FAULT TOLERANCE CORE:
A FRAMEWORK FOR APPLICATION-AWARE RELIABILITY

BY

VALENTIN SIDEA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisers:

Professor Ravishankar K. Iyer
Research Professor Zbigniew T. Kalbarczyk

Abstract

As processor manufacturers keep pushing the limits of the transistor, the reliability of computer systems has become an increasing concern. Various fault tolerance techniques have been developed in an effort to provide reliable computing in the presence of faults. These approaches suffer from either a high resource cost or high performance overhead. This thesis presents a design for a Fault Tolerance Core (FTC) that uses configurable application-aware hardware modules for improving reliability. Application-aware fault tolerance is achieved by detecting perturbations in application execution through the monitoring of processor pipeline signals. This approach leverages hardware resources more efficiently than replication. The FTC achieves low overhead by placing fault tolerance hardware separately from the processing core, minimizing the processor data collection hardware, and by performing fault detection in the background.

This thesis presents work that has been completed towards the achievement of a FTC. This work includes a hardware assisted incremental checkpoint, an application hang detector and a preliminary FTC framework for integrating these into a Leon3 microprocessor. All modules have been implemented and tested on a Leon3 synthesized atop a Stratix III FPGA running a Linux environment. A hardware fault injector capable of modifying 9 distinct processor pipeline signals has been implemented for performing validation experiments on the modules.

Table of Contents

1 Introduction

As processor manufacturers keep pushing the limits of the transistor, the reliability of computer systems in industry has become an increasing concern. With each new generation of manufacturing technology the susceptibility of processors to hardware faults has increased, as described by [1]-[5]. These faults, referred to as soft faults, are generally caused by particle strikes or excessive heat in the chip, and are manifested by changing the state of transistors unpredictably. In turn these hardware faults can affect the software executing on the system and cause a crash, hang or a silent data corruption. With the number of faults in time (FIT) of processors increasing with each generation, there is a growing necessity to develop fault tolerant techniques to prevent most of these hardware faults from manifesting at the software level.

Many fault tolerance techniques have already been developed to provide reliable computing in the presence of soft faults. These techniques can be divided into different categories based on two criteria. The first criterion refers to the method used for correcting errors. There are methods that perform forward error correction and methods that perform backward error correction. Fault tolerance techniques that fall under the forward error correction category are those that hold off committing a result until error checking has been performed and the result has been validated. Backward error correction techniques perform checks with a certain

delay or only perform checks at certain locations requiring the program to re-execute a section of the code. Re-execution can be performed with the use of a checkpoint process. The second criterion is founded on the resource used for detecting or correcting the error. It classifies fault tolerance techniques into three categories: hardware-based, software-based, and hardware software co-design.

To our knowledge all hardware-based fault tolerance techniques perform forward error recovery. Several of these have been adopted in mission critical applications. Triple modular redundancy [6] computes the same process on three separate computing systems and uses a voting mechanism for ensuring correct computation. This approach has been implemented in the computing system used on seven space vehicles including Skylab and Space Shuttle [7]. IBM has used a hardware duplication technique in the S390 G5 processor [8]. In this architecture the instruction fetch and execution units are duplicated to allow for redundant execution of every instruction. Before committing the instruction there is a comparator to determine whether the instruction has succeeded or not. Upon failure the instruction is re-executed.

Other hardware fault tolerance techniques include parity schemes for protecting register files, caches, and buses. This protection method has become the standard for server processors. Intel Xeon [9] and AMD Opteron [10] processors have multi-bit protection on all of their busses and register files. Parity protection has also been used in verifying arithmetic operations in the execution unit using BHC code [11].

With the exception of the parity schemes, all of the hardware duplication techniques mentioned above suffer from a high hardware overhead. The parity schemes do not perform full duplication and therefore require much less than 100% resource overhead. However, techniques like TMR are very expensive - more than three times the hardware cost of a single machine - because they require a voting mechanism in addition to three separate machines. The hardware duplication adopted by IBM also requires a significant hardware overhead because it essentially duplicates the entire processor pipeline.

With the appearance of multicore processors, another hardware approach has been developed, Simultaneously and Redundantly Threaded (SRT) processors [12]. In this design two threads - one leading, and one trailing - are launched to complete the same task. The trailing thread is used to perform the same computations as the main thread and checks in hardware are performed before each instruction commit to ensure both threads reach the same results. In essence, this technique is similar to the IBM redundant pipeline except that this method is more flexible because it allows the cores to be used either for running redundant threads or for running additional threads for increased performance. The approach described above has a significant hardware cost as well, but it can also be configured to provide higher performance depending on system requirements.

The hardware methods discussed so far will retry any instruction that was not performed correctly. It is important to note that not all hardware faults manifest at the software level. For any clock cycle the CPU does not use all of the transistors on a chip simultaneously. This implies that many hardware faults are inherently

masked by the architecture of the system. Similarly, not all parts of an application are critical for correct application execution. For example, let us assume that certain variables are corrupt but are used only when a program reaches a certain state B. If the program never reaches state B, the corrupt variables will never be used and the application will execute without any errors. This means that many hardware faults that occur at a low level are likely to be masked by the layers above and may never affect the execution of a program. Previous research has shown that 5% of logic faults propagate to application [13], [14]. The fault tolerance methods mentioned thus far would retry faulty instructions that will not manifest at the software level, and perform redundant operations that are not required.

Various software fault tolerance techniques have been implemented which focus more on errors found in the software. These are more accessible because they require no hardware modification. The traditional approach is N-Version programming [15] and was first proposed in 1978. This approach involves executing multiple versions of the same program on separate machines simultaneously or on the same machine at different times and then comparing the results of the two runs. The correct result is selected based on voting. Another version of this approach is process level redundancy (PLR) [16]. In this scenario all redundant threads and checking are performed automatically, transparently to the user and application, allowing easy handling of the redundant threads. This approach falls under the forward error correction category because a result is always selected from the N versions without re-execution. Similar to the ones above, this approach also costs a significant overhead. It either requires N times more

hardware resources or N times more execution time to complete executing a program.

Other techniques that use software and hardware to perform fault tolerance involve perturbation-based detection. An example of this approach is found in [17], which shows how software is used to determine patterns in the execution process. These patterns include variable range values, historical variable values, TLB misses, and other patterns. Hardware is added to the processor pipeline to monitor a select set of application variables. Error detection is performed in the hardware to alert the system of any possible faults. Another example founded on the same method is the Reliability Security Engine (RSE) [18]. The RSE is novel in that it provides a generic framework that performs data collection and can be used with multiple fault detectors. This method is unique as well because it relies on application specific configuration. Each application is manually or automatically analyzed for specific patterns that can be used to provide indications of possible faults. These methods leverage hardware resources better and require much less resource overhead than the previously mentioned techniques. The main downside of these proposals is that they alter the CPU core's longest path, requiring a slower clock cycle. This is a result of tapping pipeline signals and adding area directly into the processor's core. By adding more area inside of the CPU core it complicates routing and causes a longer path. The discussed techniques are both backward error correction mechanisms and require a checkpoint and recovery mechanism to resolve any detected errors. Since faults are not expected to occur often, the overhead incurred from rollback is not a concern.

Much work has been done on recovering either applications or full systems from faults using a checkpoint/recovery process. A few examples of these are BLCR [19], CoCheck [20], MPICH-V [21], TICK [22], Pickpt [23], and [24]. These techniques work on the basis of taking snapshots of the application state repeatedly at a certain time interval, and use various fault alarms to determine when the application requires recovery to a previously stored checkpoint. Fault alarms are usually exceptions detected either by the architecture or the operating system. These alarms could be caused by an application jumping out of scope or an arithmetic exception. Software inconsistencies, such as sanity checks, can also be used to trigger a recovery. The reliability of any computing system with a checkpoint mechanism is highly dependent on the accuracy of its fault detection mechanism.

The biggest challenge in providing a reliable computing system is keeping the overhead costs down. Triple redundancy systems provide very good reliability but come at a very high hardware cost. Software-based fault detection is generally achieved by time redundancy, which significantly degrades the system performance. Perturbation based fault detection appears to be the most efficient in terms of resource overhead at the present time. However, the current methods require modifications to the CPU pipeline, which leads to an increase in the critical path and results in decreased computational performance.

Because of the efficient use of hardware resources, we propose to use a perturbation-based approach for providing fault tolerance. In order to overcome the limitations of the previous techniques, we propose utilizing a separate heterogeneous core named Fault Tolerance Core (FTC) for providing reliability in a

single or multicore computing system. The proposed FTC is composed of a set of hardware modules that provides various fault tolerance services for executing applications. These modules are application aware and can be configured by software.

Data collection hardware is added to each of the standard computational cores to collect runtime information. The hardware is kept to a minimum, and is negligible compared to the existing CPU cores. This collection hardware simply taps some of the pipeline signals from the CPU and forwards them to the dedicated FTC. We believe this new design is able to overcome the limitations of the RSE because the small amount of hardware will have a negligible impact on the critical path.

All collected data from the CPU cores is forwarded to the FTC. The FTC is aware of the application executing on each core and uses the collected data to detect any perturbations in software execution. All of the fault tolerance computation is performed directly in hardware on this FTC. By performing this computation on a separate core we will limit its impact on the functionality of the existing CPU cores. In addition we provide a backward error recovery mechanism in order to allow application execution to progress while fault detection is performed in parallel with execution. These three features eliminate the runtime overhead incurred by fault detection.

Similar to the RSE, the proposed FTC is application aware in order to provide efficient fault detection. Each application benefits from a different set of fault detection techniques. It is important to only enable fault detectors that will benefit the application in order to maximize the reliability while minimizing the

computational overhead. By being application aware the FTC can be programmed to protect only critical applications. It can even protect only critical parts of applications to use the available fault tolerance resources in the most efficient manner. By performing fault detection at the application level, the FTC will not detect and attempt recovery from the large number of benign faults which are only visible at the hardware level [13], [14]. Recovery from these errors will affect performance and is unnecessary since the errors will not have an effect on the execution of the application.

The contribution of this thesis is an application aware framework for providing fault tolerance similar to RSE, but with a smaller performance overhead. This thesis describes a novel FTC design and exhibits the whole body of work that has been done towards achieving this goal. Three hardware modules have been implemented on an FPGA for the Leon3 [25] processor as proof of concept, and are described in the following chapters. Chapter 2 describes the design of the FTC. Chapter 3 presents a hardware aided recovery mechanism to be used with the FTC. Chapter 4 describes a hardware-based application hang detector. Chapter 5 shows a fault injector that was developed for validating the recovery and hang detection mechanisms.

2 Reliability Framework

This thesis provides a novel approach for improving the reliability of multi-core computing systems in the presence of hardware faults. The existence of these faults and their effects on hardware logic have been researched and well documented [1]-[5]. The traditional approach for overcoming these errors has been to replicate various hardware components and perform the computation of these components in parallel. Examples of this approach are [6], [8] and [12]. Such approaches benefit from a low performance overhead and high fault coverage but suffer from high hardware cost. Only replicating the instruction and execution unit of every core will incur 35% overhead [8]. Similarly re-executing an application thread on a separate core essentially incurs 100% overhead since the second core is dedicated to executing the same code.

Due to the fact that only a small number of hardware errors manifest at the application level [13], [14] it is not necessary to protect against all hardware errors. Instead, if all application errors can be detected, it means that the same reliability can be guaranteed. Various reliability methods that focus on application level faults have been investigated, for example [17] and [18]. Both approaches rely on monitoring application behavior by collecting runtime information. Reliability is improved by detecting execution anomalies based on knowledge about the application. The application information is either determined statically or dynamically. In [17] variable values are collected through a software mechanism. In [18] internal CPU pipeline signals are monitored continuously giving the reliability

engine a very detailed view of the execution behavior. The two approaches suffer from a significant performance overhead. In the first case software is used for data collection and analysis requiring significant CPU execution time for reliability. In the second example the reliability engine is tightly coupled with the pipeline. Multiple signals are collected from the pipeline requiring either complex routing from the core or for the RSE to be placed next to the pipeline. Both solutions will have a significant impact on the critical path in the pipeline. In turn this will require a reduction of the clock speed, and degrade computing performance. Moreover the proposed RSE does not scale well in a multi-core environment since a dedicated engine will be instantiated for each core.

This chapter presents a novel solution for improving computing reliability. Similar to RSE, our approach is to detect execution anomalies by dynamically monitoring application behavior using dedicated hardware. The goal is to improve the reliability of computing without impacting execution performance with a minimal hardware cost. For this purpose we propose the use of a dedicated Fault Tolerance Core (FTC). This core should be utilized as a custom hardware block that is fabricated on the same die as a commercial off-the-shelf CPU. This core is responsible for improving the reliability of a collection of applications executing on any of the CPU cores. We believe that decoupling the reliability engine from the core pipeline reliability can be achieved with negligible performance impact at an acceptable hardware cost. The FTC proposed in the present paper should be capable of servicing multiple CPU cores and therefore should scale well with larger computing systems. The remainder of the chapter describes the proposed

conceptual design and a possible implementation along with a summary of the work completed in this direction.

2.1 Fault Tolerance Core Concept Design

For the purpose of improving the reliability of computing systems we propose the use of a newly designed FTC. This component is responsible for providing reliability services to select applications executing on the CPU's cores. The core is composed of a set of custom hardware controllers called service modules (SM) that perform perturbation based application monitoring similar to [17] and application checkpoint/recovery. Examples of such controllers include: hang detection, variable range checking, memory access pattern, etc. FTC also contains data collection hardware that has access to the CPU bus and a few CPU pipeline signals. This collection hardware is shared across all service modules. Collected data is forwarded to the service modules based on configuration parameters. The controllers receive application data and perform the monitoring passively in parallel with the application. This approach should be able to perform checking passively, allowing execution to continue and faults to be detected with a certain delay after they occur. As long as the FTC has enough resources to keep up with the average application execution, the passive monitoring should not delay application execution and therefore should not incur a performance overhead.

Each service module is application aware and can be configured to monitor specific applications. For the design presented here we assume that each service module will only be configured to monitor one application at a time. We believe that each service module can be implemented to support multiple applications by

instantiating multiple sets of its hardware blocks and some extra logic. However, this is outside of the scope of this paper.

In order for the FTC to keep pace with the multi-core processing, only critical applications should be monitored. Additionally, the reliability improvements of each application will vary differently with each service module; therefore applications that do not offer a substantial benefit from service modules should be omitted. Monitoring should be configured for each system and workload to maximize the usage of each service module, but should be limited to keep up with the real-time requirements.

The rest of this chapter is divided into three sections. The first describes how the configuration and software flow should be performed. The second section describes what application data should be collected and how to integrate the FTC into a multi-core CPU. The final part describes the proposed FTC architecture and how it analyses monitored applications.

