementioned

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Type
Opcode						Ra					Rb					Unused			0	Function							Rc						Integer Operate
Opcode						Ra					Literal								1	Function							Rc						Integer Operate,Literal
Opcode						Ra					Rb					Function											Rc						Floating Point Operate
Opcode						Ra					Rb					Displacement																Memory Format
Opcode						Ra					Displacement																					Branch Format
Opcode						Function																										CALL_PAL Format

TABLE I: Alpha instruction formats

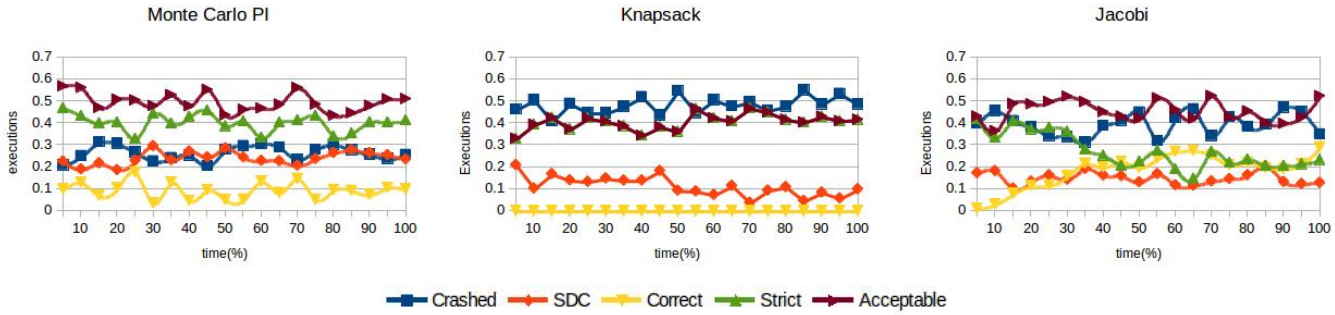

Fig. 6: Correlation of the timing of fault injection with the effect on the application.

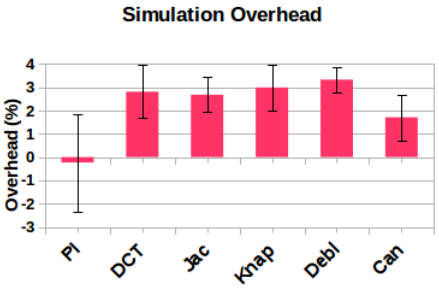

Fig. 7: GemFI average overhead compared with unmodified Gem5. The chart also depcits the 95% confidence interval for each application.

benchmarks on both GemFI and the unmodified Gem5 simulator. We measure and compare simulation time for the part of the application for which fault injection is active (between *fi_activate_inst()* calls). Despite activating the fault injection functionality, in this set of experiments we do not actually inject any faults in the GemFI simulations. Should we inject faults, the behavior of applications would potentially change, thus making the comparison between the two tools infeasible. It should be noted that, despite the fact that no faults are injected, all GemFI functionality is activated — especially the modules of GemFI that are executed on each simulated cycle, thus resulting to most overhead — apart from the last step of the process described in Fig. 2, the fault injection itself. However, the actual fault injection step would, in any case, be activated only once, with negligible overhead. Moreover, since no faults are injected, there are no opportunities to switch to atomic simple mode after fault manifestation, therefore the simulation is performed in the high-overhead O3 CPU model.

Fig. 7 depicts the experimental results, which can be considered as a worst-case overhead scenario for GemFI. The overhead varies from -0.1% to 3.3%. It is mainly dependent on the number of instructions simulated with fault injection enabled. The overhead introduced by GemFI clearly is minimal. For PI estimation GemFI appears to perform better than

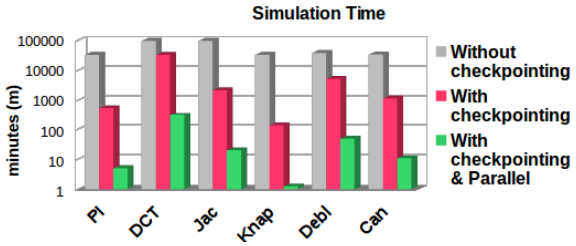

Fig. 8: Effect of GemFI optimizations on the execution time of fault injection campaigns (y-axis in logarithmic scale).

Gem5, however this observation is not statistically significant.

Using the checkpointing methodology presented in Sec. III-D, GemFI is able to significantly reduce the time for executing simulation campaigns. Fig. 8 summarizes the simulation time for the campaigns discussed in Sec. IV, with and without using the checkpointing capability to fast-forward the simulation to the point where fault injection is activated. The benefit from checkpointing is a 3x to 244x (64.5x on average) speedup with respect to the non fast-forwarded execution of the campaign. The speedup is mainly dependent on the ratio of the execution time spent for each application on the pre- and post-checkpoint code.

The third set of bars in Fig. 8 depicts the execution time of the simulation campaigns on a network of 27 workstations, using the meta-simulation infrastructure discussed in Sec. III-E. Each workstation is equipped with quad core Intel Xeon E5520 CPUs at 2.27 Ghz and 8 GB RAM each. On each workstation we execute simultaneously 4 experiments (simulations). The additional speedup, compared with execution on a simple system with checkpoint-based fast forwarding, is consistent with the number of simultaneously executed experiments (in all cases approximately 108x).

