Acceleration of Simulated Fault Injection Using a Checkpoint Forwarding Technique

Jongwhoa Na and Dongwoo Lee

Simulated fault injection (SFI) is widely used to assess the effectiveness of fault tolerance mechanisms in safety-critical embedded systems (SCESs) because of its advantages such as controllability and observability. However, the long test time of SFI due to the large number of test cases and the complex simulation models of modern SCESs has been identified as a limiting factor. We present a method that can accelerate an SFI tool using a checkpoint forwarding (CF) technique. To evaluate the performance of CF-based SFI (CF-SFI), we have developed a CF mechanism using Verilog fault-injection tools and two systems under test (SUT): a single-core-based co-simulation model and a triple modular redundant co-simulation model. Both systems use the Verilog simulation model of the OpenRISC 1200 processor and can execute the embedded benchmarks from MiBench. We investigate the effectiveness of the CF mechanism and evaluate the two SUTs by measuring the test time as well as the failure rates. Compared to the SFI with no CF mechanism, the proposed CF-SFI approach reduces the test time of the two SUTs by 29%–45%.

Keywords: Simulated fault injection, Simulation acceleration, Checkpoint and forwarding, Triple modular redundant.

Manuscript received Mar. 4, 2016; revised Apr. 13, 2017; accepted Apr. 24, 2017.

Jongwhoa Na (corresponding author, jwna@hau.ac.kr) is with the Department of Electronics, Korea Aerospace University, Goyang, Rep. of Korea.

Dongwoo Lee (dongwoo81@gmail.com) is with the Avionics Research Institute of Korea Aerospace University, Goyang, Rep. of Korea.

I. Introduction

Because of the recent developments in convergence technologies (CTs), consumers can now enjoy various convenient functions; however, these CTs can also incur various types of unexpected accidents such as sudden unintended automobile accelerations, adverse events of robotic surgical systems, and drone accidents [1]–[4]. Thus, safety has become one of the key parameters in the design of safety-critical embedded systems (SCESs). In this regard, we must identify the various types of faults that may threaten the safety of SCESs [5]. From a technology perspective, the designers of SCESs should deal with soft errors when they have to use complex VLSI parts fabricated with a 32-nm technology process node [6]. Studies have shown that environmental issues such as cosmic and terrestrial radiations electromagnetic interference, and temperature may cause single-event effects and soft errors in varying degrees, depending on various conditions [7], [8]. One way of handling these faults and failures is to employ an optimal fault tolerance mechanism (FTM) during the development of the SCESs [9], [10]. Since the requirements of functionality and dependability of modern SCESs are increasingly complicated, it is desirable to evaluate both the functionality and the safety features of the FTM in SCESs in the early stages of the development lifecycle [11], [12].

Simulated fault injection (SFI) can be useful for the analysis of the FTM in the early stages of the development lifecycle, but its slow execution speed has limited its widespread use because of the large and complex test space [13]–[15]. To reduce the test time of SFI, one group of researchers developed various types of pruning techniques to reduce the fault space [16], [17]. A recent fault-space pruning technique using a genetic algorithm reduced the fault space of a target embedded system from

1.46×10^7 to 1.99×10^5 [18], [19]. Although this technique improves the performance of the fault-space pruning technique, the test times of SFI are still significant for any nontrivial modern SCES application; as a result, we need to investigate different approaches for reducing the test time, such as accelerating the SFI tool.

In this paper, we present a method that can accelerate SFI experiments using a checkpoint forwarding (CF) technique. Although the checkpoint technique is a popular fault-tolerant mechanism for recovery mechanisms, it is also an effective mechanism for reducing the simulation time by accelerating the SFI analysis [9], [10]. In addition, we integrate the CF mechanism with the SFI using a commercial Verilog simulator to show how CF-based SFI (CF-SFI) can reduce the fault-injection simulation time and provide more detailed insights than those of system-level analysis. In SFI, the simulator executes the instructions of the simulation target model as scheduled. The unique role of SFI is to inject faults and record the effects of their propagation in the system. This implies that we know the simulation states of the SFI before the time of fault injection. Thus, instead of executing the simulation instructions one by one, we can jump into the simulation state for the fault injection process. The impact of this acceleration can be significant if the target is complex, which means that the simulation time is long, or if the number of faults is too large for accurate analysis of the reliability of the target.

From our experience with SFI, when we performed fault-injection campaigns with a large number of faults and various injection locations, times, and complex co-simulation targets, we found that some parts of the SFI process were redundant [20], [21]. If we can remove the redundant process in the SFI campaign, we may be able to accelerate the test time of the SFI. This requires constructing the checkpoints and a snapshot image file of the repeating states, which can be built during the golden-run stages of the SFI. During the injection campaign, if we can identify the repeating processes, we may be able to advance the state of the simulation to the predetermined checkpoints so that we can save some SFI execution time.

