Reliable Runtime Architecture for Multiprocessor Systems on Chip

Dimitrios Skarlatos

A Dissertation

Presented to the Faculty

of Technical University of Crete

in Candidacy for the Diploma

of Electronic and Computer Engineering

Advisor: Professor Dionisios Pnevmatikatos

Committee: Professor Apostolos Dollas and

Professor Ioannis Papaefstathiou

May 2014

© Copyright by Dimitrios Skarlatos, 2014.

All rights reserved.

Abstract

Mission critical applications rely on both hardware- and software-approaches for fault-tolerance. With the adoption of multiprocessor systems on chip (MPSoCs), processor fault-tolerance with modular redundancy has become a major issue, cost and performance wise. In this thesis first , we augment a task-parallel runtime system with support for transparent checkpoints of task data that may be written during task execution and seamlessly rerun failed tasks. The system can recover from transient errors during task execution within a single core by rerunning the failed task, as well as from permanent errors that disable a worker core by redistributing work among remaining cores. We have evaluated our implementation using six benchmarks and found that checkpointing incurs a performance overhead of 8% on average, mainly due to the cost of memory copies, and only a negligible space overhead due to the recycling of checkpoint memory. Then, in order to protect the workers runtime system beyond the execution stage, we present ASGUARDIAN, a lightweight hardware mechanism based on a task-oriented model for general programmability. The ASGUARDIAN features both store-and-forward and cut-through capabilities to reliably transfer task descriptions and arguments between main memory and available worker cores. It also isolates the workers from accessing the main memory. A hardware prototype has been implemented on a Xilinx ML605 FPGA board using the widely-used ARM AMBA protocol. Introducing the ASGUARDIAN reliability features results in a 8% average overhead on hardware resources for a configuration with four Microblaze cores. The performance overhead for the store-and-forward and cut-through implementations were 2.3x and 1.2x respectively against an unprotected, shared memory system. When compared against an -unprotected- scratchpad-based memory system, the store-and-forward version showed an overhead of 1.7x, while the cut-through version showed a speedup of 6% on average.

Acknowledgements

A part of this thesis was implemented during my internship at the Computer Architecture and VLSI Laboratory at ICS, FORTH under the advising of Dr. Polyvios Pratikakis. In addition this thesis was in the context of the FP7 DeSyRe project (287611).

I would like to thank my advisor Professor Dionisios Pnevmatikatos, for giving me the chance to do my internship at the CARV Laboratory at ICS, FORTH during the summer of 2012 where my career as a researcher begun. For his dedicated mentorship throughout the last two years, for giving me enough freedom to pursue my own goals and projects even when they were unrelated to this thesis, for always asking the right questions despite how difficult they maybe to answer, and finally for introducing me to the world of computer architecture. Professor Apostolos Dollas for his valuable knowledge and guidance related to logic circuits and FPGA-prototyping and most of all for his belief that with hard work and dedication everything is possible. Professor Ioannis Papaefstathiou for his ”Kopse ton laimo sou kai vres lisi” engineering mentality and most importantly for believing in my potential and his motivation to pursue my dreams even when there was only a slim chance of success. Polyvios Pratikakis for his mentorship during my internship at the CARV laboratory at FORTH and for his support through my first steps in research. Last but not least, Dimitris Theodoropoulos for his technical and personal support for the duration of this thesis and for always finding a way to point me to the right direction despite the ”darkness” of the hour.

In addition, I would like to thank all my friends for their support during the worst and best of times. Especially Panos, for his continued support through all the Skype calls that we had during the last five years, for his motivation to pursue my goals and for always reminding me that life is much more than studies and work. Giorgos for his continued effort to stop me from talking about research all the time despite my constant failure to do so. Dimitris for all the rides that we had with our motorcycles, and most importantly for reminding me to always keep a low profile. Panos-Alkoolikos for introducing me to binge

drinking and for his constructive criticism. Finally Filippos and Kostas for reminding me that everyone has their own dreams in life and different tools to achieve them.

Moreover I would like to thank Afroditi, for her constant love and support throughout the last six and half years, for believing in my potential even when no one else did, and most importantly for shaping me into the man that I am today.

Finally I would like to thank my family, Aspa, Panagiotis, and especially my mother Efi, for their support all these years, the quality of life that they provided, their motivation to pursue my dreams and because without them I would probably have died from hunger.

To Afroditi, my friends and, my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

System robustness and graceful degradation are becoming increasingly challenging with the explosive adoption of multi- and many-core systems in embedded and mission-critical applications. In addition, failure probabilities are bound to increase with the added cores and complexity of embedded systems [43]. Task-based programming models are becoming popular in the scientific community with the rise of OpenMP v4.0 [8, 12] and similar runtime systems [48], [17], [30], because of their ability to express parallelism at a higher abstraction level than thread programming.Tasks are defined as units of work that perform specific operations and typically they are able to run in parallel. With the use of tasks, programmers are able to utilize hardware resources with greater efficiency but without the need to hardwire threads or develop the specifics of the runtime. The runtime system and the compiler are responsible for mapping tasks to the hardware resources and perform the required synchronization and dependency detection. Although these systems have been developed with performance in mind, reliability remains an outstanding issue in embedded multi-core platforms.