2.1.1 Target Application Execution Flow

This section describes how monitored applications should be executed in the presence of the FTC. Details about the proposed core's functionality are presented in the following sections. The FTC performs fault detection at the application level; therefore each service module requires information about its target application. Depending on the reliability service provided, either specific variable addresses or function locations are necessary for fault detection. These service modules are configured with application specific parameters such as variable addresses and function locations.

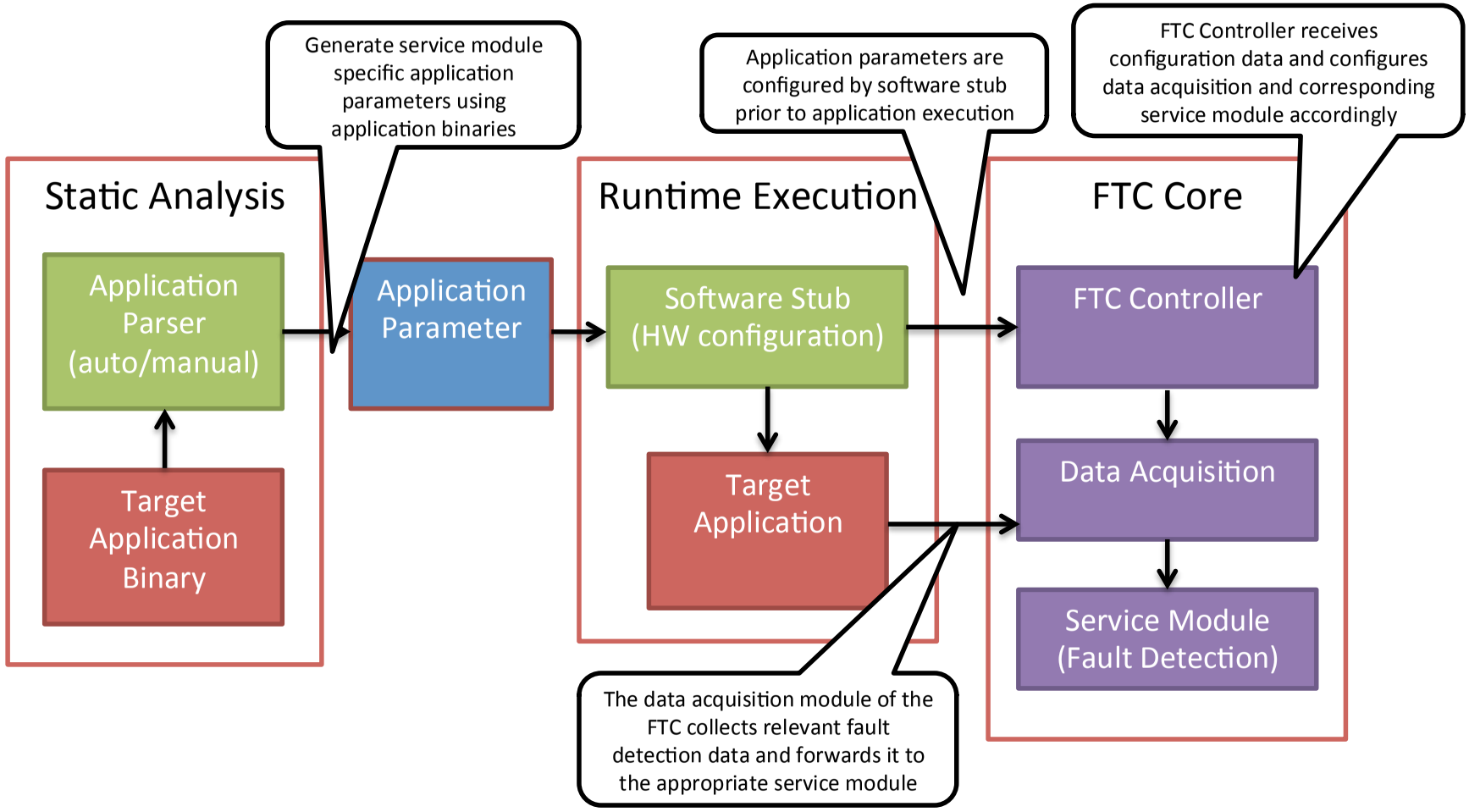

Figure 1: Monitored application profiling and execution

Figure 1 presents an outline for executing an application in the presence of the FTC. The process starts with either automatic or manual profiling of the application to determine parameters for the specific reliability service that will be provided. For example, if function-level hang detection will be performed, the profiling would result in a list of virtual addresses indicating function locations.

On the system with the FTC, a software stub should be employed for launching the target application. This software stub is responsible for reading the application parameters generated in the parsing step and for configuring the hardware for the target application. In addition the target application's PID may also be passed to the hardware if the service module requires it. As described in the following section the FTC controller is configured via the CPU bus. Since user level applications do not have access to peripheral address space, configuration can be performed by a kernel module, which has access to the FTC controller's memory space. Alternatively, the application stub can perform an mmap system call to configure the peripheral address within its address space.

At runtime the FTC collects relevant application information via the data acquisition module. This data gets forwarded to the service module that performs the necessary fault detection and has the ability to raise an interrupt to indicate a fault was detected. This process is described in more detail in the following sections.

2.1.2 Data Collection

The service modules in the FTC require runtime information from the monitored applications executing on the CPU cores. The required data is composed of control flow information, and specific variable values. In this section we give a description of the specific runtime data required and explain how to integrate the FTC core into a multi-core system in order to be able to collect the same data with no effect to the critical path.

The control flow information required for application monitoring consists of the process ID (PID), process state register (PSR), and program counter (PC). The PID is used for determining what application is currently executing on the CPU. The service modules use this information to determine whether or not to collect application data, and to look up configuration information about the application. The PSR register is used to determine when the core is executing in kernel or user space. This register is employed in conjunction with the PC by service modules that track application control flow. All of these registers need to be collected directly from the CPU cores.

Collection of specific application variable values is crucial in performing range checking and other perturbation based analysis. Variables to be collected have to be specified and configured in the FTC before the application is executed. Variables are monitored on the memory bus and therefore are identified based on a physical address. This address can be determined at the OS level through a kernel module. An example of how this works can be found in the checkpoint service chapter.

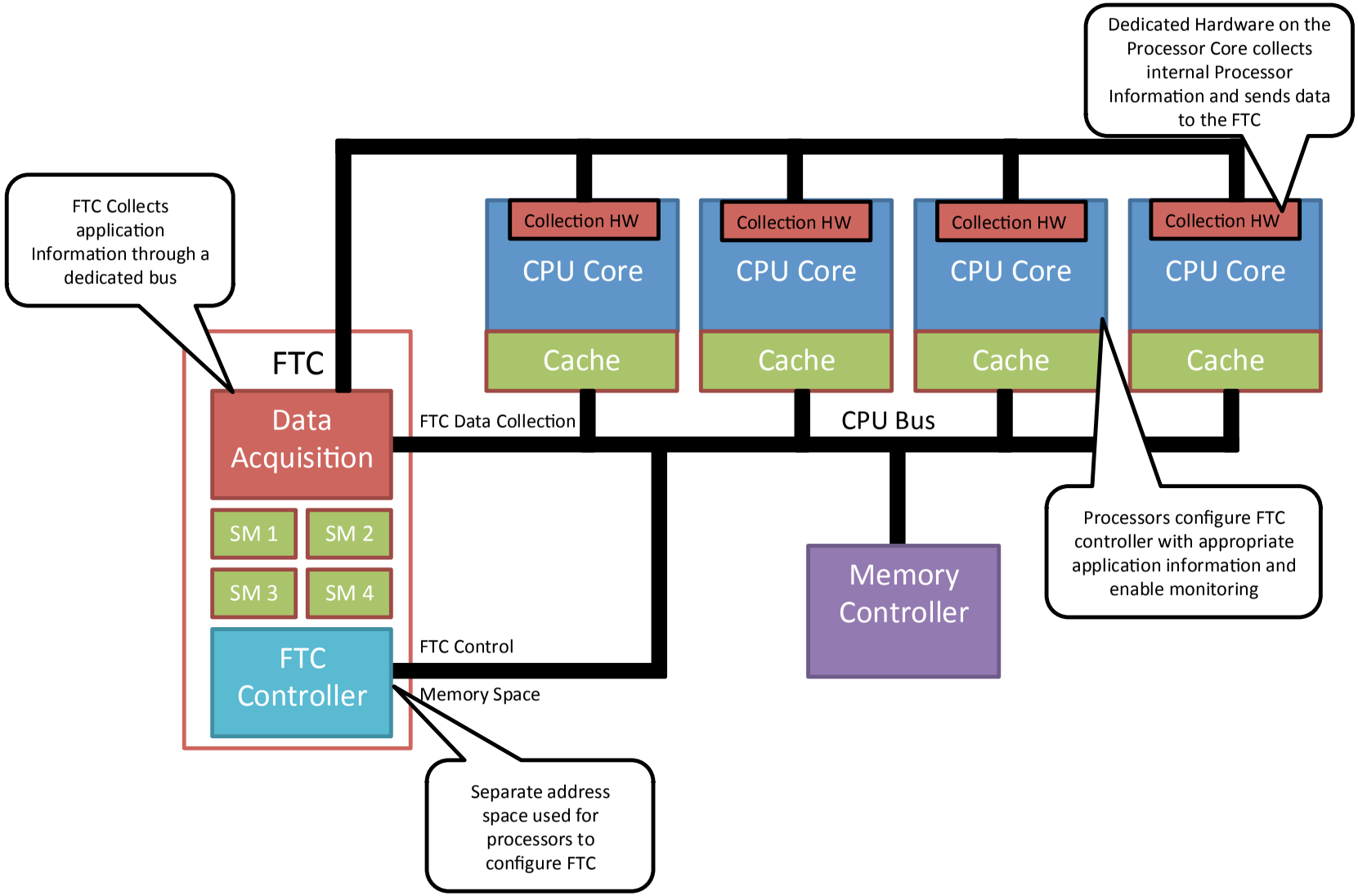

Figure 2: Integration of the fault tolerance core into a multi-core Leon3 CPU

In Figure 2 we depict how the FTC core should be integrated into a multi-core system. The interconnection will vary slightly for integration into other architectures. As displayed in Figure 2, the FTC is composed of various service modules, a data acquisition unit, and a controller. Both the data acquisition and the controller have a connection to the AMBA Bus. The FTC controller has a bus slave interface and is allocated its own memory space. This connection gives the processor visibility to the FTC configuration registers and can be configured like any traditional peripheral component. In addition, the slave interface has an interrupt associated with it. Whenever a fault is detected or the controller requires servicing, it is able to initiate a request to the processor via the interrupt.

The Data Acquisition module collects control flow data through a dedicated interface to each core. Variable values are collected through an AMBA slave bus that performs sniffing. The FTC controller will be configured with the physical addresses for various variables to be monitored. Whenever the data acquisition module detects the monitored function's address on the bus it stores the variable value and forwards it to the appropriate service module.

As displayed in Figure 2, a small piece of hardware is added to each processing core. This hardware module is simply used to pass the PSR, PC, and PID to the data acquisition. The shown hardware consists of three sets of extra registers to pipeline the values coming from the core's registers. This set of pipelined registers is a standard hardware technique for simplifying the routing inside the core and allowing the placement of the extra registers to be far away from the actual core. The method will not affect the critical path of the core since it only adds one

fanout to each of the three monitored registers. The pipelined registers are acceptable in this design because the monitoring is done passively and the data acquisition can receive the data a few cycles late.

Figure 2 shows the connection between the data acquisition and the hardware cores. The FTC receives the collected data from each core simultaneously over one dedicated path for each core. The paths are one-directional and are allowed to have a multi-cycle delay (through pipelining). This implies that they have a very low routing priority and will have a minimal impact on the existing layout of the CPU.

2.1.3 Fault Tolerance Core Architecture

The FTC is a custom hardware optimized to provide various reliability services for applications executing on a multi-core system. Since each application will benefit differently from the various service modules, the amount of processing done by each service module will vary with different workloads. In order to maximize the amount of reliability provided given a certain FTC hardware footprint, we suggest the use of a shared ALU/Local Storage hardware block that can be dynamically reassigned to different service modules.

Figure 3 depicts the FTC being composed of four blocks: a data acquisition unit, a fault tolerance controller, a set of service modules, and a shared hardware block. As described in the previous section, the FTC controller has both a bus master and slave connections. The slave reserves a dedicated address space and is configured directly by the CPU.

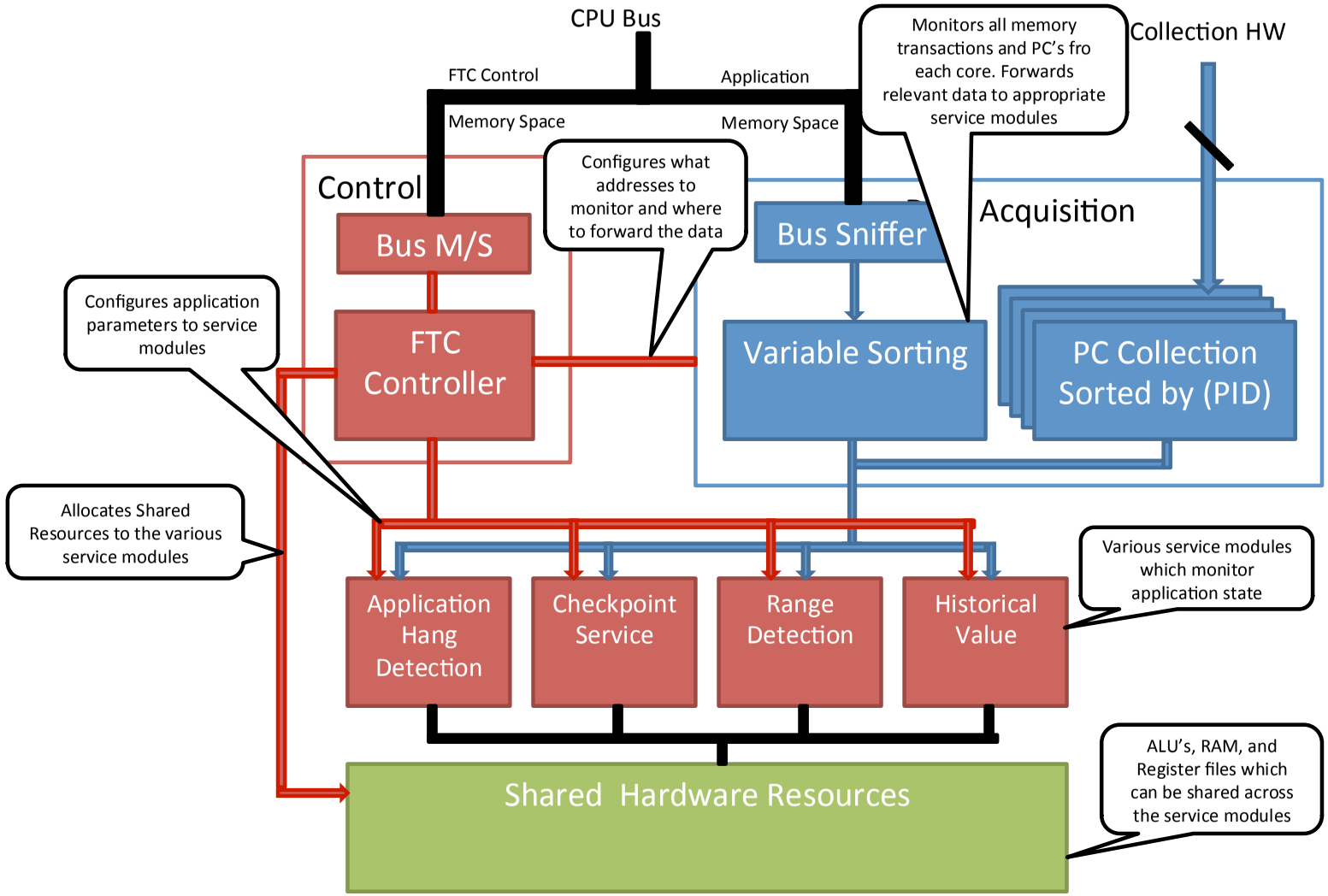

Figure 3: FTC architecture

The configuration indicates what application(s) each service module should monitor and any specific application parameters that each service module requires. If specific variables require monitoring, the controller will program their addresses into the data acquisition module. For control flow, checking the data acquisition will require the PID for these applications to forward only those PC values to the appropriate service module. Moreover the FTC controller is responsible for allocating the shared resources appropriately to each service module. The FTC controller has a bus master for any service module that requires access to memory storage.