To evaluate the performance of the CF-SFI, we developed a CF mechanism using Verilog Procedural Interface (VPI)-based fault injection tools [22]. For the co-simulation target, we built two systems under test (SUT): a single-core-based co-simulation model and a triple modular redundant (TMR) processor-based co-simulation model. Both models used the Verilog simulation model of the OpenRISC 1200 processor and could execute the embedded benchmarks from MiBench [19], [23]. We investigated the effectiveness of the CF mechanism by performing both an SFI and a CF-SFI for the two SUTs. When we compared the result with the native SFI without the CF mechanism, we found that we could reduce the test time of the two co-simulation models with the CF-SIF mechanism by 29% and 45%, respectively.

The rest of this paper is organized as follows. In Section II, we present the techniques for accelerating the SFIs. In Section III, we explain the CF mechanism implemented with the NC-Verilog simulator from Cadence. In Section IV, we discuss the experimental results of the SFIs on the two types of co-simulation model to analyze the performance improvement due to the CF scheme. Finally, in Section V, we conclude the paper.

II. Overview of the Acceleration of the Fault Injection Techniques

Using SFI, we can evaluate the reliability of the fault tolerance mechanism of the target embedded system and find the vulnerability of the system in the early stage of the development process [13]. However, the slow simulation speed of SFI poses a problem because of the large number of test cases and the complexity of the simulation models. In the literature, a single fault-injection test of the AMD Bulldozer processor simulation model took 57 min [24]. Since the number of test cases of the statistical failure rate analysis of modern embedded systems exceeds 10,000, the test time becomes prohibitive [24].

To reduce the simulation time of the SFI, we investigated the works of two groups of researchers. One group tried to reduce the fault space [16], [17]. Including the definition/use analysis technique, various pruning algorithms to reduce the fault space were proposed [25]–[29]. To reduce the fault space further while improving the accuracy at the same time, some researchers introduced heuristic pruning techniques [30]–[32]. Recently, a fault-space pruning technique augmented with a genetic algorithm was proposed to reduce the fault space [18]. Although this pruning technique can reduce the fault space, it requires static/dynamic analysis, which may limit its applicability [30].

The second group investigated tool acceleration to reduce the simulation time, wherein the tool identifies and removes unnecessary or redundant operations of the SFI [20], [21]. One such approach is to use an efficient SFI management strategy, which uses a CF mechanism. Although the checkpoint-and-rollback technique is a well-known fault-tolerant mechanism, it is also an effective mechanism that can advance the state of a system to save some execution time.

In general, most of the fault injection experiments involve the use of a large number of identical simulation models with thousands of different fault models. When observing the internal process of fault injection simulations, one may find that some parts of the injection simulation are repeatedly re-executed in the simulations of different fault models. On the basis of these observations, researchers proposed using the CF technique to accelerate the execution time.

In [27], a checkpoint mechanism for SFI was introduced, but details about the experiments were not reported. In [33], a checkpoint algorithm for fault injection was proposed for two fault injection implementations: one is modified library error injection (MODLIB) and the other is PLI fault injection (PLInject), which is a statistical fault injection technique but whose simulation results are reported without actually injecting faults. In addition, the CF mechanism was studied to accelerate the execution of virtual systems and of the application software in distributed OS environments as well [34], [35]. Recent studies on checkpoint mechanism were targeted at operating-system level or system level and therefore are not applicable to the register transfer level (RTL) or gate-level fault-failure analysis. We summarize the acceleration tools for fault injection and their performance improvement according to the level of the target model in Table 1.

In this paper, we explain the practical implementation of the CF mechanism with a commercial Verilog simulator to show how the CF-SFI can reduce the test time and minimize the overhead of the CF mechanism. Note that Verilog simulators support checkpoint commands, which can be used for Verilog simulation forwarding purposes. As will be explained in the next section, we implemented the CF-SFI using the UST (User-defined System Task) service of the Verilog simulator. Since the UST function cannot manipulate the critical or exclusive system variables of the Verilog simulator, such as the simulator time, we excluded the use of the checkpoint commands of the simulator for the CF-SFI.

Table 1. Comparison of the simulation acceleration tools.

Model level	Tool	Acceleration method	Target	Improvement
RTL/Gate level	Our Research	Checkpoint SFI	OR1200 RTL	~1.8 ×
Architecture level	Relyzer [30]	Fault Pruning	Sparc V9	2 – 6 ×
	Smart Injector [32]		SimpleScalar	66 ×
System level	GemFI [36]	Checkpoint (DMTCP)	Gem5	64.5 ×
OS level	CFI [34]		Virtual Machine	8 ×

III. Checkpoint-Forwarding-Based Simulated Fault Injection

The CF-SFI technique reduces the simulation time by advancing the simulation clock cycles to the checkpoint by loading a snapshot image file of the corresponding simulation state. Although the forwarding method can reduce the simulation time, it also incurs overheads such as the snapshot image loading time, which is due to the slow disk I/O time. In Section III.1, we explain the implementation of the CF SFI using the VPI of the commercial Verilog simulator. Then, in Section III.2, we present a CF management strategy for minimizing the overhead and maximizing the efficiency.