Traditional fault-tolerance methods are based on software or hardware modular redundancy, which introduce overheads in terms of complexity and energy consumption [18]. Many proposed software solutions rely on costly thread or process duplication. Numerous

software- and hardware-based reliability approaches have been proposed [24, 25, 54, 50, 40, 52, 41, 36, 11, 4, 35]. Checkpointing provides fault tolerance by creating snapshots of a systems state so in case of failure the system is able to recover. It is not straightforward, however, when a parallel system is at a consistent state, as all threads must synchronize to ensure the whole system is safe to checkpoint. This reduces the available parallelism, as all threads must wait for a global checkpoint. Perthread checkpointing solves this problem, but it may require the programmer to reason about thread-local and thread-shared data, reason about the invariants of the program data structures and keep track of thread-changes transactionally. This makes the checkpointing of thread programs tedious and error prone.

Algorithm 1 Example unprotected pseudocode of the runtime of the workers.

```
1: while workload ≠ finished do
2:     localTaskID ← findNextTaskID()
3:     localTask ← transferTask(localTaskID)
4:     compute(localTask)
5:     transferCompletedTask(localTask)
6:     updateRuntime()
7: end while
```

Algorithm 2 Example pseudocode of the runtime of the workers with RelyBDT.

```
while workload ≠ finished do
    localTaskID ← findNextTaskID()
    localTask ← transferTask(localTaskID)
    checkpointInOutArgs(localTask)
    compute(localTask)
    while errorDetected do
        restoreCheckpoint(localTask)
        compute(localTask)
    end while
    transferBackCompletedTask(localTask)
    updateRuntime()
end while
```

Algorithm 1 presents an example pseudocode of an unprotected runtime system with each worker executing its workload. As long as there are still tasks to be computed the worker will identify the next available task, compute the result and transfer it back updating

the runtime. We separate the operations of the worker in two phases. The execution phase where the worker performs the computation of the task and the runtime phase where the worker transfers tasks and updates the shared structures. The worker may fail during the execution of each phase causing a computation to fail, compute the wrong result, corrupt shared structures or render the system unable to perform any operation.

This thesis presents RelyBDT and ASGUARDIAN; combined provide a complete reliable solution for task based programs protecting both the runtime stages and the execution stages of the system.

First we introduce RelyBDT, a fault-tolerant runtime system for the reliable execution of task-parallel programs. RelyBDT abstracts over checkpointing for the programmer by performing transparent checkpointing of parallel tasks. RelyBDT recovers from transient task faults by recomputing the faulted task, and from permanent core faults by rescheduling the failed cores tasks to different cores. We have implemented RelyBDT as an extension of BDDT [48], a deterministic task-parallel runtime system. Our evaluation of the RelyBDT using six benchmarks found that checkpointing incurs a performance overhead of 7.93% on average mainly due to the cost of memory copies, and only a negligible space overhead due to the recycling of checkpoint memory. The pseudocode of the runtime system augmented with RelyBDT is presented in algorithm 2. RelyBDT protects the computation stage by creating local checkpoints of the arguments to be altered during the computation of the task. In case of an error detection the checkpoint is restored and the task is recomputed.

Although RelyBDT provides sufficient coverage on a multi-core environment during the execution stage, the worker runtime system remains vulnerable to errors. As a consequence, there can be cases where, due to the transient and permanent errors occurrence, an unreliable worker core may access prohibited memory areas and corrupt vital system and runtime data. Hence, to establish reliable runtime transactions, it is very important to employ a protected mechanism that provides *memory address isolation* (similar to the

sandbox [34] protection mechanism found in many contemporary operating systems), by allowing each worker to access only pre-specified memory areas.

Algorithm 3 Example pseudocode of the runtime of the workers with ASGUARDIAN.

```
while workload ≠ finished do
    localTask ← task_request()
    compute(localTask)
    task_complete()
end while
```

Algorithm 4 Example pseudocode of the runtime of the workers with RelyBDT and ASGUARDIAN.

```
while workload ≠ finished do
    localTask ← task_request()
    checkpointInOutArgs(localTask)
    compute(localTask)
    while errorDetected do
        restoreCheckpoint(localTask)
        compute(localTask)
    end while
    task_complete()
end while
```

Towards this goal, we propose the *ASGUARDIAN*, a hardware mechanism that is able to reliably handle the memory management and transfers needs of the runtime system. In addition to protection, it imposes a minimal hardware resources overhead of 7.86% on average, and provides an average speedup of 6% when compared to an unreliable scratchpad-based system. The pseudocode of the runtime system augmented with the ASGUARDIAN is presented in algorithm 3. The ASGUARDIAN is responsible to identify the next task and transfer it to the local memory of the worker. After the task computation is complete the task is transfered back and the runtime gets updated from the ASGUARDIAN. Algorithm 4 highlights the pseudocode of runtime system with both RelyBDT and ASGUARDIAN.