The data acquisition module is responsible for sniffing the memory bus for specific physical addresses specified by the controller. For each physical address, the acquisition module keeps track of which service module requires the value of this variable. Whenever one of the configured variables is detected the address and value of the variable are forwarded to the appropriate service module. For control flow monitoring, the data acquisition module is configured with the application PID to be monitored and any PC value received for that specific PID is forwarded to the appropriate service module. Since multiple cores are being monitored simultaneously, the acquisition module has to keep track of each core's PID and forward all of the PC values from each core to the service module. In the Leon3, pipeline stages typically require multiple clock cycles to complete. This implies that PC values are not generated at the clock cycle rate. Through buffering, the service modules should be able to keep pace with the PCs from the multiple cores.

The service modules contain all the hardware resources required to perform the specific reliability service. Later in the thesis, two implemented service modules are described for check pointing and hang detection. The other service module examples are not covered by this thesis.

The shared hardware resources block is comprised of ALU and local storage units that can be used differently by each service module. Depending on the service modules implemented, the ALU units in this block may not require sharing. The sharing technique can be a simple FIFO of requests with a tag for which module made the request. When the computation is complete the tag will indicate which service module requested the operation. The local storage should be implemented as sets of block rams. The controller can configure each block ram to be multiplexed by a specific service module.

2.2 Completed Work

In order to better understand the optimal FTC architecture and the data acquisition requirements, research is first performed on designing and testing custom hardware to provide various reliability services. The service module needs are used to determine the requirements of the FTC. At this stage, every service module should be implemented with its own interconnect to the CPU, to enable it to function correctly. The FTC simply provides a shared interconnect and local storage for all of the service modules to use. Once the service modules are designed and evaluated, the FTC constraints will be apparent and the optimal design can be implemented.

For the purpose of implementing and testing a system with the proposed FTC core, we used the Leon3, an open source SparcV8 CPU. The processor has been selected because it is described in VHDL and has a well-established community with good technical support. The Leon3 processor also has a number of supported Linux distributions available. The CPU is synthesized and executed on an Altera Stratix III FPGA. All experiments have been executed in a Linux environment executing atop a single-core Leon3 processor.

The Linux distribution used was Sparc Linux 3.4.4. This version is built around the 2.6.36 Linux Kernel. The distribution comes with Busy Box and cross compilers for building other applications and libraries. The said distribution also includes an SSH and SCP which was used to transfer files between the Leon3 and x86 host machine. All test applications are cross-compiled on an x86 machine with the cross compilers provided with the Linux distribution. The binaries and other input files were all copied over to the Leon3 environment using SCP.

Figure 4 depicts the completed work and the way it was integrated into the Leon3 system. The completed service modules are presented in the remainder of this thesis; they are a checkpoint/recovery service, and a low granularity hang detector. The above components have been integrated to work on the same system. The hardware setup displayed in Figure 4 is the same for all experiments referenced in this thesis.

A simplified FTC controller has been implemented to allow the processor to access both of the service module's control registers over the same AMBA Bus. The Leon3 core is modified with a small hardware collection block. The block is used to

collect PC and PID information from the CPU and is passed through a dedicated lane to the FTC. The hang detector requires this data to monitor control flow. In addition to these service modules, a hardware fault injector has been developed for evaluating the functionality of our modules. The fault injector is also displayed in Figure 4. This injector is contained fully within the Leon3 pipeline and is described in Chapter 4.

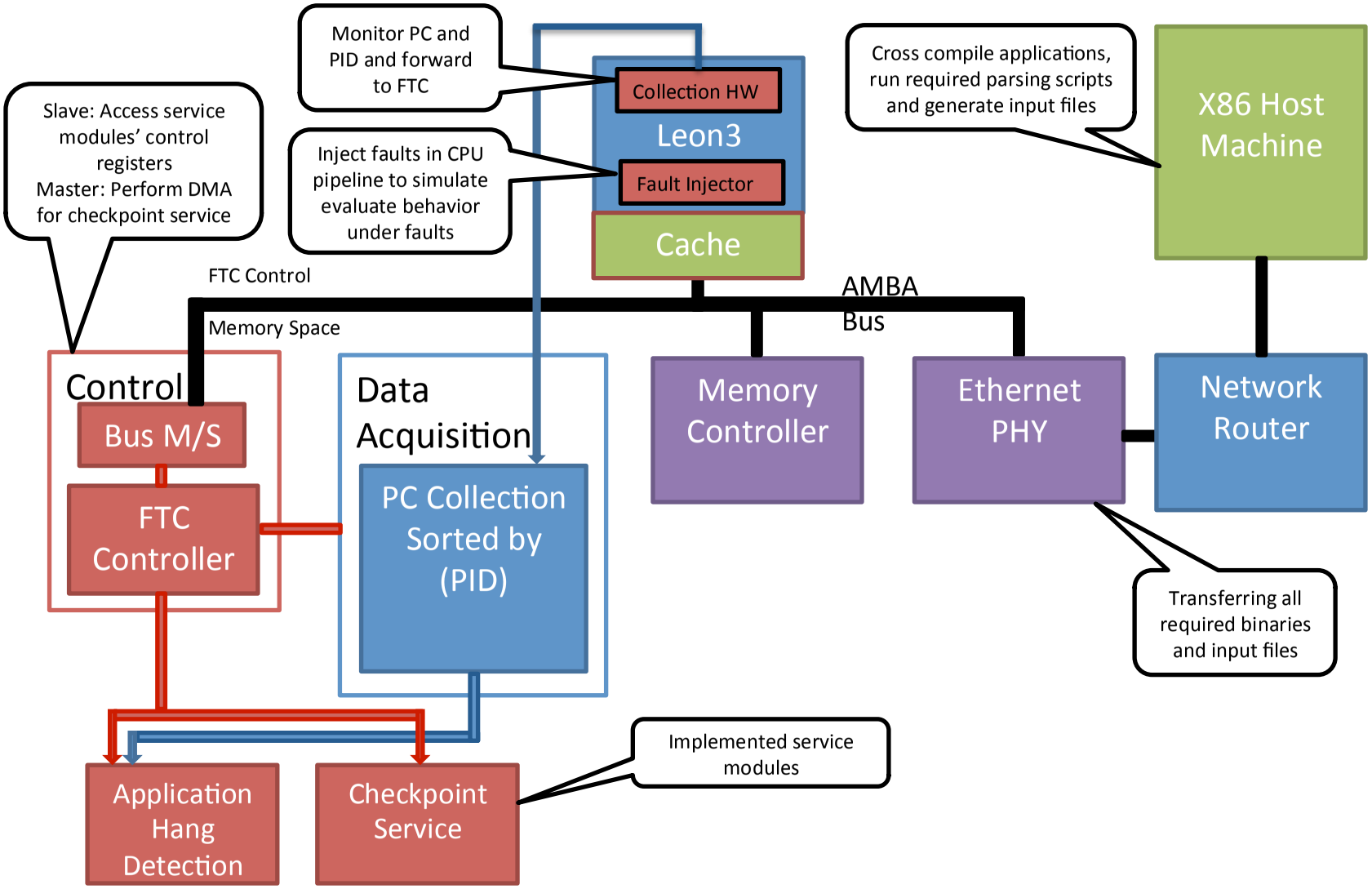

Figure 4: Completed work and experimental setup

3 Checkpoint

In computing systems checkpoint is an essential mechanism for ensuring system availability. Checkpoint enables the system to continuously take snapshots of running applications; in the presence of a fault, the application can be rolled back to the most recent snapshot and continue execution with minimal downtime. Under error free execution, checkpoint incurs performance overhead. To make checkpointing attractive, the performance overhead must be minimized.

All checkpoint mechanisms work on the basis of taking a snapshot of the running application. The snapshot required to recover an application consists of all application memory, opened files, sockets and IO devices. This chapter focuses on checkpointing applications that only require memory state recovery. The application memory state has the largest footprint and therefore will be the most time-consuming. Saving sockets and IO device states are outside the scope of this thesis and have been addressed in ReViveI/O [26]. In this work the I/O traffic is buffered during each checkpoint and if a recovery is required the I/O is played back so that the application receives the same input during the second execution.

The biggest factor to affect the performance overhead is the method adopted for the memory duplication process. The most rudimentary approach is to perform a full memory copy of the application at each checkpoint interval. This strategy, however, causes a great performance overhead since it requires a large amount of memory bandwidth. For this reason, the most widely accepted approach is to duplicate only a select region of application memory during each checkpoint. This

approach is called incremental checkpoint [19]-[23]. The method of selecting which data to save and at what time interval varies between the different checkpoint types.

Application checkpoint can be performed either fully within the application, semi-transparently by the application with the help of the operating system, or fully within the operating system transparent to the application. Checkpoints performed at the application level can take advantage of the execution process and the known critical data to take snapshots when the least amount of data is required. However, the transparent checkpoint is the most desired method since it requires no modifications or knowledge of the application.

The most efficient transparent checkpoint approach is to save state incrementally. It is accomplished by duplicating only data that has changed from the previous checkpoint. The operating system keeps a secondary copy of the complete application state, which is updated at the rate of the desired checkpoint interval. The efficiency of this approach comes from its small checkpoint footprint, especially for applications with temporal memory locality. In checkpoint mechanisms the amount of data transfer per checkpoint interval is directly related to the performance overhead.

This thesis advances on the incremental checkpoint mechanism by using hardware DMA (direct memory access) to reduce the performance overhead. This new approach allows the CPU to be used for normal execution while the memory copy is performed. Furthermore, the backup copy is stored in local memory, allowing for minimal checkpoint overhead. The rest of this chapter describes the design and implementation of our checkpoint mechanism.

3.1 Design Choices

In order for the proposed checkpoint mechanism to be easily incorporated into existing systems, it must work with unmodified binaries. The technique advanced here provides application recovery transparent to the application. This has been achieved by implementing the newly created design into a kernel module, which has complete access to the application's execution state and memory space. Whenever the kernel module saves or recovers the state of the target application, it puts the application in the frozen state (removed from the schedule list). The state of the frozen application can be reverted to a previous state without corrupting the application's execution.

To reduce performance overhead incurred by checkpointing, we minimize the amount of data to checkpoint. This is achieved by implementing an incremental checkpoint mechanism where only the dirty pages within the checkpoint interval are saved.

The dirty bits in the page table entries are used to determine which pages have been modified since the last interval. These bits are automatically set by the MMU hardware whenever a write is performed to the corresponding virtual memory. The operating system has the option to use these bits. Typically, operating systems employ these bits for performing memory swapping to the hard drive. In our Linux environment, our memory swapping is disabled and therefore modifying these dirty bits has no consequence. In order for this checkpoint mechanism to be incorporated into a computing system with memory swapping enabled, one needs to add another set of identical dirty bits that get set by the MMU in the same fashion.

The operating system will be free to manipulate one set for swapping and one set for checkpointing.

Most of the checkpoint process is performed in software. We developed a kernel module to determine memory pages that requires duplication. A hardware module was developed for performing the grunt work of copying the application memory to a back up location in local memory. We believe local memory is the best place to backup application memory because it is fast, and with Chip Kill, and various ECC [27], [28] techniques, memory is a reliable storage location. In addition RAM is relatively inexpensive and can be added to computing systems to accommodate for the larger memory footprint. The hardware DMA module implemented for this checkpoint mechanism copies memory at the page size granularity of 4KB. Hardware configuration is accomplished by specifying arrays of source and destination memory addresses. By using this DMA module, the performance impact of the checkpoint is reduced by freeing the processor to perform other tasks.

Performing an incremental checkpoint requires the bookkeeping of all application memory pages and their backup location. Tracking is performed with a linked list where each node holds the source and destination addresses for one memory page. The nodes are kept in an ascending address value similar to the application's virtual memory map; this allows easy traversal and insertion of new memory pages.

This checkpoint technique allows very short checkpoint intervals of 1 second. Incrementally backing up memory results in a proportional relationship

between checkpoint size and interval, thereby reducing checkpoint interval results in a quicker backup of data. In addition, saving the backup copy to memory is fast, resulting in minimal application downtime.

3.2 Implementation

This section describes the implementation of our checkpoint mechanism. Our design consists of two software components: a hardware module, and a few kernel modifications. A user level application stub is used for starting the target program and invoking the checkpoint at the desired interval. A kernel module is utilized for keeping track of the application memory pages, and for programming the DMA hardware. Finally, a hardware module performs the memory copy.

A total of four kernel modifications were made to enable our checkpoint and recovery mechanism. Two flags were added to the task_struct: one to mark the process as being checkpointed (PF_CHECKPT) and a second to mark the process as being in recovery. Another flag was added to the vm_area_struct to prevent the kernel from overwriting the dirty bits of the pages we checkpoint. Finally, we modified the fault handler to invoke the rollback mechanism upon a failure.