1. Implementation of the Checkpoint Forwarding Function

We developed the checkpoint forwarding (CF) function using the user system task (UST) service of the Verilog procedure interface (VPI), which is explained in the IEEE 1364 programming language interface standard [22]. We developed three UST functions: 1) $checkpoint-create UST for creating a snapshot image

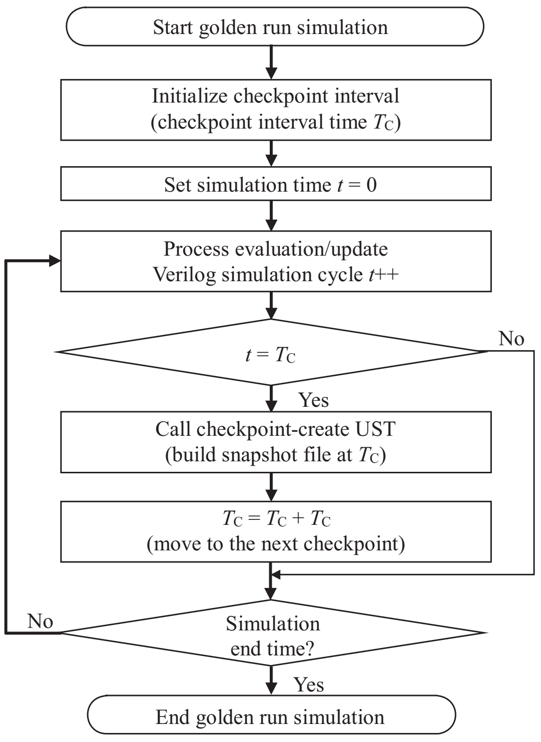

Start golden run simulation
Initialize checkpoint interval (checkpoint interval time T_C)
Set simulation time $t = 0$
Process evaluation/update Verilog simulation cycle t++
$t = T_C$
No
Yes
Call checkpoint-create UST (build snapshot file at T_C)
$T_C = T_C + T_C$ (move to the next checkpoint)
No
Simulation end time?
Yes
End golden run simulation

Fig. 1. Flowchart of the golden-run simulation.

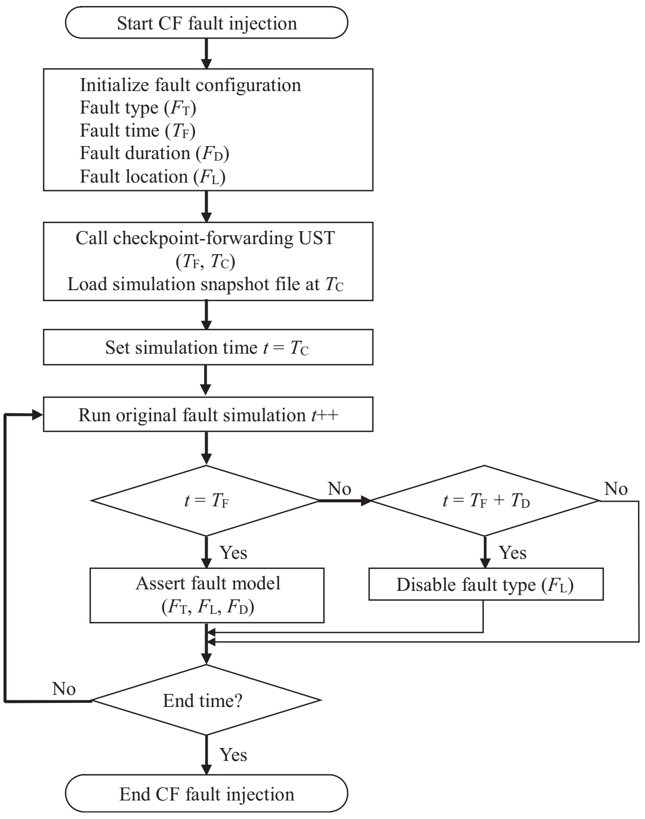

Start CF fault injection
Initialize fault configuration
Fault type (F_T)
Fault time (T_F)
Fault duration (F_D)
Fault location (F_L)
Call checkpoint-forwarding UST (T_F, T_C)
Load simulation snapshot file at T_C
Set simulation time $t = T_C$
Run original fault simulation t++
$t = T_F$
No
$t = T_F + T_D$
No
Yes
Assert fault model (F_T, F_L, F_D)
Yes
Disable fault type (F_L)
No
End time?
Yes
End CF fault injection

Fig. 2. Flowchart of the checkpoint forwarding fault injection simulation.

file, 2) $checkpoint-forwarding UST for forwarding to the checkpoint, and 3) $fault-injection UST for injecting faults into the simulation model. We explain the function of these three UST in Table 2.

We now explain the golden-run simulation (Fig. 1) and the fault injection simulation (Fig. 2) using the checkpoint forwarding mechanism. Firstly, during the golden-run simulation, we call $checkpoint-create UST to create the checkpoint snapshot file of the related state variables of the system at the checkpoints as follows:

Step 1. Initialize the checkpoint interval (T_C) and the start of the golden-run simulation.

Step 2. At the selected checkpoint time, call the checkpoint-create callback functions to build the snapshot file.

Step 3. Resume the golden-run simulation and update the simulator time.

Step 4. Repeat steps 2 and 3 until the end of the Verilog simulation.