The contributions of this thesis are the following:

- We introduce RelyBDT, a fault-tolerant runtime system as an extention of the BDDT. RelyBDT creates task checkpoints at runtime on a per worker fashion and is able to recover from transient and permanent faults.

- We present the ASGUARDIAN and its microarchitecture; a novel hardware mechanism that abstracts away memory management and transfer needs of a task-based runtime system, by providing accelerated and reliable transfer capabilities of task descriptors and arguments (metadata) to worker cores with cut-through or store-and-forward approaches.

- We describe the design and implementation of an Application Programming Interface (API) that enhances the programmability and ease of use of the ASGUARDIAN.

- We evaluate RelyBDT on server-grade system using six benchmarks and multiple workloads targeting different application requirements like options pricing, Fourier transformations, linear algebra kernels and parallel sorting.

- We evaluate the ASGUARDIAN with a synthetic benchmark and a range of workloads on a bare-metal task based runtime system with multiple reliable and unreliable task-transfer capabilities. Our experiments have been executed on FPGA-based MPSoC that integrates an ASGUARDIAN module with one master and three worker cores.

The rest of this thesis is organized as follows. Chapter 2 presents traditional types of errors and fault tolerance techniques. Chapter 3 describes related work. Chapter 4 introduces RelyBDT and its evaluation and Chapter 5 presents ASGUARDIAN and its evaluation. Chapter 6 presents the Verification & Validation steps and features of RelyBDT and the ASGUARDIAN, and Chapter 7 concludes this thesis.

Chapter 2

Fault Tolerance

2.1 Outline

This chapter presents the traditional errors, fault tolerance methods, and models. Section 2.2 provides an overview of type of Errors and Data Corruption, Section 2.3 presents the Fault Tolerance and Error Recovery on a system level, Section 2.4 describes Modular Redundancy approaches, Section 2.5 presents Error Detection and Correction techniques and Section 2.6 provides an overview of memory protection.

2.2 Errors and Data Corruption

Single Event Upsets (SEUs) are defined as the state change caused by radiation exposure, cosmic rays (most common soft errors in DRAM cells), electrostatic discharge or other factors. The change of the systems state, from the transistor level to the application level, may disrupt the correct operation of the system. Faults are separated in two basic categories, (i) Transient faults that cause a computation to fail or compute the wrong result, while the hardware remains operational and it suffices to restart the computation. (ii) Permanent faults that cause hardware to fail and be unable to perform any computations.

Data corruption is the category of errors relevant to the system's data and may occur during the reading, writing or processing stages. Data corruption is separated into two types, undetected and detected errors. Detected errors are either permanent and cause irreversible loss of data or are temporary and several mechanisms are able to detect and recover/correct the dataset. Silent or undetected errors represent the worst case of errors because of the inability of the system to perform any kind of operations to protect the data or notify the user. As result, silent errors go unnoticed and may have devastating results.

In both cases the masking of errors, in the electrical, logical, or temporal level may result in the error to not alter the output of the system. In the electrical level, the masking occurs when the electrical disruption or noise does not alter the state of the transistor. Logical masking occurs when the output of a logic gate is not altered from the error, e.g when the inputs of a logic OR gate are both '1' and an SEU turns one of the two signal to '0' the output of the gate remains '1'. In this case the error does not propagate to the output of the system and the error is considered masked. Finally an error is temporally masked when an error occurs in specific time frame which does not end up disrupting the result of the system (e.g the output was used before the SEU).

2.3 Fault Tolerance and Error Recovery

Fault-tolerance is a characteristic of the system and represents the ability of the system to handle disastrous events. The main goal of fault-tolerance is the graceful degradation of the system where the system continues to operate without any loss in the quality-of-service(QoS) or with reasonable loss of quality depending on the severity of the fault.

Error recovery represents the steps that a system has to perform in order to continue operating correctly in case of an error. Roll-forward and roll-back are the two possible methods to recover the system when an error is detected. Roll-forward tries to correct the error without altering the state of the system. Roll-back transfers the system back to a