3.2.1 Checkpoint

The checkpoint process is displayed in Figure 5. The steps described in the text refer to the steps in the figure. In step (1) the application stub performs a fork to execute two separate processes. The child process first writes its PID to a pipe and then executes the target application.

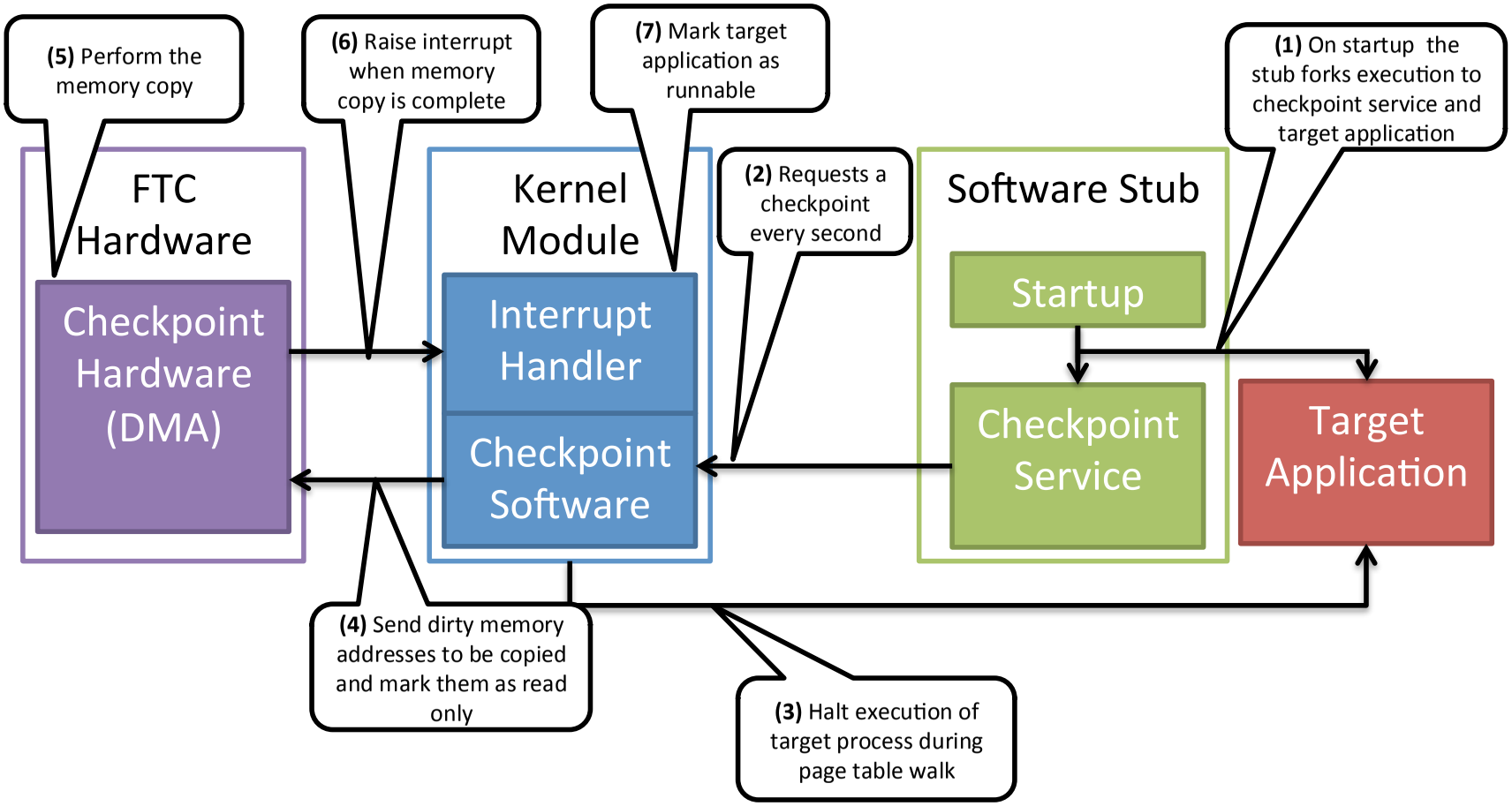

Figure 5: Checkpoint process overview

The parent process reads the child PID from the pipe and in step (2) invokes the checkpoint function of the kernel module every second until the child process terminates.

The following steps repeat every time a checkpoint is invoked. In step (3) the kernel module first checks if the target process exists and is not halted. Afterwards, the task_struct flag PF_CHECKPT is set. Our modified kernel prevents this task from being scheduled when the flag is set. Since we are executing in a single core environment, we can be certain that the target process is not currently executing. If we were executing in a multi-core environment the kernel would require further modifications to first mark the process as checkpoint pending and only enable the PF_CHECKPT flag once the task is no longer executing.

In step (4) the kernel module manually copies the target application's task_struct to a backup location allocated by the kernel module. The task structure backup is required because it contains the application state at the time it was removed from execution by the scheduler. Afterwards the kernel module walks the application's page table, creates a temporary list of all dirty pages, and clears their dirty flag. Furthermore these pages are also added to a permanent linked list that keeps track of source and destination addresses to ensure every application page has a unique backup page. After the temporary list is complete it configures the DMA hardware with the source and destination addresses. The hardware module has a block ram for holding these addresses. Once the block ram is populated, the software writes a register to specify the number of entries in the block ram and another register to invoke the memory copy.

In step (5) the hardware DMA performs the memory copy. A state machine is used to look through the specified number of entries in the block ram and performs a copy from the source to the destination memory pages. Upon memory copy completion in step (6), the state machine raises an interrupt and the CPU returns execution to the kernel module. In the final step (7), the kernel clears the PF_CHECKPT flag to allow for the process to be scheduled.

3.2.2 Recovery

In order to validate our checkpoint mechanism we developed a recovery mechanism that is triggered by the operating system's fault handler. In the following section we describe the validation experiments performed with a hardware fault injector (described in chapter 4 of this thesis). The recovery process is depicted in figure 6. When the application crashes our recovery process starts with the invocation of the modified fault handler in step (1). If the fault handler determines the crashed application is being checkpointed, the process continues with step (2). In this step, the fault handler sends a rollback signal to the checkpoint service and runs schedule() to halt the execution of the target application process. In step (3), when the checkpoint service process becomes active, the rollback signal is received. This signal will cause the checkpoint service to invoke the kernel module's rollback() function which initiates the memory rollback process.

In step (4), the kernel module sets the PF_CHECKPT flag to prevent the scheduling of the target process. The permanent linked list of page addresses

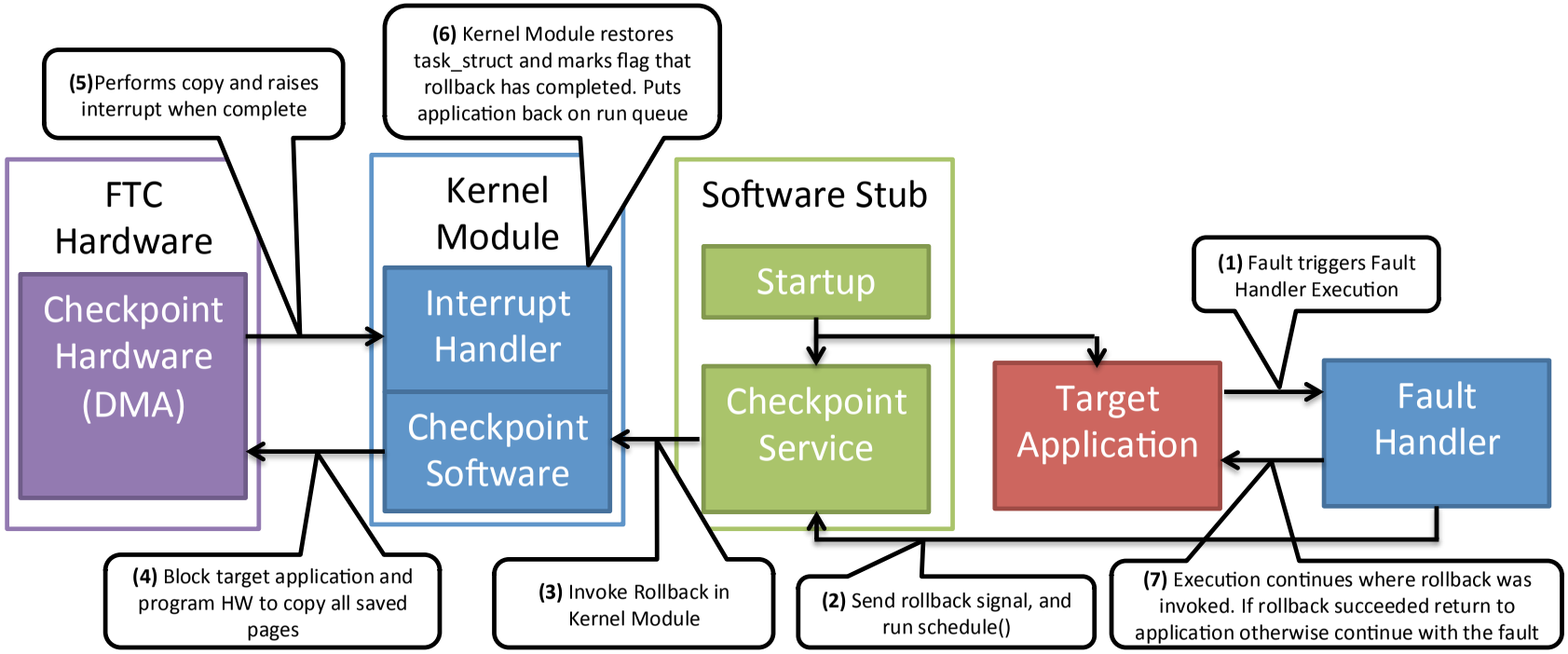

Figure 6: Recovery process overview

referred to in step (4) of the checkpoint process includes all of the application's memory pages and their backup location. This list contains all of the necessary information for restoring all of the application's active memory space. The addresses from this linked list are configured into the hardware DMA module; for each memory page the backup address is used as the source and the original address is used as the destination. In step (5) of the recovery process, the memory copy is performed by the DMA hardware. Upon completion, an interrupt is raised to return execution to the kernel module. At this point the kernel module completes the rollback by manually restoring the application's task_struct from the backup location used by the checkpoint process. In step (6), the kernel module clears the PF_CHECKPT flag to allow the target application to continue execution.

If the rollback was successful, when the target application is scheduled again, it will start executing from the state that it was last checkpointed in. If, however, rollback was not successful, the application will return to the fault handler step (7), which was called when the application first crashed. In this case we allow the fault handler to continue with killing the target application.

3.3 Validation Experiments

In order to test the functionality of our checkpoint mechanism, experiments were performed for validating the hardware DMA module and the successful application recovery. All experiments from this and the following section are performed on the Leon3 system displayed in Figure 4 from section 2.2. The experiments are executed in a Linux environment as described in section 2.2. All

applications are cross-compiled on an X86 machine and transferred to the Leon3 system via SCP.

All experiments are performed with a checkpoint interval of 1 second on three different applications: matrix multiplication, GCC compiler, and BZ compression. All of these applications write their outputs to a file. These output files are compared against a golden run to validate the correct output of the program. Matrix multiplication was always executed on an input of two 400x400 matrices. The application was chosen because it has a simple control flow that is resilient to faults. In the development process this application was useful because even after memory corruption due to improper rollback, we were able to pinpoint errors by performing a diff of the incorrect output to the golden output.

The GCC compiler was selected because it has a very complex control flow and has a very large memory footprint. This application is very susceptible to memory corruption faults due to incorrect checkpoint/rollback. Due to the complexity of the application, we believe this to be a wholesome test for validating rollback functionality. The input chosen for the compiler is an OGG encoder. This application has one of the largest single file source codes with oggenc.c being 1.7 MB. The full compilation of this application requires about 4 minutes on the Leon3 and allows us to take a significant amount of checkpoints at 1 second intervals throughout the execution of the program.

The BZ compression was selected because it has a large memory footprint and it is memory-intensive. The BZ compression was executed on a 119 MB tar file containing all of the GCC binaries and libraries. This application generated an

average of 50 dirty 4KB memory pages per second requiring a large number of memory transfers during every checkpoint. Since the checkpoint overhead scales with the amount of dirty pages requiring backup, this memory-intensive application is a good test for evaluating the performance of our checkpoint.

The hardware DMA needs to correctly copy every memory byte on top of regular CPU bus traffic. This process was verified by adding a software checker function to the kernel module. This function compares every byte of memory that the DMA module copied for each checkpoint. When enabled, this software routine is called immediately after the hardware interrupt that indicates all memory copy has been performed. By executing this software function after every checkpoint memory transfer, we can ensure the validity of our hardware module. This software validation experiment was performed with all three test applications. Each application was executed ten times. In all test cases the software validation test reports all checkpointed memory is copied successfully.

To validate the functionality of our entire design, experiments of recovery under faults are also performed. Specific application errors are injected into checkpointed programs using the Hardware Fault Injector. The fault type selected for this experiment is a PC bit flip for the bit in position 28. The injection is performed on a random location in the code. This injection type is selected because it ensures the application PC will jump out of scope causing an immediate application to crash. As a result the OS will invoke the Linux fault handler, which is modified to invoke the application rollback to the last checkpoint performed as described in the previous section. The current design only supports rolling back to

the last checkpoint; therefore low latency fault detection is required. The recovery experiment is performed with the three applications mentioned above.

The rollback tests were executed ten times with each application. The injection time during each run was different to ensure successful recovery from various states. During all 30 test runs the application output matched the golden run. Under the aforementioned PC fault, the only way for the application to complete execution correctly is by restoring the full application state to the most recent checkpoint and re-executing that section of the application. The successful application completion after the injected fault indicates that our checkpoint and rollback mechanisms function properly.

3.4 Performance Experiments and Analysis

The overhead of our checkpoint mechanism is evaluated by measuring execution time of the three test applications with and without the checkpoint. The programs used for performance experiments were the same as the ones listed in the previous section: matrix multiplication, GCC compiler, and BZ compression. All applications were checkpointed at intervals of one second, and run ten times. The inputs for these applications are the same as the ones used in the previous section.

The variation between all ten runs of the same test never exceeded one second. Table 1 shows the average runtime with and without checkpoint for every application. The matrix multiplication, GCC, and BZ compression, had 15, 20, 50 average number of dirty pages per checkpoint. The performance overhead is related to the average number of dirty pages per checkpoint. More dirty pages per

checkpoint will require the hardware DMA to copy more pages bottlenecking the memory bandwidth for a longer period of time during each checkpoint.