This process of creating the checkpoint snapshot image files at each of the checkpoints continues until it reaches the final checkpoint in the golden-run simulation as illustrated below.

After the golden-run simulation, we performed the fault injection simulation using the fault models with various

Table 2. Summary of the user system task.

User system task		
Name	Input/output	Function
$checkpoint-create	Input: Checkpoint interval Output: Snapshot files	- Identify simulation objects (module, reg, net) of target Verilog model - Calculate checkpoint time using checkpoint interval - Build snapshot files
$checkpoint-forwarding	Inputs: Snapshot file Fault injection time Checkpoint interval	- Identify simulation objects of target Verilog simulation model - Calculate forwarding time - Update simulator state variables using the corresponding snapshot file
$fault-injection	Inputs: Fault model (time, type, location, duration) Output: Fault injection VCD	- Process original Verilog sim. - If (simulation time = fault time) then {replace faulty location with fault type hold fault type during fault duration}

fault attributes. During the fault injection simulation, the Verilog simulator calls the $checkpoint-forwarding UST function as follows:

Step 1. Before the start of the simulation, the simulator checks for the fault inject time.

Step 2. It searches for the nearest checkpoints just before the injection time.

Step 3. The snapshot file of the selected checkpoint is loaded to forward the simulation time.

After loading the snapshot file, the simulator restarts the normal simulation. When the simulator reaches the fault injection time, it calls $fault-injection UST to inject the fault into the simulation target model. The $fault-injection UST function performs the fault injection operation as follows:

Step 1. During the evaluation phase of the Verilog simulator, it identifies the attributes of the fault model.

Step 2. During the update phase of the Verilog simulator, it modifies the original value of the target variable into a fault value.

After the $fault-injection UST function executes, the simulator resumes the simulation. In particular, if the fault model is a transient fault model and it passes the fault holding time, then it restores the original value of the fault variable. Using the parameters in Table 3, we can summarize the fault injection procedure as follows.

Table 3. Summary of the variables for checkpoint forwarding.

Symbol	Meaning
N_T	Total number of clock cycles in a fault injection simulation
N_C	Number of clock cycles in a checkpoint interval
T_0	Unit clock cycle time
T_C	Checkpoint interval time
T_F	Fault injection time
T_L	Snapshot file loading time
T_{BC}	Simulation time *before* the selected *checkpoint*
T_{AC}	Simulation time *after* the selected *checkpoint*

2. Optimization of the CF Technique

A. Conditions for the Checkpoint Trigger

Using the checkpoint and forwarding mechanism during SFI, we can reduce the simulation time *before checkpoint* (T_{BC}), which is the time from the starting point to the selected checkpoint as shown in Fig. 3. However, this introduces an overhead time that is the snapshot file loading time (T_L). This implies that the exploitation of the performance improvement of the CF-SFI is only possible when the snapshot loading time (T_L) is smaller than the forwarded simulation time (T_{BC}). Thus, we may set $T_L < T_{BC}$ as a triggering condition of the CF mechanism in SFI.

For the operation of the CF mechanism, we build a snapshot image at every checkpoint interval during the golden-run simulation stage. We can record the average loading time of the snapshot files (T_L) at every checkpoint. After the golden-run simulation, we perform the fault injection simulation to calculate T_{BC} using the performance parameters from Table 3. From Fig. 3, for a given fault injection time (T_F), the CF mechanism forwards the simulation time from the initial point to the nearest checkpoint, which is the floor function of $\lfloor \frac{T_F}{T_C} \rfloor$, where T_C is the number of clock cycles for a checkpoint interval. Thus, T_{BC} can be expressed in time units as follows:

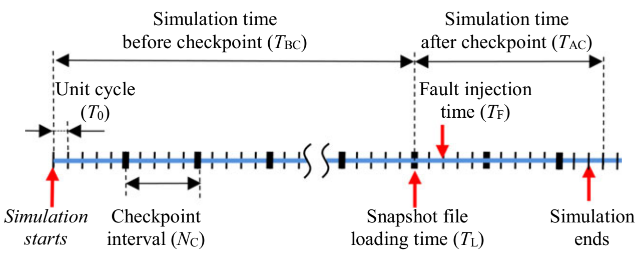

Simulation time before checkpoint (T_{BC})
Simulation time after checkpoint (T_{AC})
Unit cycle (T_0)
Fault injection time (T_F)
Simulation starts
Checkpoint interval (N_C)
Snapshot file loading time (T_L)
Simulation ends

Fig. 3. Design parameters of the checkpoint forwarding (CF) mechanism.

$$T_{BC} = \left\lfloor \frac{T_F}{T_C} \right\rfloor \cdot N_C \cdot T_0. \tag{1}$$

Substituting T_{BC} into the triggering condition, we find the following condition for the activation of the CF mechanism:

$$T_L < \left\lfloor \frac{T_F}{T_C} \right\rfloor \cdot N_C \cdot T_0. \tag{2}$$

We can exploit the advantages of the CF-SFI if this inequality is satisfied; otherwise, we disable the CF mechanism and perform the baseline SFI (without the CF mechanism).