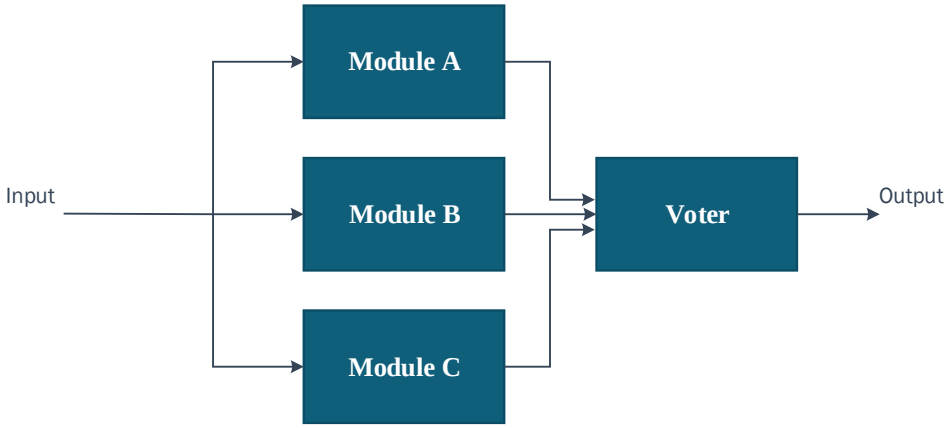


Figure 2.1: Triple Modular Redundancy: By passing the input through three different modules that perform the same operation on the input and by passing the outputs to a voter; the system is able to detect the correct output by choosing the most frequent output.

safe/correct state by using checkpoints. The main overhead of checkpointing is that the system has to store enough information at different intervals in order to be able to recover in case of errors. Finding such intervals may prove a difficult task especially when it comes to parallel programs. Depending on the fault-tolerance level of the system both methods can be utilized for different kind of errors.

2.4 Modular Redundancy

Modular redundancy is based on the assumption that the probability of the occurrence of the same error in the same time in different modules is extremely low and as a result the error will be detected. Dual Modular Redundancy (DMR) provides fault tolerance by duplicating components. One way to use DMR is to have the extra component as a backup and use it when the main component fails. A different approach is to use both components in parallel in order to detect error and correct errors. Lockstep systems execute the same operation in parallel and compare the results. The main drawback of DMR is that when an error is detected the operation that caused the error has to be re-executed in order for the system to correct the error. Triple Modular Redundancy alleviates this problem by using

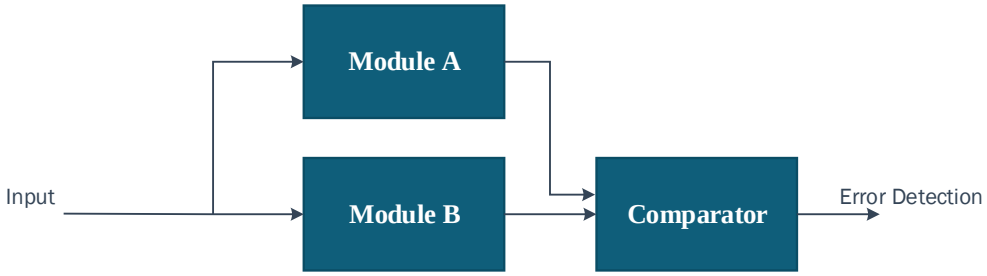


Figure 2.2: Dual Modular Redundancy: By passing the input through two different modules that perform the same operation on the input and comparing the results the system is able to detect when an error occurred but is not able to decide which of the two outputs is the correct one.

three independent modules in parallel and a voting system. In that way the system is able to compare the results and choose the correct outcome. While these methods provide a vital solution to the issue of fault-tolerance the come a the cost of extra resources. Figures 2.1 and 2.2 provide an example of a TMR and a DMR system respectively.

A different approach is to use redundancy in time by executing the same set of instructions on the same hardware resources multiple times and compare the results. This method eliminates the resources overhead and provides fault coverage for transient errors. Permanent faults cannot be covered through redundancy in time because in case of a permanently faulty unit, the results of each run will always be the same (with the same error) and as a result the comparison cannot be trusted.

2.5 Error Detection and Correction

Interconnect and communication error detection utilizes information and coding theory to detect and correct errors that occur during the transfer of data. Repetition codes, Checksums, Parity bits and CRCs are the most common techniques [16, 31, 32, 33]. Repetition codes are based upon a predetermined coding scheme that sends the same block of data a specified number of times. The received blocks are then compared against each other to

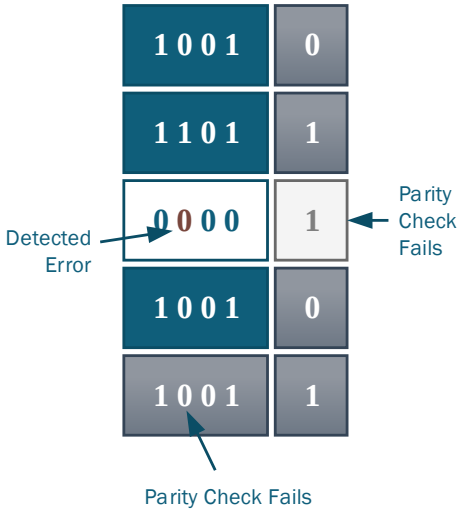


Figure 2.3: Parity:In this table parity bits are inserted for each row and each column. Bits in green boxes represent data bits and bits in gray boxes represent parity bits. The parity bit is '0' for even number of '0' and odd otherwise. In the third row and second column an example of error detection is presented where both parity checks fail.