Table 1 Average Checkpoint Overhead

	Average Normal Execution (s)	Average Checkpoint Execution (s)	Overhead (%)
Matrix Multiplication	88.6	91.6	3.4%
GCC Compiler	236.4	243.8	3.1%
Bzip Compression	1514.5	1582	4.5%

The overhead of our checkpoint mechanism is no larger than 4.5% even for memory intensive tasks. This result is very difficult to compare to existing checkpoint mechanisms since none of these have been implemented on the Leon3 processor. The architectural and memory bandwidth difference make it impossible to determine which method is best. In high performance machines the bandwidth is extremely large and the time required to copy all of the extra data is much smaller. The maximum theoretical memory bandwidth of the Leon3 processor used in our experiment is 2GB/s compared to an Intel Core i7 that is 21GB/s. This difference in memory bandwidth will have an effect on the performance of our technique if it is implemented on an Intel machine.

This mechanism stores checkpoints in RAM to reduce overhead. The fact allows for very small checkpoint intervals of one second. In addition, the hardware DMA module is optimized to use the full available bus bandwidth for transferring

checkpoint data. The Leon3 CPU cannot attain the same performance since it requires extra memory operations for maintaining variables and fetching instructions.

We believe the new checkpoint mechanism can perform very well in a multitasking environment on a commercial CPU where the memory bandwidth is much larger than the application requirements. The DMA hardware uses lost cycles in the memory bus, and checkpointing can be performed in parallel with other applications executing on the CPU. We believe that in such an environment the performance impact on the overall system is much less than 4.5%.

4 Application Hang Detection

The most challenging task in providing fault tolerant computing is detecting errors when they occur. The only faults that require detection are those that manifest themselves at the software level. Hardware faults can lead to one of three errors at the application level: crash, hang, or silent data corruption. Crashes are often caused by the application jumping out of scope and are detected by the operating system. Silent data corruption errors are considered the most challenging to detect and the most crucial for the correction functioning of the application. In this particular case, an application may complete with no indication of anything malfunctioning and still the produced result will be incorrect. Application hangs can be difficult to detect since the application continues to execute instructions. However, the application is stuck and keeps executing the same piece of code forever.

For the purpose of accurately detecting hangs, we implemented an application hang detector in hardware that monitors the number of instructions executed by specific functions. The number of instructions executed within a code block can be statistically bounded [29]. The hang detector relies on this principle by comparing the instruction count of the currently executing function against historical values that are maintained for each monitored function. Historical values are readjusted based on the instruction count at the exit of the specific function. The instruction count is performed directly in hardware and does not affect the execution of the target application. The new hardware provides detection at no

performance impact. We believe that monitoring the application at this level of granularity can achieve a very high detection accuracy.

4.1 Design Choices

Application hangs can occur in various ways. The three most common are: dead locks, infinite loops, and incorrect function parameters. In all of these cases the application becomes stuck executing the same code section forever. In order to detect these cases the hang detector was designed to monitor how long the application spends in either all or a set of user-defined functions. Our method of detection guarantees that any hang occurring in the monitored functions will eventually be detected. Depending on the regularity of the application, this type of function level monitoring can achieve low latency hang detection.

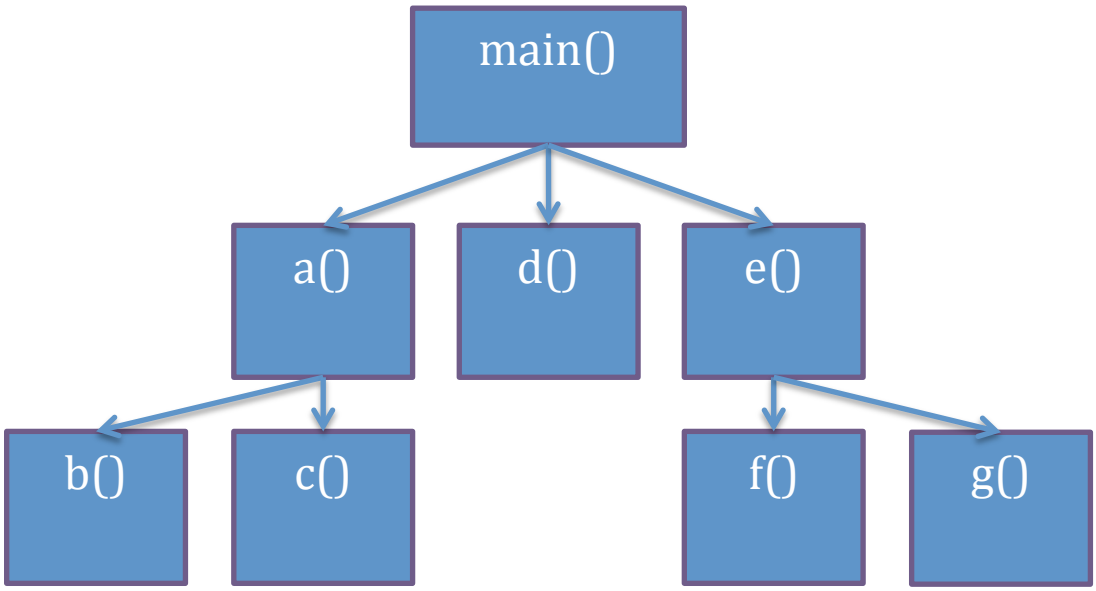

Figure 7: Simple function call graph example

Our monitoring hardware requires storage for each tracked function. To reduce area overhead, monitored functions can be grouped together. To illustrate this concept, we refer to the simple function call graph, displayed in Figure 7. Here all arrows

represent a function call. For example, since function a() points to function b() it means that within function a() function b() is called. Functions in a given application such as function a(), b(), and c() in the graph can be grouped together by only monitoring their parent, function a(). This grouping produces the best results when the sum of the instruction counts across the grouped functions does not vary significantly across various executions. For high-varying functions, monitoring should be performed for individual functions to increase detection coverage and decrease detection latency.

The hardware module that performs the hang detection requires the access to the processor's PC, privilege status register (PSR), and process ID (PID). The PC is required to determine function entry/exit points, and to count the number of instructions executed. Monitoring of the PSR is performed to ensure that only instructions executed in the application space are counted. The PID register is required to ensure that only instructions executed by the monitored application are tracked. Our dedicated hardware uses a set of counters to maintain instruction counts across function calls. The hardware is configured with all of the monitored function entry and exit points before the application is executed. The points are used to determine which function is currently active and which counters are to be enabled. The following section describes the hang detector hardware in more detail.

A hang is declared whenever the instruction counter goes beyond a historically determined threshold multiplied by a configurable coefficient. When the application is launched the threshold for each monitored function should be a very large user-defined value, which is adjusted for every function whenever the function

completes. The equation used for evaluating the new threshold value is a simple weighted average function with an alpha value:

(old threshold) * α + (new threshold) * (1-α).

The alpha can be selected as 0.5 or 0.75 to use shifters instead of multiplier and save hardware resources. Further discussion of this variable is provided in section 4.4.

4.2 Implementation

The hardware setup used for implementing the hang detector is displayed in Figure 4 from section 2.2. The hang detection process is outlined in Figure 8. The hang detector is comprised of three software components and a hardware module. The first software component is a Python parsing script that reads the target application's assembly file and prints all the functions with their entry and exit locations. The second and third software components are displayed in Figure 8 as one block. A software stub is used to perform a fork in which the parent reads the text file generated by the script and invokes the kernel module, which configures the hardware control registers. The child of the software stub invokes the target application. The Hang Detector HW from Figure 8 is shown in further detail in Figure 9. This component monitors the execution of the target process and raises an interrupt upon hang detection.

A Python script was written to parse the assembly file on a host x86 machine. The assembly file contains the virtual address for every instruction in the program. Virtual addresses are used since those correspond to the PC value during execution, therefore the PC is the register the hang detector monitors.

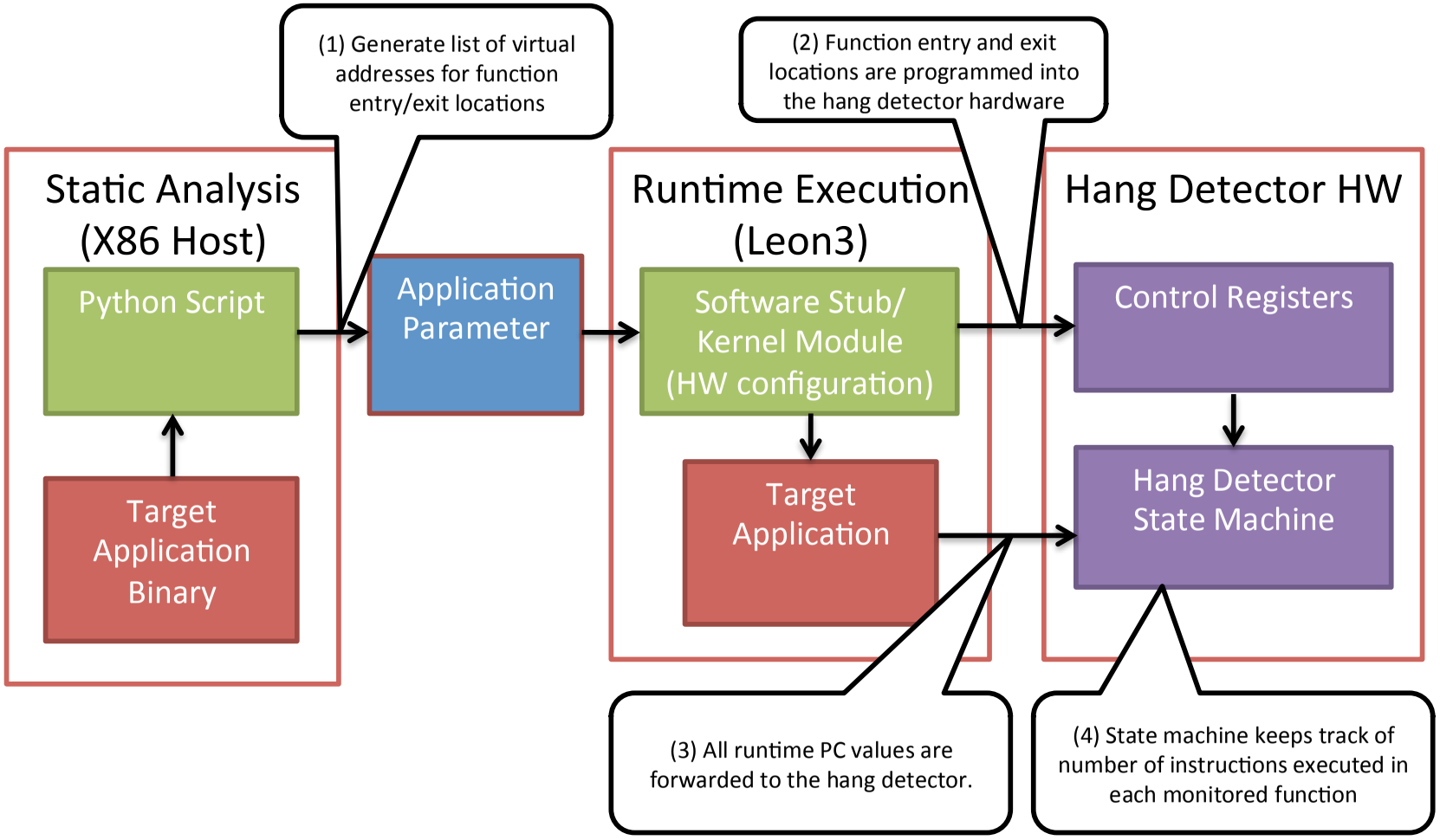

Figure 8: Application hang detection process outline

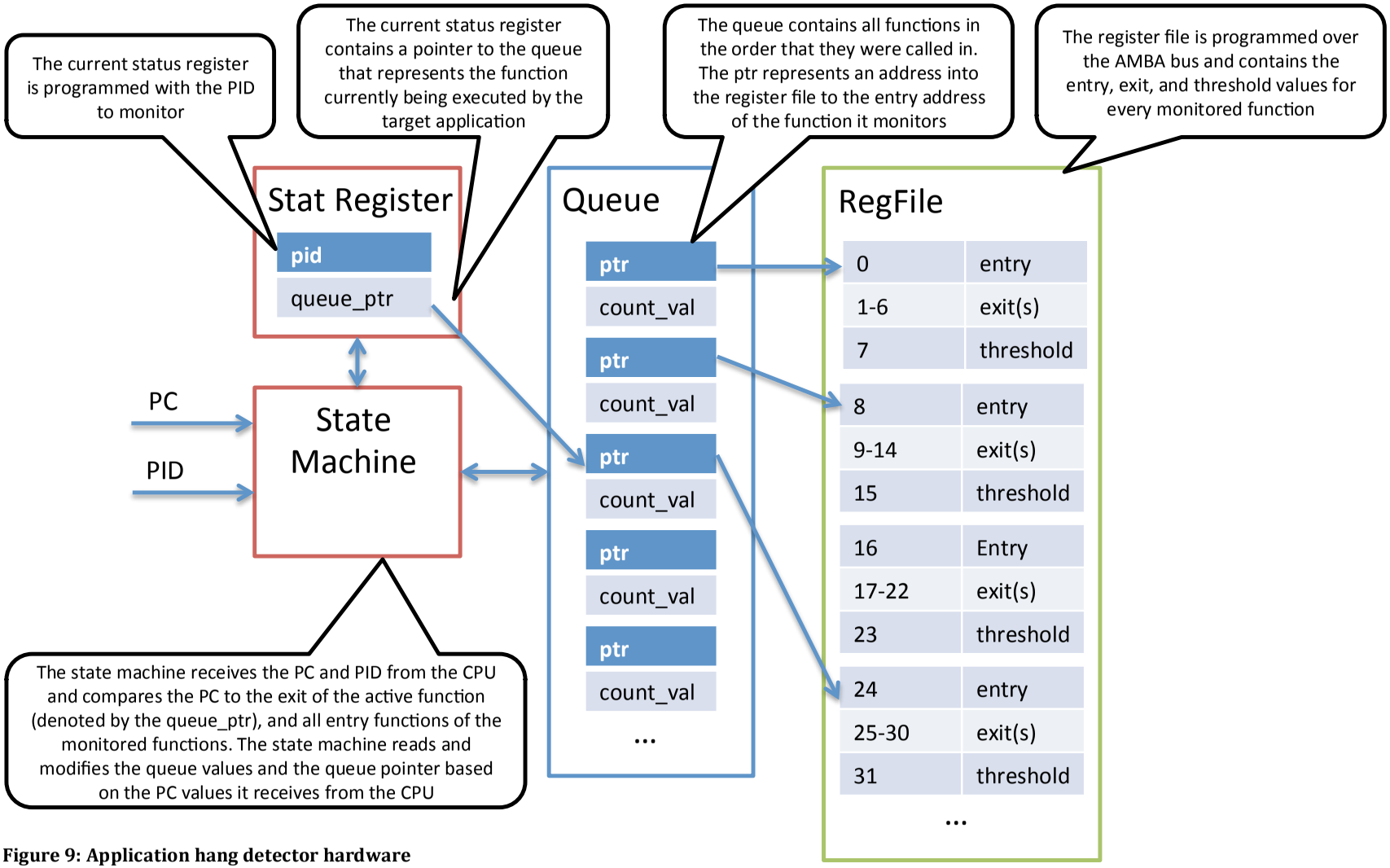

Figure 9: Application hang detector hardware

The script walks through every instruction and searches all function entries. For each function found it stores the address of the first instruction as the entry point. Every RET instruction that is encountered is treaded as an exit location for the corresponding function.