For a given injection simulation, we can calculate the checkpoint interval T_C and the average clock cycle time T_0. Moreover, we can find the fault injection time T_F before the beginning of each fault injection simulation. Thus, at the starting time of each CF-SFI execution, we can calculate the simulation time before the checkpoint T_{BC} and decide whether or not to use the CF mechanism.

B. Assessment of the Checkpoint Overhead

In this subsection, we evaluate the relationship between the execution time of the fault injection simulation without the CF mechanism (T_{BL}) and the time of the simulation with the CF mechanism (T_{CF}). In Fig. 4, the baseline simulation time (T_{BL}) is the sum of the simulation time before the selected checkpoint (T_{BC}) and the simulation time after the checkpoint under the control of the CF mechanism (T_{AC}):

$$T_{BL} = T_{BC} + T_{AC}. \tag{3}$$

Moreover, the simulation runtime of the CF function (T_{CF}) is the sum of the snapshot loading time (T_L) and the actual simulation time after the checkpoint (T_{AC}):

$$T_{CF} = T_L + T_{AC}. \tag{4}$$

From Fig. 4, for brevity, we define the crossing point of T_{BL} and T_{CF} as the checkpoint forwarding tipping point (CFTP). With the CFTP as the center, we can divide the graph into an overhead area (left region) and a time-saved area (right region). In the blue-colored region, the amount of time saved by the CF mechanism can be formulated as follows:

$$T_{BL} - T_{CF} = T_{BC} - T_L = \left\lfloor \frac{T_F}{T_C} \right\rfloor \cdot N_C \cdot \; T_0 - T_L > 0. \tag{5}$$

By comparing the two graphs in Fig. 4, we can see that the time savings of the CF mechanism increase as the

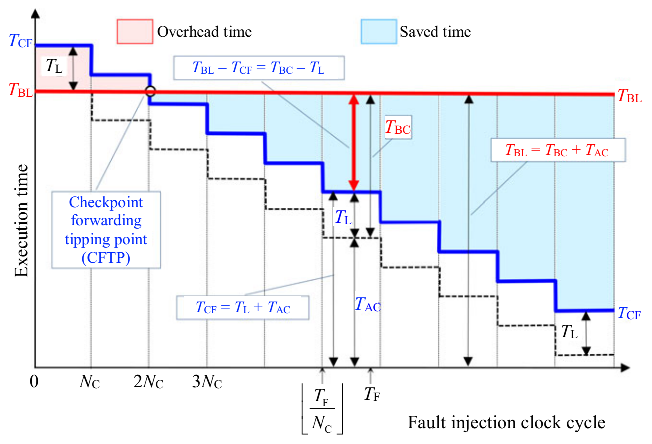


Fig. 4. T_{CF} represents the simulation time of fault injection with the CF mechanism, whereas T_{BL} represents the normal simulation time. The checkpoint forwarding tipping point (CFTP) acts as a reference point between the overhead region and the accelerated region for the CF mechanism in the simulated fault injection (CF-SFI).

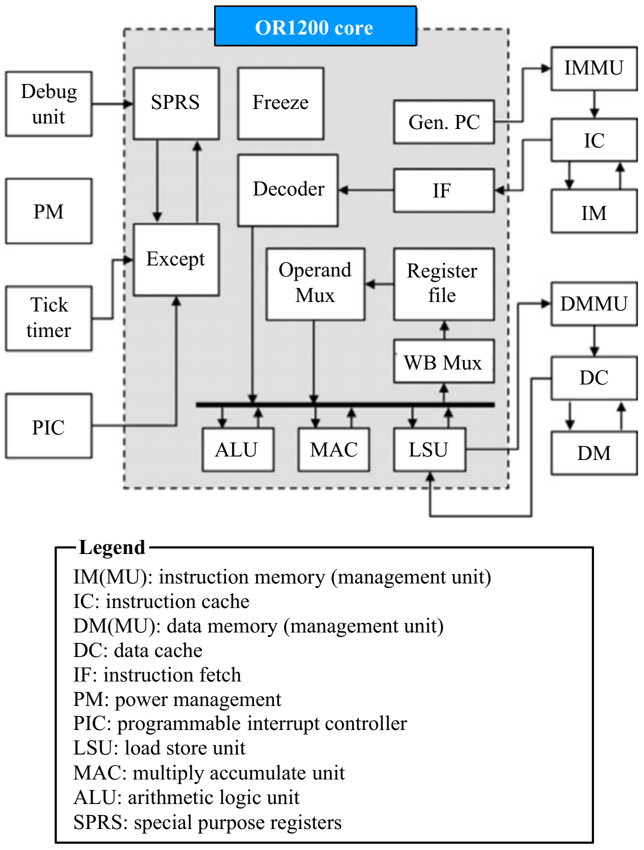
injection time T_F increases or the snapshot file loading time T_L decreases.