detect errors. While repetition codes are simple to use are extremely inefficient. Checksum is the arithmetic sum of a fixed word length and may include multiple other Error Detection and Correction techniques. Depending on the complexity of the checksum different trade-offs between performance and protection arise. Parity bits are a simple method that adds a parity bit to a fixed number of source bits based on the number of '1' or '0' in the source bits. Parity bits can detect only single or odd number of errors. Advanced parity bit techniques like horizontal redundancy checks add an extra layer of protection. An example of the parity bit is shown in Figure 2.3. A commonly used hardware technique is the Cyclic Redundancy Checks (CRCs) which is the combination of a single-burst-error-detecting cyclic code and a non-secure hash function. CRCs are useful in detecting burst errors because of their cyclic properties and are very easy to implement in hardware because of the shifting operations that take place.

2.6 Memory Access Protection

Memory protection provides control over memory accesses and usually is part of the operating system (OS). The main insight of the memory access protection mechanisms is to protect memory locations to be accessed from processes that do not have the proper rights and to not allow processes to access un-allocated memory sections. The result of these mechanisms is to usually terminate the violating process. The most common memory protection methods are memory segmentation and paged virtual memory. Protection keys are used in more advanced systems, like the Intel's Itanium [27] and IBM's System/360 architectures [3]. Memory segmentation is based upon dividing the memory into segments. A memory location is characterized by a value that identifies the segment and an offset value that identifies the offset within the segment. Paged virtual memory divides the memory address space into blocks of equal size which are called pages. Special hardware, like the Translation Lookaside Buffer, helps the the operating system in managing the pages and the memory of the system. An application is unable to access pages that has not been explicitly allocated to it and a page fault is generated. Protection keys allocate a specified key to a block of memory. In addition protection keys are given to each application, when a memory access occurs the memory block key and the application key are validated in oder to protect the memory of the system.

Chapter 3

Related Work

3.1 Outline

This chapter provides an overview of the related work and is organizes as follows. Section 3.2 presents related work in Task Parallelism, Section 3.3 highligts related work in Software Reliability and Checkpointing while Section 3.4 presents related work in Hardware Reliability and Checkpointing. Finally Section 3.5 provides related work in Transactional Memory.

3.2 Task Parallelism

Traditional task-parallel programming models try to abstract over threads and offer a higher-level abstraction for expressing parallelism. Cilk [7] is a task based, shared memory, programming model that allows the programmer to specify recursively spawned tasks, which are efficiently scheduled on threads using continuations. Sequoia [39] is a programming language used in development of parallel, hierarchy-aware and portable applications. In these models the programmer is responsible for synchronizing parallel tasks to avoid address space aliasing and races. Figure 3.1 highlights an example of the task based execution model.

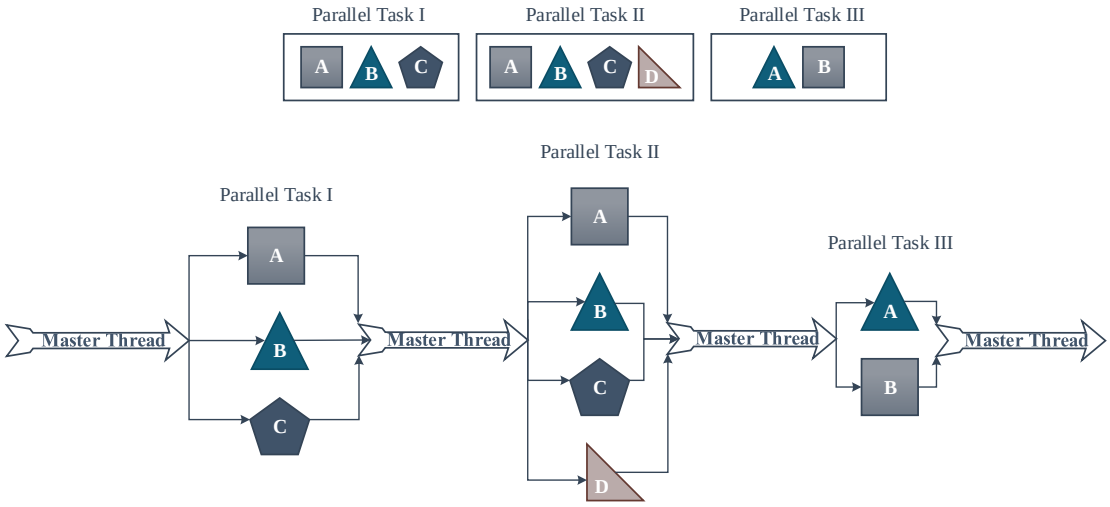


Figure 3.1: Task Model: Parallel Task I, II, III present example tasks with different blocks of code to be executed. The master thread forks over multiple threads where each threads executes a block of code.