VI. Related Work

The impacts of faults have been evaluated by several research groups. Different approaches are used towards injecting

faults such as software fault injection, simulation fault injection and physical level fault injection.

RIFFLE [9] and MESSALINE [10] inject faults at the pin level, while FIAT [11] and FERRARI [11] implement software-level fault injection. Simulation based fault injection has the ability to model complex system with great accuracy however, ensuring that the simulated models are realistic and restraining simulation time are significant challenges. Examples of fault simulators are MEFISTO [12] and VERIFY [13]. MEFISTO inserts faults into VHDL models. This method provides high accuracy in both the location and the timing of the fault, as well as high validity of results. The drawback of using MEFISTO is the overhead of the system evaluation; the system is based in mutants [14], which is static information and the model has to be recompiled for each experiment. In VERIFY, another VHDL simulation-based fault injection toolset, basic logic gates (AND, OR, etc.) have been extended with extra signals, allowing to alter their behavior based on these external signals. Although the methodology is quite efficient in simulating different fault models, recompilation of the framework is required, introducing significant overhead.

Czeck and Siewiorek [15] performed a similar analysis through fault injection using bit-flip faults in their simulation model. However, their approach is limited to a specific configuration. Gaisler [16] injected faults into the register file of a SPARC V8 simulator, while enhancing the register file fault tolerance by adding ECC bits on each register. In [17] fault injection is performed on simulated processor similar to ALPHA 21264 or AMD ATHLON and the system tolerance is enhanced by providing extra hardware support.

In our work we provide a tool which is based on a broadly used reconfigurable simulator (Gem5). The purpose of our tool is to support any arbitrary fault model, by allowing the user to describe the faults to an input file. Moreover to the best of our knowledge GemFI is the first infrastructure that can target specific applications areas, while minimizing the changes to the original source code of the application under test.

VII. Conclusion

In this paper we introduced GemFI, a new simulator enabling fault injection of transient, intermittent and permanent faults. GemFI simulates unreliable environments in full system, cycle accurate mode. It is not limited to specific fault models, but is easily extensible and facilitates support of future fault models. Moreover, GemFI features such as checkpointing allow the execution of large-scale fault injection campaigns.

In order to validate GemFI and evaluate its performance, we executed a number of fault injection campaigns on different applications. We found that the outcome of fault injection campaigns was the expected one, according to the characteristics of each application. Moreover, the overhead of GemFI over Gem5 proved minimal, whereas the performance optimizations in GemFI have a profound effect on the execution time of simulation campaigns.

In the future, we plan to extend GemFI with fault injection capabilities outside the processor, namely on the processor / memory interconnect, as well as on external I/O devices. Moreover, we plan to enhance it with realistic fault models, associating the supply voltage (V_{dd}) with the error rate in different system components. Our goal is to study the limits of aggressively reducing power consumption at the expense of correctness, yet within the error tolerance of applications and the software stack.

Acknowledgment

This work has been partially supported by the EC within the 7th Framework Program under the FET-Open grant agreement SCoRPiO, grant number 323872.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[2] C. R. Yount and D. P. Siewiorek, "A methodology for the rapid injection of transient hardware errors," *IEEE Trans. on Computers*, 1996.

[3] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop Proc," in *Proc. of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[4] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG 2000 still image compression standard," *Proc. of the IEEE International Symposium on Signal Processing (ICSP)*, 2001.

[5] L. Fan, S. Ma, and F. Wu, "Overview of AVS video standard," in *Proc. of the IEEE International Conference on Multimedia and Expo (ICME)*, 2004.

[6] C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," in *Proc. of the Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2009.

[7] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: quantified error and confidence," in *Proc. of the Conference & Exhibition on Design, Automation & Test in Europe (DATE)*, 2009.

[8] S. T. Welstead, *Fractal and wavelet image compression techniques*. SPIE Optical Engineering Press, 1999.

[9] H. Madeira, M. Rela, F. Moreira, and J. G. Silva, "RIFLE: A general purpose pin-level fault injector," in *Proc. of the European Dependable Computing Conference (EDCC)*, 1994.

[10] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. on Software Engineering*, 1990.

[11] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Trans. on Computers*, 1990.

[12] E. Jenn, J. Arlat, M. Rimn, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool," in *Proc. of the Symposium on Fault-Tolerant Computing (FTCS)*, 1994.

[13] V. Sieh, O. Tschche, and F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions," in *Proc. of the Symposium on Fault-Tolerant Computing (FTCS)*, 1997.

[14] M. Rimén, J. Ohlsson, J. Karlsson, E. Jenn, and J. Arlat, "Design guidelines of a VHDL-based simulation tool for the validation of fault tolerance," ESPRIT Basic Research Project (PDCS-2), Tech. Rep., 1993.

[15] E. W. Czeck and D. P. Siewiorek, "Effects of transient gate-level faults on program behavior," in *Proc. of the International Symposium on Fault-Tolerant Computing (FTCS)*, 1990.

[16] J. Gaisler, "A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture." in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2002.

[17] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2004.