IV. Checkpoint Forwarding Simulation Results

To evaluate the performance of the CF-SFI, we designed the CF-SFI using the NC-Verilog simulator from Cadence. Moreover, we built two co-simulation models, a single-core-based hardware one and a TMR-based one, both of which were based on the Verilog simulation model of the OpenRISC 1200 processor (Fig. 5) and the fast Fourier transform (FFT) software from Mibench embedded benchmark suites [19], [23]. The OpenRISC OR12K processors used 5-stage pipelines, and supports virtual memory and DSP. We performed the CF-SFI experiments on a Linux server with a 2.33-GHz Intel Xeon E5345 processor.

1. Checkpoint Forwarding Simulated Fault Injection

As a first step, we performed a golden-run simulation to build the snapshot file at the checkpoint intervals and to find the value of the key parameters of the CF mechanism for the given fault injection campaign. During the golden fault-injection simulation run using the single-core co-simulation model, we measured the simulation clock cycles to be 705,000. Thus, assuming that a single clock cycle of the Verilog simulator was 20 ns, the baseline simulation time for the co-simulation model would be 14.1 ms. Executing this Verilog simulation on the Linux server, we measured the average value of 100 physical simulation times to be 10.24 s.



Legend
IM(MU): instruction memory (management unit)
IC: instruction cache
DM(MU): data memory (management unit)
DC: data cache
IF: instruction fetch
PM: power management
PIC: programmable interrupt controller
LSU: load store unit
MAC: multiply accumulate unit
ALU: arithmetic logic unit
SPRS: special purpose registers

Fig. 5. OpenRISC 1200 Architecture.

In the second phase, using these checkpoint snapshot files from the golden-run simulation, we performed the CF-SFI with the baseline co-simulation models to measure the performance of the CF mechanism.

For the fault model, we selected single transient stuck-at-1 faults since the objective of this study is to find the effectiveness of the checkpoint and forwarding mechanism in the fault injection campaign. Also, note that the stuck-at-1 fault model is more crucial for the simulation targets including processors because the data path in a processor uses more 0s than 1s [20].

We injected 100 random transient stuck-at-1 faults, which is a sufficiently large number of faults to measure the average simulation time of the model with the CF-SFI. In Table 4, we summarize the size and number of checkpoint snapshot files for four checkpoint periods: 0.5 ms, 1.0 ms, 1.5 ms, and 2.0 ms. For the baseline co-simulation model, we compared the execution time of each CF-SFI with that of the SFI without the CF mechanism and found that we could improve the CF mechanism runtime by 26%–29% over the baseline simulation. From the table, we can see that, as the checkpoint interval increases, the number and size of the snapshot files increase and, thus, the time saved by the CF-SFI decreases.

Table 4. Simulation parameters for the baseline injection and checkpoint forwarding injection.

SFI types	Checkpoint period (ms)	Number of snapshots	Snapshot size (MB)	Average runtime (s)	Improvements of CF-SFI over baseline SFI (%)
Baseline SFI	N/A	N/A	N/A	10.53	N/A
CF SFI	0.5	28	106.4	7.44	29.3
	1.0	14	53.2	7.55	28.3
	1.5	9	34.2	7.71	26.8
	2.0	7	26.6	7.78	26.1

2. Simulated Fault Injection Using CFTP

We now compare the simulation runtime of the CF mechanism with the CFTP to that of the SFI without the CF mechanism and the CF-SFI without CFTP.

Firstly, we calculate the CFTP of the single-core and TMR co-simulation models. From (2), to calculate the CFTP, we must calculate T_0, which requires the physical time value of the unit clock cycle, which was set to 20 ns in the Verilog simulation. We can calculate T_0 in two steps: 1) measure the time to execute a block of code and 2) divide it by the number of clock cycles in the block.

In our experiments, the average physical execution time of the CF SFI for 500 clock cycles was measured as 6.6688 ms. Therefore, the physical time of the clock cycles T_0 was 6.6688/500=0.013 ms/cycle. During the same simulations, we measured the average snapshot file loading time T_L as 1,120 ms or 86,153 clock cycles. By substituting these values into (5), we can determine the CFTP; we summarize its value for the four checkpoint intervals in Table 5.

For the CF-SFI of the single-core and TMR co-simulation models with the single transient stuck-at-1 fault model, we measured T_0, T_C, and T_L and calculated the CFTP, as given in Table 6. From the table, we can see that the execution time of the CF-SFI over the baseline SFI was improved by 28% for the single-core co-simulation model and 42% for the TMR model, respectively. This shows that the CF mechanism can be more effective as the size of the co-simulation model increases. Note that, in our experiments, we employed rather simple co-simulation models that ran for 705,000 simulation clock cycles because our purpose was to evaluate the effectiveness of the CF mechanism in the context of a simulated fault injection campaign. Thus, if we extend the complexity of our simulation target models, we can expect the impact of the CF mechanism to increase accordingly. In the case of the CFTP, the performance increase of the single-core model only showed a 1% difference, whereas that of the TMR model showed a 4% improvement. As in the case of the CF mechanism, we can expect the performance of the CFTP to be fully exploited if the complexity of the simulation models is greater.

Table 5. Calculation of CFTP using (5) for T_0 = 0.0133 ms and T_L = 1,120 ms at four checkpoint intervals of 0.5 ms, 1.0 ms, 1.5 ms, and 2.0 ms.