Dependence analysis in task-based programming models has been shown to improve performance of general purpose programs [49]. In these systems tasks define their effect *input*, *output* or both (`inout`) on their arguments. The runtime then uses versioned hyper-objects to track task argument dependencies and versioning of objects in order to increase parallelism. To break anti-dependence and output dependencies a versioning mechanism determines the view of an object for each thread.

3.3 Software Approaches

RAFT [53] is a speculative runtime fault tolerance mechanism designed for single-threaded applications with deterministic output. It duplicates the original application and executes both versions in parallel, using double the amount of registers and memory. RAFT speculates the return values of system calls, avoiding synchronization barriers and only verifies values that escape the user space.

Shoestring [21] is a symptom-based instruction duplication technique that provides opportunistic soft error reliability. A compiler analysis is utilized to find vulnerable code, based on a specific set of symptoms (e.g., memory access exceptions). The instruction duplication mechanism selects a tree of instructions to duplicate. Checker nodes are injected along the code to compare the results of the leaf nodes from the original and the duplicated tree.

ReVive [35] is a rollback recovery mechanism based on shared memory multiprocessors with a special hardware mechanism that performs logging and parity updates. The proposed mechanism creates checkpoints at three different consistency levels. Global checkpoints that require all processors in the system to synchronize. On the processor level where each processor creates a checkpoint of its local state but in the same time forcing any communicating processor to create its own checkpoint and finally periodical processor-local checkpoints without interactions with other processors. In addition to the different types of checkpoints, checkpoint separation takes place based on on how the working data are separated from the checkpoint data. Finally checkpoints are grouped in three categories based on the storage type, external, safe internal and unsafe internal. Based on the error detected, different level of checkpoints are used to restore the system.

SWIFT [38] is a single-threaded, compiler-based fault tolerant technique which injects duplicate and comparison instructions at compile time to detect faults. The memory system is protected through the use of error correcting code (ECC). SWIFT uses an optimized control-flow checking mechanism which utilize control blocks with dynamic signatures, to avoid the cost of branch validation code.

ASSURE [42] provides a fault tolerance technique for server applications by implementing rescue points with a moderate performance overhead. Rescue points are locations in the application where error handling takes place and can be found offline. When a “live” error occurs for the first time,a new personal copy of the application is used to search for rescue points that are able to handle the error. Once a suitable point is found, a patch is

applied trough binary injection to the online application. The patched version of the application is able to take checkpoints at the rescue point, so when the same error occurs a rollback can take place and recover from the fault automatically.

Static techniques have also been used in the past to identify *idempotent regions* of code that can be re-executed without checkpointing [10]. Such analyses are orthogonal to our work and can be used complementarily to identify and optimize unnecessary checkpoints, further reducing overhead.

3.4 Hardware Approaches

Hardware based reliability is also a vital solution for task based programs. In [45] the authors present a runtime system for MPSoCs with a master and multiple worker cores. Task-based code annotations in combination with a Java C-to-C compiler that generates the master and worker source codes, provide a complete task-oriented solution for FPGA-based MPSoCs. Moreover, an extension of this work is presented in [46] with fault-tolerant application execution features; the system is protected against transient and permanent faults by creating checkpoints at runtime, similarly to RelyBDT. In addition, it introduces a new set of pragma-based annotations to developers that allows them to provide alternative task implementations, which do not utilize certain worker submodules. This feature allows the runtime to utilize partially damaged workers with undamaged subsystems.

The MADNESS Project [11] presents a system level approach for reliability in NOC-based MPSoCs. In order to increase the fault tolerance of the system a hardware layer that cooperates with the runtime manager is able to migrate the process on a faulty core to working ones.

Interconnect error detection is usually based on link errors. Parity, CRC and SECDED [16, 31, 32, 33] are common ways to protect the network from such faults.

In [9] a hardware mechanism is proposed to accelerate the management of tasks of the StarsSs [30] programming model. While this hardware mechanism increases the performance and scalability of the system, it does not provide fault tolerance.

Masubuchi et al. propose a hardware fault recovery mechanism (FRM) in [28]. The proposed FRM targets server application where checkpoints are acquired and managed by the hardware periodically and the hardware mechanism is able to record the recovery data by using a special Before Image Buffer scheme. The main drawbacks of the proposed system is that all the processors in the system have to synchronize to start the recovery operation, the main memory is rolled back and the cashes are invalidated. While the FRM is able to recover from a number of faults, our system can be combined with a software task-based checkpoint mechanism (like RelyBDT and the mechanism presented in [46]), hence the whole system can be protected without main memory rollbacks, cache invalidations and core synchronization.