In step (1) of Figure 8, the Python script writes the results to a file that can be read by the application stub. This file contains as many entries as there are functions found. For each function entry there is one address for the function start, 6 addresses for function exits, and 1 value for the threshold value. The threshold value is a configurable constant value that is used across all functions. This value should start with a very high value. Through multiple runs this variable will adjust itself based on the method described in the previous section.

The entries generated by the script for each function will be programmed directly to the hardware control registers. Therefore, the number of addresses for exit locations needs to match the hardware. The value of 6 exit addresses was chosen to contain each function entry within a group of 8 values. Each of these values corresponds to a hardware configuration register. Having this group size will ensure that every function entry will have its address aligned to 8 locations, and therefore simplify the configuration register lookup. None of the applications examined in this paper contained more than 4 exit locations; this is why we believe 6 exit locations to be sufficient. If needed, this value can be expanded to 14 to keep the entries in groups of 16.

In step (2) of Figure 8, the application stub performs a fork in which the child process writes its PID to a pipe, sleeps for 2 seconds, and executes the target

application. The parent reads the child's PID and the script with entry/exit locations to determine the correct configuration for the hardware. Since the CPU interfaces with the hang detector over the standard AMBA bus as displayed in Figure 4, a kernel module was implemented for accessing the hardware configuration registers. The software stub invokes the kernel module with the correct configuration parameters. Once configured, the hardware performs all necessary monitoring. In this step the child process is required to sleep for 2 seconds to ensure that the parent has finished programming the hardware before the target application is executed.

As depicted in Figure 9, the hardware contains a register file of multiple sets of 8 32-bit entries. Each set contains a function entry point address, 6 exit point addresses, and a threshold value (duration). The hardware can be instantiated with a variable number of register sets at the expense of hardware resources. In addition to the storage overhead, there is an extra comparator instantiated for each register set to allow the monitoring of all entry functions simultaneously. The number of register sets instantiated in the register file will determine the amount of functions that can be monitored by the hardware.

The hang detector also uses another storage structure of multiple sets of 2 32-bit entries called the queue. Each set is comprised of a pointer that is the index into the register file for the monitored function and a counter for the number of instructions executed by that function. The hardware has a register that keeps track of the active queue index called queue_ptr. Every time a function entry is found, the queue index (queue_ptr) is incremented and the index of the new function is

entered in the queue at the new position (ptr). The counting is performed until either a new function entry or the current function's exit is detected.

For every function exit, the instruction counter (count_val) in the active position along with the corresponding function's threshold value is used to determine the new threshold value as described in the previous section. The result overwrites the threshold value in the register file. When the function exits the queue, the index also gets decremented. If the active counter (count_val) exceeds twice the threshold value, the hardware raises an interrupt to inform the operating system that a hang has been detected.

The size of the queue is configurable and the only overhead is the number of registers instantiated for the structure. Consequently, a large queue can be instantiated without running out of resources or overcomplicating the hardware. If the function call depth is larger than the queue, the detector will loose track of the function calls. This will cause the hang detector to enter an unrecoverable state where it may miss detection or cause false alarms. In this case, the hang detector will require a reset, which is performed automatically when the configuration process is executed.

The hang detector process has also been implemented to allow the execution of a training phase where the hang detection is disabled and the hardware simply updates the threshold value during runtime. When the application exits, the threshold values are used to overwrite the values in the application parameter list generated by the Python script. This allows the target application to be tested with historical values from previous runs instead of random large numbers.

In order for the hang detector to keep track of the current PID, we modified the scheduler code. Whenever there is a context switch to a new process, the modified code configures our hardware with the active PID. This method simplifies the interconnect between CPU and hang detector as compared to the fault injector described in the following chapter. This method uses the CPU bus to configure the hardware and since we only add one bus write for every context switch, the overhead is insignificant.

4.3 Validation Experiments

Experiments are performed in order to test the correct functionality of our hang detector. All tests from this section are performed on the Leon3 system displayed in Figure 4 from section 2.2. The experiments are executed in a Linux environment as described in section 2.2. All applications are cross-compiled on an X86 machine and transferred to the Leon3 system via SCP.

The functionality is evaluated with three simple applications: matrix multiplication, quick sort, and LU reduction. The applications are kept small to ensure all of the functions fit within the hardware register set. Small applications are also employed because it is easier to understand of the applications' control flow to perform fault injections.

Hangs are invoked with the use of the hardware fault injector described in the next chapter. In every application the faults injected are ALU bit flips. The faults are injected at the first increment of certain loop counters. The 31st bit is always flipped to cause the counter to become an extremely large negative number. This

causes the injected function to execute for a much longer period of time, thus simulating a hang.

For the experiments in this section the initial threshold value for each application is determined by first executing each application in a training phase as described in the previous section. Experiments for each application are performed with the same input as the one used in the training phase. By using the same inputs we believe the threshold instruction counts to be very accurate to the instruction counts during a normal execution of these tests.

For each application three different loops from different functions are targeted for injection. Each injection simulates an infinite loop hang. The three applications are each executed 15 times with 5 runs for each injection location. For all injections the hang detector correctly raises the hang interrupt. The same applications are also executed with the hang detector 5 times each without any fault injection. In this scenario the hang detector does not indicate any hang.

4.4 Analysis

The false alarm and detection accuracy of this module cannot be quantified since they will vary greatly with the application and function you select to monitor. The best results can be achieved by choosing the correct set of functions to monitor for each application and even the application input. This section will describe the expected false alarm and detection accuracy based on the application behavior.

For large applications the hang detector cannot monitor all of the functions due to hardware limitations. In this scenario the correct set of functions need to be selected to maximize the detection accuracy and minimize the false alarms. All

monitored functions need to have an upper bound to the instruction count. Functions that depend on input size or loop forever should be either omitted or the code should be refactored to call an auxiliary function inside of the (infinite) loop, and the monitoring should be performed on the auxiliary function. This should allow the main function to loop forever without the detector raising any alarm while still monitoring a large amount of the executing code.

When a function is selected for monitoring, the hang detector counts the instructions executed by all of that function's children as well, unless the child function is also selected for monitoring. This allows the user to analyze more functions than the number of selected functions.

Detection accuracy can be maximized by analyzing functions that take the largest execution time in the program. These functions are the most susceptible to faults since they spend the most time on the CPU. Since the hang detector raises an alarm once an upper bound of executed instructions is reached, any hang contained within the monitored functions should be detected. Dead locks can be detected as well since these operations often execute the same few instructions until the lock is released. If the lock is not released these instructions will loop forever.

False alarms can be minimized using a high multiplier for the instruction count threshold. The hang detector raises an alarm every time the instruction count is larger than the multiplier * threshold value. With a higher multiplier the hang can be declared at a much later time increasing the detection latency. A true hang can still be detected because the instruction count goes up forever and the multiplier * threshold will eventually be reached.

For any function that is not monitored and whose callers are not monitored, a hang will go undetected. Likewise, if the control flow jumps between two monitored functions, the hang detector will not be able to detect this problem. In order to avoid this missed detection, the hang detector can be extended to count multiple functions simultaneously. When entering specific child functions, the instruction counter of the parent can continue counting. This is analogous to grouping the parent function with the children. However this method will reduce detection time by also monitoring at a low granularity.

5 Fault Injector

This chapter presents the design and implementation for two fault tolerance modules. In order to validate these mechanisms, it is crucial to have a controlled test environment to simulate faults as they occur in real systems. For this purpose we developed a hardware fault injector for use with the Leon3 processor. This fault injector allows us to modify bits in various signals of the processor's pipeline during the execution of any specific instruction within a target application. Injections can be performed in the program counter (PC), decoded instruction, memory read/write data, and various other data path signals. This injector also allows us to perform bit flips, stuck at 0, or stuck at 1 faults, and supports both single and multi-cycle injections. The injector performs injections without interfering with the execution flow.

Typically, injectors fall into two categories: hardware-based such as [30]-[33] and software-based fault injectors such as [34]-[38]. Hardware injectors can be either based on radiation induced injections or pin induced injections. Software methods normally function by halting execution at a trigger point, modifying the value of a variable in memory, register, or disc location and resuming execution. This model is acceptable because it relies on the principle that any hardware fault that affects an application will ultimately cause a modification to a value in memory. While this principle holds true in most scenarios, it does not simulate the more complex ways in which a single hardware fault could affect multiple memory locations. An example of this case is a fault that changes an arithmetic operation

stored to a register used as a memory pointer. Such an error could affect a select set of memory locations and the same set may not be simulated with this model by modifying the pointer in memory. A control flow error, such as a PC injection in a random code location, is also impossible to simulate with this model. In addition, a multi-cycle error may affect consecutive instructions in different ways, due to hardware sharing across different instruction types. Such an error cannot be simulated by a single memory injection.

Hardware fault injectors simulate faults more accurately than software-based injectors [39]. Software injectors can only modify values that software has access to like memory locations or registers. Our hardware injector supports various injection types directly into pipeline signals. Through this method we can produce a behavior in the CPU similar to that which occurs during a hardware fault.

The fault injector presented in this paper can provide more accurate performance analysis than the software injectors. The injector presented here is configured before the application is executed and injections are performed fully within hardware without affecting the control path; therefore, no extra instructions need to be executed after the application is started. Software methods cannot achieve the same execution accuracy because they require a system call and extra software to perform the injection.

In the following sections, we describe the design and an implementation of our hardware fault injector in the Leon3 system as described in section 2.3. The chapter also describes how the fault injector can be used, and some validation experiments that were performed.

5.1 Design Choices

In an effort to achieve the most accurate simulation environment for evaluating software behavior under faults, we developed a hardware fault injector. This fault injector allows for the modification of specific data path signals during the execution of one or multiple instructions in the target application. We believe this approach of injecting faults directly into the data path to model faults is better than the traditional memory manipulation approach.

In real systems, faults most often behave as single-cycle bit flips in a random positioned logic gate. Depending on which gate is affected, the code execution could be affected in various ways. The application could suffer a small data error, or it could modify a counter variable, resulting in a hang or significant application error. Hardware errors can even affect the program counter, causing the execution path of the program to change drastically. Our design allows the injection of faults into various pipeline signals. The signals that we allow injection to are: Program Counter, Instruction Register, Source Operand1, Source Operand2, ALU result, Memory Data Write, Memory Address, Memory Data Read, and Register Result.

The fault injector we designed is able to inject faults in the target application passively, without interfering with the order of operations executed. The fault injector contains a configurable number of control register sets. These registers are programmed prior to the execution of the target application and trigger automatically on the pre-configured instruction. This guarantees that the control flow of the injected program will behave identically to how it would in the presence of the same fault in a real environment.

In the injection of multiple errors in one run, the hardware can be synthesized with a configurable number of control register sets. Each control register set can be preconfigured with one injection target. The target FPGA type used is the limiting factor to the number of sets that can be instantiated. With the addition of each set, the timing becomes tighter and the hardware resources used increase.

When investigating the behavior of hardware faults, we find that the most common faults are caused by radiation particles hitting transistors in the chip. Usually, these faults only cause errors for a short time (around one clock cycle). However, it has been observed that with smaller transistors, the charge takes longer to discharge; sometimes it can last for multiple clock cycles. Even though these faults occur less often (only high energy particles can cause this) [40], we believe that it is necessary to be able to inject this type of fault. This is the reason why we support the injection of long duration faults. The user can specify the amount of clock cycles to modify the injected signal for.

Software-based fault injectors rely on flipping bits to ensure that an error was introduced. However, when injecting long duration faults one cannot simply flip the value of the bit during each cycle of the error; this does not follow real system behavior. Affected paths (depending on the circuit) behave like a stuck at 0 or stuck at 1 bit for the duration of the fault. In our design we consider this behavior and allow the specification of 3 different types of injection: bit-flip, stuck at 0, and stuck at 1. Software-based fault injectors cannot achieve this [41]. In addition, the fault injector also supports the injection of multiple bits of the injection signal.

The fault injector presented in this paper was designed to allow for an accurate simulation of a large variety of fault types on any target application without introducing extra experimental variables. We achieve this by using a robust injection controller that is tightly coupled with the processor pipeline and is configured from software before the target application is executed.

5.2 Implementation

This section describes the implementation details of our hardware fault injector. An outline of the hardware and its integration into the pipeline is displayed in Figure 10. The injector hardware is composed of three components: the injection blocks, the controller, and a Process ID (PID) monitor. The injector consists of 9 injection blocks; each one is responsible for modifying one of the pipeline signals we allow injection to: Program Counter, Instruction Register, Source Operand1, Source Operand2, ALU result, Memory Data Write, Memory Address, Memory Data Read, and Register Result. The PID monitor is used to determine the active process being executed by the processor. The controller contains multiple sets of control registers and a configuration interface (described later) that allows software full access to the registers. The control register set(s) are used for specifying all of the parameters required for injection. Each register set consists of an injection location (PC value), injection type, injection duration, and a bit-mask. When enabled, the controller continuously checks the pipeline PC signal against all PC injection control registers. If the trigger condition matches any of the control sets and the configured PID matches the active PID, an injection occurs.

In addition to the injector, Figure 10 shows multiple 2-1 multiplexors added to the data-path, one for each injected signal. These injection multiplexors are used to select between the normal signal and the injected signal. The controller is connected directly to the select line of the multiplexors, and selects the injected signal whenever the controller is triggered; otherwise, the original signal is passed through.

The injectors are responsible for modifying the pipeline signal as specified by the controller. Figure 10 displays each of the injectors being continuously fed with its specific 32-bit original (not modified) signal coming from the pipeline. For each injection block, the controller specifies the injection type (stuck at 1, stuck at 0, or bit flip), and the injection bit-mask. The Bit-Mask is used as a bit-enable to determine which bits of the data path signal should be modified. All supported injection types - bit flip, stuck at 0, and stuck at 1 - are generated in parallel through an XOR, AND, and OR gate, respectively. The outputs of the 3 gates pass through a multiplexer, which uses the injection type from the controller to determine which injected signal to output. The injection blocks continuously provide their specific injected signals back into the pipeline. Whenever the controller is triggered, it enables the appropriate injection multiplexor in the pipeline to select the injected signal instead of passing the original signal.