Checkpoint Interval (N_C)	ms	0.5	1.0	1.5	2.0
	Clock cycles	25,000	50,000	75,000	100,000
$\frac{1}{N_C}\frac{T_L}{T_0}$		3.36	1.68	1.12	0.84
$\frac{T_F}{T_C}$ = CFTP		4	2	2	1

Table 6. Comparison of the native SFI, CF-SFI without CFTP, and CF-SFI with CFTP (CFTP* in checkpoint interval number).

Co-simulation model	SFI type	Snapshot file load time T_L (ms)	CFTP*	Simulation runtime (s)	Improvements over baseline (%)
Single-core OR1200	baseline	N/A	N/A	10.53	N/A
	CF	1,120	N/A	7.55	28.3
	CFTP		2	7.45	29.2
TMR OR1200	baseline	N/A	N/A	23.09	N/A
	CF	3,483	N/A	13.38	42.1
	CFTP		3	12.54	45.7

V. Conclusions

We presented a method for accelerating simulated fault injection experiments by utilizing the checkpoints and the checkpoint forwarding (CF) technique. To evaluate the performance of the CF-based simulated fault injection (CF-SFI) technique, we implemented the CF-SFI using the User-defined System Task (UST) service of the NC-Verilog simulator from Cadence. Moreover, we implemented two co-simulation models, a single-core-based hardware one and a TMR-based one, both of which consisted of the Verilog simulation model of the OpenRISC 1200 processor, the FFT software from MiBench, and a transient stuck-at-1 fault model. From the simulation results, we confirmed that CF-SFI can improve the simulation time by 45% when compared with the baseline SFI without the CF mechanism. If we would integrate the advanced pruning algorithm with the CF mechanism in the simulated fault injection, we might be able to further reduce the test time of a simulated fault injection campaign.

Acknowledgement

This work was supported by Aerospace Component Technology Development Project funded by the Ministry of Trade, Industry and Energy (MOTIE, Korea) [Development of L-Shape Integrated Multi-Function Air Data System for the Supersonic Aircraft: 10067079].

References

[1] D.W. King et al., "UAV Failure Rate Criteria for Equivalent Level of Safety," *Int. Helicopter Safety Symp.*, Montréal, Canada, Sept. 26–29, 2005, pp. 1–9.

[2] R. Schaefer, "Unmanned Aerial Vehicle Reliability Study," Office of the Secretary of Defense, Washington, D.C., USA, 2003.

[3] M.L. Shooman, "Bohr Bugs, Mandel Bugs, Exhaustive Testing and Unintended Automobile Acceleration," *Int. Conf. Soft. Rel. Eng.*, Dallas, TX, USA, Nov. 27–30, 2012, pp. 5–6.

[4] H. Alemzadeh et al., "Systems-Theoretic Safety Assessment of Robotic Telesurgical Systems," *Int. Conf. SAFECOMP 2015*, Delft, Netherlands, Sept. 23–25, 2015, pp. 213–227.

[5] K. Potiron et al., *From Fault Classification to Fault Tolerance for Multi-agent Systems*, London, UK: Springer, 2013.

[6] S. Borkar, "Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, Nov. 2005, pp. 10–16.

[7] S. Mukherjee, *Architecture Design for Soft Errors*, San Francisco, CA, USA: Morgan Kaufmann, 2011.

[8] R. Velazco et al., *Radiation Effects on Embedded Systems*, Netherland: Springer Science & Business Media, 2007.

[9] I. Koren et al., *Fault-Tolerant Systems*, San Francisco, CA, USA: Morgan Kaufmann, 2010.

[10] D.J. Sorin, *Fault Tolerant Computer Architecture*, San Rafael, CA, USA: Morgan and Claypool, 2009, pp. 1–104.

[11] A. Moniruzzaman et al., "Comparative Study on Agile Software Development Methodologies," *Global J. Comput. Sci. Technol.*, vol. 13, no. 7, July 2013, pp. 5–18.

[12] C. Ebert et al., "Embedded Software: Facts, Figures, and Future," *Comput.*, vol. 42, no. 4, Apr. 2009, pp. 42–52.

[13] A. Benso et al., *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, NY, USA: Springer Science & Business Media, 2003.

[14] M. Kooli et al., "A Survey on Simulation-Based Fault Injection Tools for Complex Systems," *Int. Conf. Des. Technol. Integr. Syst.*, Santorini, Greece, May 6–8, 2014, pp. 1–6.

[15] H. Ziade, R. Ayoubi, and R. Velazco, "A Survey on Fault Injection Techniques," *Int. Arab J. Inform. Technol.*, vol. 1, no. 2, July 2004, pp. 171–186.

[16] P. Maistri and R. Leveugle, "Towards Automated Fault Pruning with Petri Nets," *On-line Testing Symp.*, Lisbon, Portugal, June 24–26, 2009, pp. 41–46.

[17] M. Maniatakos et al., "AVF Analysis Acceleration via Hierarchical Fault Pruning," *Eur. Test Symp.*, Trondheim, Norway, May 23–27, 2011, pp. 87–92.