Fault tolerance for MPSoCs through dynamic task scheduling has been proposed in [4]. In order to detect errors each task is scheduled two times in different processing elements, and in case of an error detection the task is computed a third time and a voting process takes place. In addition global errors are detected when a processing element fails to complete the computation of a task in a predetermined time frame.

A main difference of the proposed system is that the hardware mechanism that is responsible for the scheduling of the tasks and the global error detection is considered fault-free and that the system requires extra processing elements, one for error detection and two for correction. Hence, restricting the graceful degradation and performance of the system while increasing the hardware resources required, especially for systems with a small number of cores. On the other hand, the ASGUARDIAN is augmented with extra modules in a TMR fashion and is able to protect the main memory of the system when the worker cores fails permanently or transiently while greatly reducing the required hardware resources independently of the number of cores in the system.

Moreover, the proposed mechanism considers faults during the computation of tasks and not during the memory access operations, as a result the main memory may be corrupted during these steps. This scenario is covered from the ASGUARDIAN because the memory accesses are performed in compliance to the the tasks footprint and TMR modules. In addition, RelyBDT is a checkpoint software solution that provided an error detection mechanism is able to recover from transient computation and permanent worker faults without the use of extra hardware resources and thread synchronization. Finally, profiling of the tasks is not required, hence the ASGUARDIAN and RelyBDT are able to support dynamic workloads without alterations.

A difference of our system from other proposals [11, 47] and typical memory protection mechanisms like virtual memory and the MMU, is that our system does not require interrupt support neither requires a real-time OS like [5, 2, 23], which significantly reduces the workers required resources and the complexity of the system.

3.5 Transactional Memory and Memory Management

The notion of rollback and retrying execution is inherent in Transactional Memory (TM) models [22]. Several software transactional memory runtimes use checkpointing or similar techniques to save and restore the state of transactional variables to consistent points. TM, however, is a lower-level model than task-parallelism; TM programs use threads to express parallelism and TM to enforce synchronization. Moreover, TM most often enforces a mutual exclusion semantics of synchronization, whereas runtime dependence analysis in task-parallel models enforces a dataflow semantics that deterministically produces the same result as the sequential program.

Epoch-based cache management (ECM) in collaboration with an explicit bulk prefetcher has been proposed in [29]. The runtime is able to use information provided from the hardware in order to prefetch tasks accordingly. The prefetcher is similar to

an RDMA engine but with the capability to allow software to explicitly transfer memory ranges that correspond to tasks. This work is orthogonal to our proposal and can work as a performance extension to the ASGUARDIAN in order for the system to prefetch tasks in a fault-tolerant mode.

Chapter 4

RelyBDT

4.1 Outline

This chapter presents RelyBDT, a software checkpointing mechanism at the thread-level, build around a task-based runtime system. The rest of this chapter is organized as follows. Section 4.2 provides an overview of the fault model and our assumptions, Section 4.3 presents the design of the checkpoint mechanism and Section 4.4 presents the evalution of RelyBDT.

4.2 Assumptions and Fault Model

The semantics of task-parallel programming models make checkpointing easier and more efficient than in previous approaches targeting thread programs. Namely, a task is a computation that can run in isolation (usually a function call), has clearly defined input and output arguments, and is restricted to only access those. For example, the code `spawn f(input A, output B)` executes function `f(A,B)` in parallel to the rest of the program, taking into account that `f` reads its first and writes its second argument. Figure 5 demonstrates the spawn function where different parts of the Fibonacci function are calcu-

Algorithm 5 Example Fibonacci calculation with the spawn function. Each spawn is a thread which calculates the next Fibonacci value. Finally, the threads are synced and the final result is calculated.

$Fib(n)$
if $n < 2$ **then**
    $returnn$
else
    $x \leftarrow spawnfib(n-1)$
    $y \leftarrow spawnfib(n-2)$
    $sync$;
    $return(x+y)$
end if

lated with different threads, then the threads are synced and the final result is calculated. Arguments can be either read-only (input), write-only (output) or read-write (`inout`). In implicitly synchronized task-parallel models like BDDT, moreover, tasks are guaranteed to have exclusive access to their output and `inout` arguments, because the runtime system automatically detects dependencies between tasks that access the same memory and schedules them correctly to avoid races. In BDDT, a "master" thread executes the main program and spawns parallel tasks, while a set of "worker" threads execute these tasks when all their dependencies are satisfied.

We take advantage of the task semantics and assume that tasks will respect their memory footprint (input and output arguments) even in the presence of faults. That is, we assume that the hardware can either detect faults as soon as they occur [37] or use memory protection mechanisms [38] to detect and stop a faulty task from corrupting any part of memory other than its arguments. This assumption greatly simplifies checkpointing, because the system need only restore task argument memory before retrying the task. Moreover, task arguments are accessed only by the task during its execution, meaning that both checkpointing and restoring from a checkpoint become thread-local operations, without requiring synchronization with other threads. In contrast, checkpointing a multithreaded application requires all threads to stop as any thread can access any memory address.