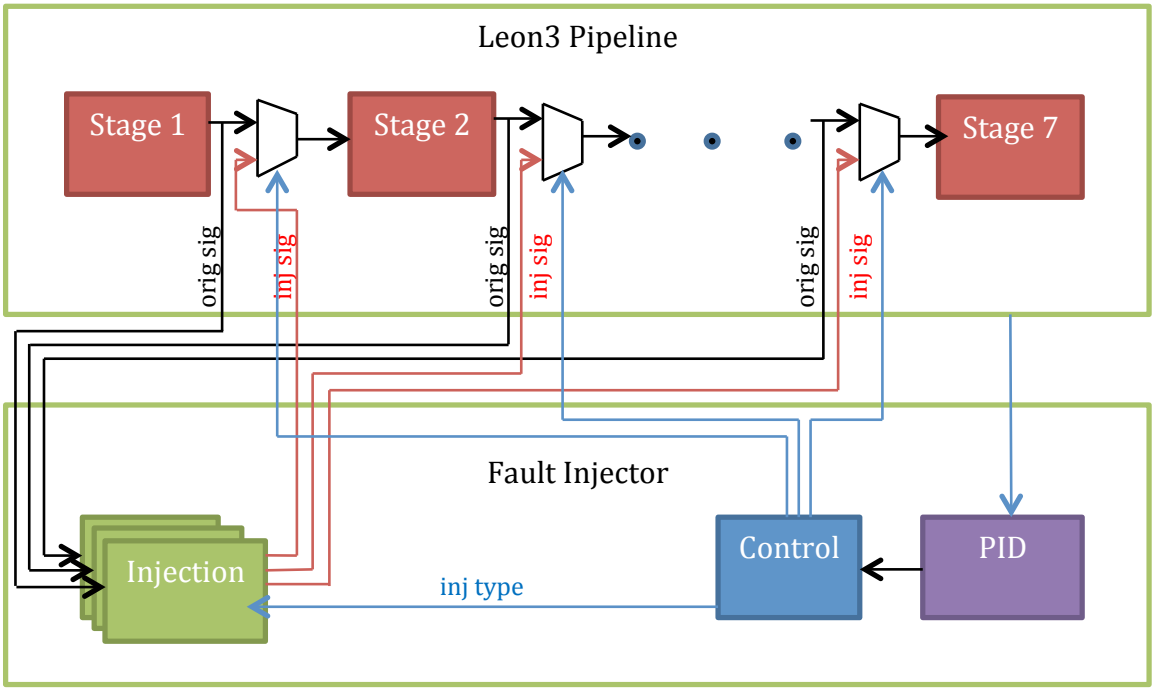

Figure 10: Fault injector hardware outline and integration into Leon3 pipeline

As shown in Figure 11, the injector controller is composed of control register sets and control blocks. The control register sets are used for specifying injection parameters. Each set is used to specify one injection. The number of register sets can be specified before synthesis. This number will represent the number of injection targets the injector will support. Each register set is composed of 5 32-bit registers. Register 0 specifies the Program Counter value at which the injection should occur. Register 1 is used as a mask value to indicate which bits in the data path signal to enable injection for. Register 2 specifies the type of injection to perform. The type register indicates single or multiple cycle error, and the location of the fault (which data path signal to modify). Register 3 is only used for multiple cycle error types; it indicates the number of cycles to assert the error for. Register 4 is used to indicate the PID of the process to inject.

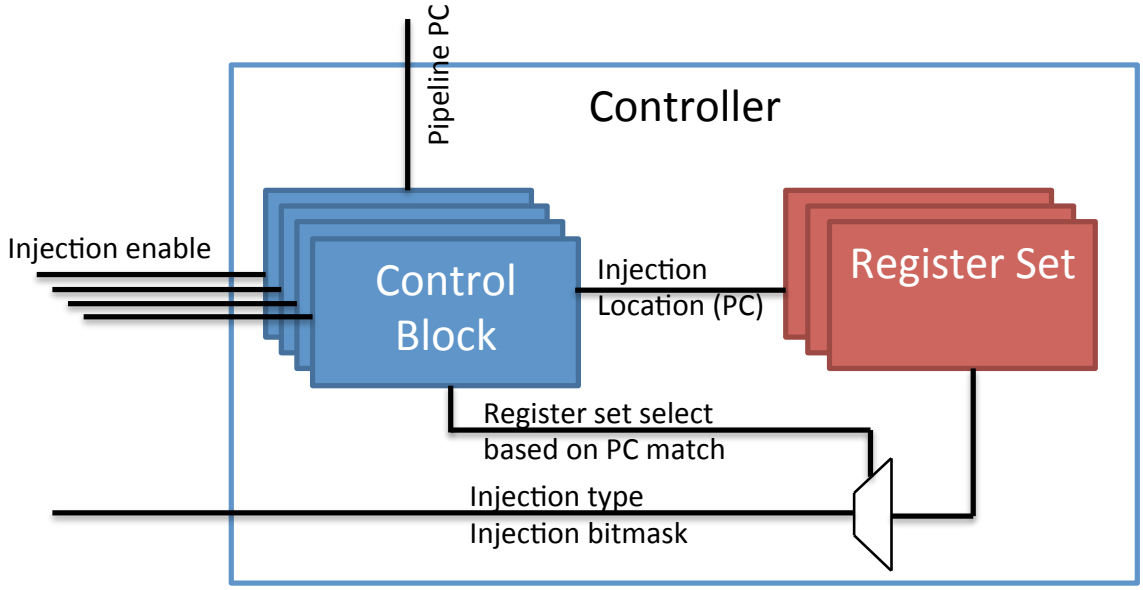

Figure 11: Fault injector controller block outline

The controller contains one control block for each supported injection signal. These control blocks are responsible for triggering an injection whenever both the injection PC location and configured PID match the active PC and PID of the processor. Each control block compares the processor's PC against all of the configured PC injection values (from the register sets) in parallel. If a match is found by any of the control blocks, the matched register set's injection type and bit-mask are forwarded to all of the injection blocks. In addition, the active control block will enable its appropriate injection multiplexor to select the signal from the injector.

In addition, the controller also comprises a configuration interface. The configuration interface gives software access to the control registers. The registers can be accessed from software by the SparcV8 CPOP1 instruction as depicted in Figure 12. The communication interface monitors the instruction register, and various operand data for every operation executed in the pipeline. When the op code matches the CPOP1 instruction, it triggers a read or a write operation

(depending on bit 8) to the fault injector registers. The field [6:0] of the instruction indicates which of the injector registers to access. Upon a write to the fault injector, the data written is read from a register in the CPU register file specified by bits [18:14] of the instruction. On a read instruction, the data from the fault injector is stored into the CPU register specified by bits [29:25] of the instruction. This is accomplished with the use of an additional multiplexor added to the destination register data.

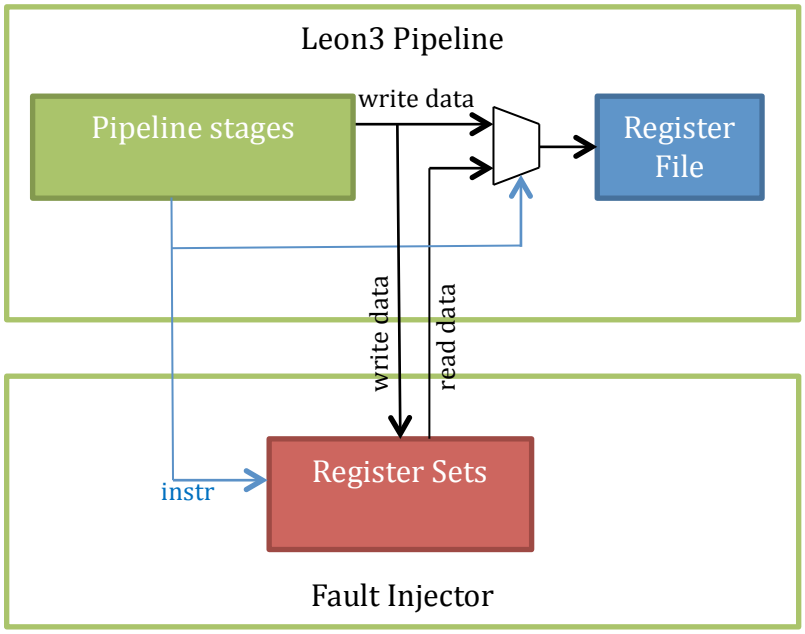

Figure 12: Fault injector configuration interface

We developed a small module to keep track of the active PID on the processor. This module works by monitoring the SPARC alternate store instruction to address 0x200. The SPARC Memory Management Unit (MMU) uses this value to determine the index into the Page Directory for the active process. The Linux operating system we used utilizes the same index as the PID value. By keeping track of this value, we always know the active PID on the processor.

In order to use this injector easily with any application, a software stub was written, which programs the injector with the desired injection location and executes the target program. The Injection PC, fault type, mask, duration, and target application can all be specified in the command line. In order to ensure the injection is performed in the target application, the software stub performs a fork. The child process writes its PID to a pipe and executes the target. The parent application receives the PID and programs the fault injector accordingly.

5.3 Validation Experiments

The fault injector was evaluated on the hardware setup displayed in Figure 4 from section 2.2. To validate the fault injector, a small software routine was written that performs each injection signal type and checks if the correct injection was performed. For each of the 9 supported injections a small test code with a sanity check was written to test the functionality of the injector. Under normal execution, the sanity check prints “fail”. However, with the correct injection in the test code the sanity check prints “success”.

For each injection type, the test application first configures the hardware registers with the appropriate injection location and type. The injection type and location are manually selected to alter the execution in a controlled manner. After configuration, the test code is executed. During execution, the configured injection occurs and modifies the behavior of the test code. At the end of the test code, a sanity check is performed and a print statement is generated to indicate whether the injection was successful or not. As an example following the test case for the ALU injection check:

```
1: int a = 2, b = 1, c, d;
2: c = a + b;
3: d = a;
if(d == c){ success}else{fail}
```

For this test code, the injector was configured to flip the LSB of the ALU output during the execution of line 2. Under normal operation c will equal 3 but with the injected fault the result should be 2. Every injected pipeline signal was validated in a similar fashion. The tests succeeded for all injection types.

In addition to the above test, we also validated the long duration faults. This test was performed using the Altera hardware debugger provided in the Quartus toolset called Signal Tap. This hardware allowed us to view a snapshot of the real time signals from the injector and processor pipeline. The long duration fault was executed with all injection types, using the same software routine as before, but with the long duration fault enabled. In all cases the Signal Tap waveforms showed the injection assertion and the pipeline bits being manipulated for the configured number of clock cycles.

6 Related Work

This thesis presents a technique for providing fault tolerance through the use of a Fault Tolerance Core. Three modules used to describe this topic are presented: a checkpoint recovery process, an application hang detector, and a fault injector. This section provides a list of related work to all of the four topics covered in this paper.

6.1 Related Work on Fault Tolerance Core

The Fault Tolerance Core uses perturbation based error detectors. These detectors are aware of certain patterns in execution for each application and collect runtime information to determine any irregularities in execution. The work presented in [17] is a related work that is based on the same concept. In this thesis four perturbation based detectors are presented: historical variable value, dynamic variable value, bit-invariance, and a bloom filter. This thesis presents the coverage of using such detectors; however, it does not address the issue of collecting the runtime information in a way to minimize performance overhead. The FTC is designed to be an interface for using such perturbation based hardware modules. The FTC collects runtime information from the CPU and delivers it to the hardware modules in a way to designed to minimize the impact on the CPU core.

The work most closely related to the FTC described in this paper is the Reliability and Security Engine (RSE) [18], a hardware reliability framework for integrating programmable hardware modules that provides application aware fault tolerance. The hardware modules used by the RSE are configured for each application to provide the most efficient fault tolerance. The hardware modules

receive runtime information, which is used for determining any perturbations in the execution. The general concept of this design is very similar to the FTC. The main difference here is that the RSE is tightly coupled with the CPU pipeline and significantly impacts the critical path. This component is designed to receive signals directly from the CPU pipeline. This requires the RSE to be incorporated in the CPU core, requiring internal pipeline signals to pass trough an additional number of gates. In addition, the number of transistors added to the core will complicate routing and placement, which will increase the critical path. The proposed FTC only collects a small number of pipeline signals, and routes them away from the core. This allows the FTC to be located away from the core. The FTC uses buffers for the collected pipeline signals to allow the processor to execute freely ahead of the fault detectors.

6.2 Related Work on Checkpoint Recovery

The second fault tolerance topic that is addressed in this paper is the Checkpoint Recovery (C/R) module described in chapter 3. A significant amount of research has been devoted to this topic. Berkley Labs Checkpoint Restart (BLCR) [19] is one of the most well-known application checkpoint mechanisms. This C/R implementation uses kernel and user level support for providing checkpoint transparent to the application. Applications can be checkpointed without any modification. This mechanism checkpoints the application's registers, virtual address space, and open files. The original implementation was a rudimentary full application checkpoint. Various extensions have been developed for this C/R mechanism including an incremental checkpoint extension. The said extension

propagated the dirty bit of the page table entries to the user level through a kernel patch. In this fashion the user level checkpoint service can make use of this data to determine the application's dirty pages within a checkpoint interval. Another incremental checkpoint mechanism is TICK [22]. This C/R implementation performs systemwide incremental or full checkpoint. It only supports uniprocessor systems and is implemented as a kernel thread. This mechanism is fully implemented in the kernel, and maintains the state of the entire system. Pickpt [23] is a page-level incremental checkpoint process. This implementation stores checkpoints to disk, and incorporates some space-efficient techniques for minimizing disk space usage. The checkpoint mechanism in [24] is very similar to the proposed checkpoint in Chapter 3. It is an incremental transparent application checkpoint. It relies on kernel modifications to determine pages requiring checkpointing. This implementation also uses internal memory for storing the checkpoint data. The difference between this checkpoint and the one proposed in this thesis is that ours uses a hardware DMA module for performing the copy of the checkpointed memory.

In message passing parallel applications checkpointing is a more complex problem. Messages can be sent and received out of order and it becomes difficult to ensure the whole application's state is checkpointed correctly. In this category there are two types of checkpointing mechanisms: coordinated and uncoordinated. CoCheck [20] is a coordinated C/R mechanism for checkpointing message passing parallel systems. This mechanism checkpoints every thread of the process and takes a snapshot of the state of the inter process messages. The threads are checkpointed using checkpoint libraries used in Condor [42]. The CoCheck system uses a resource

manager that receives checkpoint requests and coordinates all of the threads and the messaging protocol to achieve a state that can be saved and later restored.