[18] H. Schirmeier, C. Brochert, and O. Spinczyk, "Rapid Fault-Space Exploration by Evolutionary Pruning," *Int. Conf. SAFECOMP 2014*, Florence, Italy, Sept. 8–12, 2014, pp. 17–32.

[19] M.R. Guthaus et al., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *IEEE Int. Workshop Workload Characterization*, Austin, TX, USA, Dec. 2, 2001, pp. 3–14.

[20] D. Lee and J. Na, "A Novel Simulation Fault Injection Method for Dependability Analysis," *IEEE Des. Test Comput.*, vol. 26, no. 6, Nov. 2009, pp. 50–60.

[21] J. Na and D. Lee, "Simulated Fault Injection using Simulator Modification Technique," *ETRI J.*, vol. 33, no. 1, Feb. 2011, pp. 50–59.

[22] C. Dawson, S.K. Pattanam, and D. Roberts, "The Verilog Procedural Interface for the Verilog Hardware Description Language," *IEEE Int. Verilog HDL Conf.*, Santa Clara, CA, USA, Feb. 1996, pp. 17–23.

[23] D. Mattsson and M. Christensson, "*Evaluation of Synthesizable CPU Cores*," M.S. thesis, Dept. Comput. Eng., Chalmers Univ., Sweden, 2004.

[24] C. Constantinescu, M. Butler, and C. Weller, "Error Injection-Based Study of Soft Error Propagation in AMD Bulldozer Microprocessor Module," *IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Boston, MA, USA, June 25–28, 2012, pp. 1–6.

[25] D.T. Smith et al., "A Method to Determine Equivalent Fault Classes for Permanent and Transient Faults," *Rel. Maintainability Symp.*, Washington, D.C., USA, Jan. 16–19, 1995, pp. 418–424.

[26] A. Benso et al., "Fault-List Collapsing for Fault-Injection Experiments," *Rel. Maintainability, Symp.*, Anaheim, CA, USA, Jan. 19–22, 1998, pp. 383–388.

[27] L. Berrojo et al., "New Techniques for Speeding-up Fault-Injection Campaigns," *Des. Autom. Test Europe*

Conf. Exhibition, Paris, France, Mar. 4–8, 2002, pp. 847–852.

[28] R. Barbosa et al., “Assembly-Level Pre-injection Analysis for Improving Fault Injection Efficiency,” *Int. Conf. Eur. Dependable Comput.*, Budapest, Hungary, Apr. 20–22, 2005, pp. 246–262.

[29] J. Grinschgl et al., “Efficient Fault Emulation based on Post-injection Fault Effect Analysis (PIFEA),” *Int. Midwest Sypm. Conf. Circuits Syst.*, Boise, ID, USA, Aug. 5–8, 2012, pp. 526–529.

[30] S.K.S. Hari et al., “Relyzer: Application Resiliency Analyzer for Transient Faults,” *IEEE Micro*, vol. 33, no. 3, May 2013, pp. 58–66.

[31] B. Döbel et al., “Investigating the Limitations of PVF for Realistic Program Vulnerability Assessment,” *Int. Conf. HiPEAC Workshop Des. Rel.*, Berlin, Germany, Jan. 21–23, 2013.

[32] J. Li and Q. Tan, “SmartInjector: Exploiting Intelligent Fault Injection for SDC Rate Analysis,” *IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst.*, New York, USA, Oct. 2–4, 2013, pp. 236–242.

[33] C. Hesscot et al., “A Methodology for Stochastic Fault Simulation in VLSI Processor Architectures,” Semiconductor Research Corporation, June 2010.

[34] C. Artho et al., “Using Checkpointing and Virtualization for Fault Injection,” *Int. Symp. Comput. Netw.*, Shizuoka, Japan, Dec. 10–12, 2015, pp. 347–372.

[35] H. Schirmeier, L. Rademacher, and O. Spinczyk, “Smart-Hopping: Highly Efficient ISA-Level Fault Injection on Real Hardware,” *IEEE Eur. Test Sym.*, Paderborn, Germany, May 26–30, 2014, pp. 1–6.

[36] K. Parasyris et al., “GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates,” *Int. Conf. Dependable Syst. Netw.*, Atlanta, GA, USA, June 23–26, 2014, pp. 622–629.

Jongwhoa Na received a BS in Electronic Engineering from Sogang University, Seoul, Rep. of Korea, an MS in Computer Engineering from Wayne State University, Detroit, MI, USA, and a PhD in Computer Engineering from the University of Arizona, Tucson, USA. He is a Professor of Electronic Engineering and Avionics at Korea Aerospace University, Goyang, Rep. of Seoul. His research interests include dependable embedded systems, safety and security certification, and assistive embedded systems. He is a member of the IEEE.

Dongwoo Lee received a BS in Information and Communication Engineering from Hansei University Gunpo, Rep. of Korea, and an MS and PhD in Avionics Engineering from Korea Aerospace University, Goyang, Rep. of Seoul. He is a researcher at the Avionics Research Institute of Korea Aerospace University. His research interests include dependable embedded system design, computer architecture, and fault injection.