An uncoordinated checkpoint mechanism for message passing applications is MPICH-V [21]. This mechanism does not require any coordination of the messaging protocol. Through message logging, the C/R implementation can checkpoint the full application at any time. The out of order messages are known at the time of the checkpoint from the log. A hardware assisted uncoordinated checkpoint mechanism for shared memory multiprocessors is REVIVE [43]. In this C/R implementation the directory controller is modified to perform direct logging of memory writes. The logs are kept as part of a checkpoint and are replayed in case of a rollback.

6.3 Related Work on Application Hang Detection

Traditional crash and hang detectors have been implemented as watchdog mechanisms. Gouda and McGuire [44] provide descriptions and analysis of various heartbeat protocols. These detectors function with the use of a watchdog timer process that requires repeated heartbeat messages from the target application or system to determine whether it is still functioning.

Other hang detectors have been designed to use OS information to determine the health of an executing program. An example is [45], which uses system calls to determine hangs. In this implementation, application profiling is used to determine normal system call patterns. If the system call pattern is violated or no calls occur, a fault is declared. The detector implemented in [46] uses a very similar principle for detecting application hangs but with more OS monitors. Besides system calls, the mentioned implementation uses OS signals, task schedule timeout, waiting times on

semaphores, holding times on critical sections, process exit codes, and I/O throughput.

Hang detectors have also been implemented using hardware performance counters [47]. These performance counters are available in most modern off-the-shelf CPUs. The counters indicate time and can be controlled through software. In [47] the performance counters are used to determine execution time required for various blocks of code. For each monitored code section additional functions are called to configure the performance counter, one for the entry and one for the exit. Each block of code has a maximum execution time determined through profiling. This approach is very similar to the hang detector described in this thesis. The difference is that the performance counter method requires execution overhead for controlling the counters. The hang detector described in this thesis does not require any modifications to the application and does not incur any overhead other than the initial hardware configuration.

6.4 Related Work on Fault Injection

For a long time, fault injectors have been used to validate the dependability of computing systems. Fault injectors can be categorized in two main classes: software-based and hardware-based. Hardware-based fault injectors mainly use radiation or manipulating voltages at the pins of a CPU. This type of fault injection is better at providing an accurate simulation of faults because it can modify bits in locations that cannot be accessed by other means.

RIFLE [30] is an example of a hardware fault injector that performs injections through CPU pins. This framework produces deterministic faults that can

be reproduced. It is flexible enough to be adapted to a range of target systems. MESSALINE [31] is another pin-level fault injector. This system allows stuck-at and long duration faults to be injected in the target CPU. It has a management module used to automatically generate fault injection sequences.

FIST [32] is an injector that uses radiation to cause single or multiple bit-flips. This design enables the injection of faults deep within a CPU in locations that are inaccessible to software code or pins. Moreover, it has a very random distribution of faults similar to real errors. The downside of this injector is that there is no control over the injection. This means that the same injection cannot be performed twice.

Reference [33] describes a fault injector designed for VHDL simulation. This injector is integrated into a commercial simulator and can perform gate, register, or chip level injections. It provides automatic injection into target models. It supports a variety of fault models like bit-flips, stuck-at, and long duration faults. This mechanism allows for a very detailed view of the fault propagation and error manifestation. This tool cannot be used for full CPU VHDL models. It can only handle less medium-complexity models.

Many software fault injectors have been used in validation. Software injectors are easier to integrate into systems, and can still provide valuable details of the dependability of the target system. FERRARI [34] can inject memory and bus faults. It uses software traps to perform injections. Faults can be injected either based on a timer or a program counter value. The fault types supported are permanent or transient in a memory address, or a data line. FTAPE [35] is another

software-based fault injector. This injector can introduce bit-flip faults into user-accessible registers, memory locations, and the disk subsystem. FTAPE uses fault injection drivers added to the operating system to perform the injections. XCEPTION [36] is an injector that uses the processor's exception to trigger faults at specific addresses. It is implemented as an exception handler. This injector uses a mask to modify either single or multiple bits in either a register or a memory location. The mask is used to determine which bits of the signal to modify with a stuck-at or bit-flip fault. EXFI [37] uses trace exceptions common in commercial CPU's. The trace exception handler is modified to perform injections. This tool can inject single bit-flip transient faults in any part of the application or user registers.

NFTAPE [38] is a more versatile fault injector designed to test a full heterogeneous distributed system by supporting a large variety of injectors. This injector uses a common control mechanism and triggers to manage injections across the whole system. It uses various injectors including a hardware-based LAN injector, driver-based injectors, debugger-based injectors, target-specific injectors, and performance-fault injectors. It can inject faults in the form of bit-flips in registers, kernel and application virtual memory, random physical memory, LAN connections, and I/O communication. This injector also supports a complete variety of trigger types: path-based, time-based, and event-based.

7 Conclusions

This thesis presents a novel approach for providing fault tolerance through the use of a heterogeneous Fault Tolerance Core (FTC). This core is designed as a framework for incorporating various application-aware fault tolerance hardware modules. It collects application runtime information directly from the processor pipeline and forwards it to the hardware modules. The hardware modules use this data for determining perturbations in target applications.

A hardware-assisted checkpoint module, and an application hang detector hardware module have been developed towards achieving the goal of a FTC. All hardware modules have been implemented and tested on an open source Sparc V8 Leon3 processor synthesized on an Altera Stratix III FPGA. The operating system used for the experiments was the Sparc Linux 3.4.4 distribution built around the 2.6.36 Linux Kernel.

The checkpoint module provides incremental checkpoint services to applications with an overhead no larger than 4.5% with a 1 second checkpoint interval. This checkpoint mechanism is implemented as an incremental checkpoint and uses the dirty page bits in the page table entries for determining dirty pages during each checkpoint interval. The checkpoint mechanism was validated by checkpointing and recovering the GCC compiler.

The application hang detector monitors the number of instructions executed inside each user-defined function. This module runs in the background and does not interfere with application execution; therefore it does not incur any overhead. This

mechanism requires no modifications to the target application. The hang detector has been validated with the use of a hardware fault injector also described in this thesis.

For validating the functionality of our fault tolerance modules a hardware fault injector has been developed for the Leon3 system. The fault injector can inject faults directly into the processor pipeline. It supports bit flip, stuck at 0, stuck at 1, long and short duration faults in 9 pipeline signals. This injector has been used in all validation experiments described in this thesis.

References

[1] P. Shivakumar, M. Kistler, S. Keckler, D. Burger and L. Alvisi, 'Modeling the effect of technology trends on the soft error rate of combinational logic', Proceedings International Conference on Dependable Systems and Networks, 2002.

[2] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta and C. Dai, 'Impact of CMOS process scaling and SOI on the soft error rates of logic processes', 2001 Symposium on VLSI Technology. Digest of Technical Papers (IEEE Cat. No.01 CH37184), 2001.

[3] T. Karnik, B. Bloechel, K. Soumyanath, V. De and S. Borkar, 'Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18 Î¼', 2001 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.01CH37185), 2001.

[4] G. Saggese, N. Wang, Z. Kalbarczyk, S. Patel and R. Iyer, 'An Experimental Study of Soft Errors in Microprocessors', IEEE Micro, vol. 25, no. 6, pp. 30-39, 2005.

[5] S. Borkar, 'Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation', IEEE Micro, vol. 25, no. 6, pp. 10-16, 2005.

[6] R. Lyons and W. Vanderkulk, 'The Use of Triple-Modular Redundancy to Improve Computer Reliability', IBM Journal of Research and Development, vol. 6, no. 2, pp. 200-209, 1962.

[7] A. Cooper and W. Chow, 'Development of On-board Space Computer Systems', IBM Journal of Research and Development, vol. 20, no. 1, pp. 5-19, 1976.

[8] M. Mueller, L. Alves, W. Fischer, M. Fair and I. Modi, 'RAS strategy for IBM S/390 G5 and G6', IBM Journal of Research and Development, vol. 43, no. 56, pp. 875-888, 1999.

[9] *Intel® Xeon® Processor E7 Family*, white paper, Intel Corp., 2011.

[10] C. Keltcher, K. McGrath, A. Ahmed and P. Conway, 'The AMD opteron processor for multiprocessor servers', IEEE Micro, vol. 23, no. 2, pp. 66-76, 2003.

[11] V. Khorasani, B. Vahdat and M. Mortazavi, 'Analyzing Area Penalty of 32-Bit Fault Tolerant ALU Using BCH Code', 2011 14th Euromicro Conference on Digital System Design, 2011.

[12] S. Reinhardt and S. Mukherjee, 'Transient fault detection via simultaneous multithreading', ACM SIGARCH Computer Architecture News, vol. 28, no. 2, pp. 25-36, 2000.

[13] R. Chillarege and N. Bowen, 'Understanding large system failures-a fault injection experiment', [1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers, 1989.

[14] N. Wang, J. Quek, T. Rafacz and S. Patel, 'Characterizing the effects of transient faults on a high-performance processor pipeline', International Conference on Dependable Systems and Networks, 2004, 2004.

[15] Liming Chen and A. Avizienis, 'N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation', Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years, 1995.

[16] A. Shye, J. Blomstedt, T. Moseley, V. Reddi and D. Connors, 'PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures', IEEE Trans. Dependable and Secure Comput., vol. 6, no. 2, pp. 135-148, 2009.

[17] P. Racunas, K. Constantinides, S. Manne and S. Mukherjee, 'Perturbation-based Fault Screening', 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 2007.

[18] N. Nakka, Z. Kalbarczyk, R. Iyer and J. Xu, 'An architectural framework for providing reliability and security support', International Conference on Dependable Systems and Networks, 2004, 2004.

[19] P. Hargrove and J. Duell, 'Berkeley lab checkpoint/restart (BLCR) for Linux clusters', J. Phys.: Conf. Ser., vol. 46, pp. 494-499, 2006.

[20] G. Stellner, 'CoCheck: checkpointing and process migration for MPI', Proceedings of International Conference on Parallel Processing, 1996.

[21] G. Bosilca, A. Bouteiller, F. Cappello and S. Djilali, 'MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes', Supercomputing, 2002.

[22] R. Gioiosa, J. Sancho, S. Jiang and F. Petrini, 'Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers', ACM/IEEE SC 2005 Conference (SC'05), 2005.

[23] J. Heo, S. Yi, Y. Cho, J. Hong and S. Shin, 'Space-efficient page-level incremental checkpointing', Proceedings of the 2005 ACM symposium on Applied computing - SAC '05, 2005.

[24] L. Wang, Z. Kalbarczyk, R. Iyer, H. Vora and T. Chahande, 'Checkpointing of control structures in main memory database systems', International Conference on Dependable Systems and Networks, 2004, 2004.

[25] "Leon3 multiprocessing CPU core", http://www.gaisler.com/doc/ leon3-product-sheet.pdf.

[26] J. Nakano, P. Montesinos, K. Gharachorloo and J. Torrellas, 'ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers', The Twelfth International Symposium on High-Performance Computer Architecture, 2006., 2006.

[27] T. J. Dell, "A White paper on the benefits of chipkill-correct ECC for PC server main memory," IBM Microelectronics Division, Nov, 1997.

[28] X. Jian, N. DeBardeleben, S. Blanchard, V. Sridharan and R. Kumar, 'Analyzing Reliability of Memory Sub-systems with Double-Chipkill Detect/Correct', 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing, 2013.

[29] Y. Li, S. Malik and A. Wolfe, 'Performance estimation of embedded software with instruction cache modeling', Proceedings of IEEE International Conference on Computer Aided Design (ICCAD), 1995.

[30] H. Madeira, M. Rela, F. Moreira and J. Silva, 'RIFLE: A general purpose pin-level fault injector', Dependable Computing â€" EDCC-1, pp. 197-216, 1994.

[31] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins and D. Powell, 'Fault injection for dependability validation: a methodology and some applications', IIEEE Trans. Software Eng., vol. 16, no. 2, pp. 166-182, 1990.

[32] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson and U. Gunneflo, 'Using heavy-ion radiation to validate fault-handling mechanisms', IEEE Micro, vol. 14, no. 1, pp. 8-23, 1994.

[33] J. Baraza, J. Gracia, D. Gil and P. Gil, 'A prototype of a VHDL-based fault injection tool', Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2000.

[34] G. Kanawati, N. Kanawati and J. Abraham, 'FERRARI: a tool for the validation of system dependability properties', [1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing, 1992.

[35] T. Tsai, R. Iyer and D. Jewitt, 'An approach towards benchmarking of fault-tolerant commercial systems', Proceedings of Annual Symposium on Fault Tolerant Computing, 1996.

[36] J. Carreira, H. Madeira and J. Silva, 'Xception: a technique for the experimental evaluation of dependability in modern computers', IIEEE Trans. Software Eng., vol. 24, no. 2, pp. 125-136, 1998.

[37] A. Benso, P. Prinetto, M. Rebaudengo and M. Reorda, 'A fault injection environment for microprocessor-based boards', Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270), 1998.

[38] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk and R. Iyer, 'NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors', Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000, 2000.

[39] H. Ziade, R. Ayoubi and R. Velazco, 'A Survey on Fault Injection Techniques', The International Arab Journal of Information Technology, vol. 1, no. 2, pp. 171-186, 2004.

[40] C. Lisboa, M. Erigson and L. Carro, 'System Level Approaches for Mitigation of Long Duration Transient Faults in Future Technologies', 12th IEEE European Test Symposium (ETS'07), 2007.

[41] Mei-Chen Hsueh, T. Tsai and R. Iyer, 'Fault injection techniques and tools', Computer, vol. 30, no. 4, pp. 75-82, 1997.

[42] D. Thain, T. Tannenbaum and M. Livny, 'Distributed computing in practice: the Condor experience', Concurrency Computat.: Pract. Exper., vol. 17, no. 2-4, pp. 323-356, 2005.

[43] M. Prvulovic, Zheng Zhang and J. Torrellas, 'ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors', Proceedings 29th Annual International Symposium on Computer Architecture, 2002.

[44] M. Gouda and T. McGuire, 'Accelerated heartbeat protocols', Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183), 1998.

[45] S. Forrest, S. Hofmeyr, A. Somayaji and T. Longstaff, 'A sense of self for Unix processes', Proceedings 1996 IEEE Symposium on Security and Privacy, 1996.

[46] A. Bovenzi, M. Cinque, D. Cotroneo, R. Natella and G. Carrozza, 'OS-level hang detection in complex software systems', IJCCBS, vol. 2, no. 34, p. 352, 2011.

[47] L. Wang, Z. Kalbarczyk, W. Gu and R. Iyer, 'Reliability MicroKernel: Providing Application-Aware Reliability in the OS', IEEE Transactions on Reliability, vol. 56, no. 4, pp. 597-614, 2007.