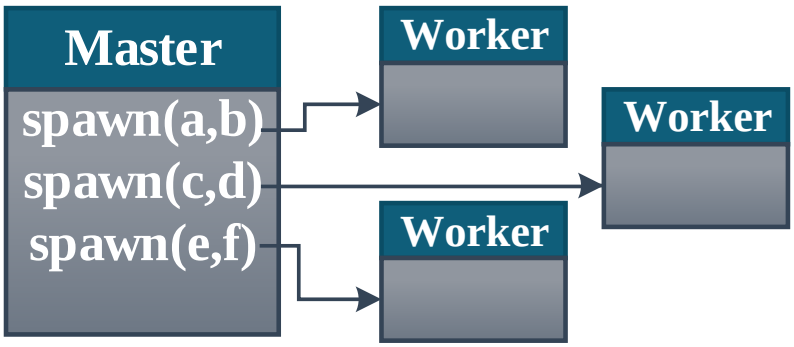


Figure 4.1: Master-Worker model: The master threads spawns tasks to different workers.

We assume two kinds of faults: (i) Transient faults that cause a computation to fail or compute the wrong result, while the hardware remains operational and it suffices to restart the computation. (ii) Permanent faults that cause hardware to fail and be unable to perform any computations. In both cases, we assume that the faults will occur during the execution of tasks, meaning that no faults can happen at the “master” core, nor when a “worker” is running the runtime system between tasks. In future work, we plan to extend the fault model and checkpointing implementation to support these cases.

4.3 Design

The BDDT runtime system uses a ”master” thread to spawn tasks and a pool of ”worker” threads to execute them. Figure 4.1 highlits this model. Under the assumptions described above, all checkpoint operations are run by the worker-threads. It is safe to checkpoint task data in parallel without synchronization, because of the semantics of tasks, namely that a task will never be scheduled concurrently with other tasks accessing the same data. Balancing the checkpointing, fault detection and fault-recovery workload across many threads gives scalability to our design and does not introduce bottlenecks.

4.3.1 Transient Faults

Worker threads checkpoint a task's arguments just before executing it. Each worker thread uses a checkpoint buffer into which it copies the task arguments. We recycle the checkpoint buffer when possible, to avoid the overhead of allocating and deallocating checkpoint memory. The checkpoint only contains the contents of `inout` arguments, because input-only arguments can not get corrupted/altered during execution and output-only arguments are always rewritten during execution.

We assume that transient faults can be detected as they occur, so that task execution stops and the runtime system resumes control of the core. We simulated transient faults using a "coin-flip" test after every task to decide whether the task is faulty and needs to be re-run. We used a uniform-distribution pseudo-random number generator to generate faults with various probabilities. When a transient fault is detected, the runtime system restores the checkpoint by copying the contents of `inout` arguments from the checkpoint buffer to the original locations and retries the task. Note that in reality, a fault may be detected before the end of a task, causing it to be restarted earlier; our fault emulation thus overapproximates the overhead because it always allows the faulty task to terminate before retrying.

4.3.2 Permanent Faults

We assume permanent faults may completely disable a core while a task is running. We simulate permanent faults in the same way, using a "coin-flip" function based on uniform distribution to create various probabilities of failure. Recovery from permanent faults is more complex than for transient faults, because it involves different worker cores detecting the error and redistributing all work scheduled to the faulty worker core.

To detect permanent faults in worker cores we have the runtime in each worker core periodically check for permanent faults in other workers. Upon detection of a permanent fault, one of the live workers will take over recovery: disable the task queue of the faulty

worker core so that no new tasks are scheduled there and redistribute all tasks in that queue to the remaining workers via task-stealing. The recovery worker also takes over the task that was running when the permanent fault occurred, restores its data from the checkpoint buffer of the disabled worker and executes the task locally.

4.4 Experimental Evaluation

We evaluate our implementation using six task-parallel benchmarksrunning on a 3.3GHz, 4-core i5-2500 CPU with 4GB RAM running Ubuntu 12.04 Server. All reported times are averages of 5 runs. We used the following benchmarks because of their argument characteristics, as they represent all the different scenarios and because they have already been used to evaluate the performance and scalability of BDDT.

Black-Scholes is a mathematical model for financial markets, taken from the PARSEC [6] benchmark suite. We used 12,800,000 options and a 292MB dataset, resulting in 100,000 (dynamic) tasks. This application does not have any `inout` arguments, so it requires no checkpointing. The overhead shown only corresponds to the re-execution of tasks.

GMRES computes the generalized minimal residual method for solving non-symmetric systems of linear equations. We used 13,107 nodes and a block size of 128MB, which amounts to 249,717 tasks. This application models the worst case checkpointing scenario, where all tasks use `inout` arguments and memory copy operations become a bottleneck. Moreover, each task uses a small part of memory, requiring lots of small checkpoints. The overhead is the combination of the checkpoint functions and the re-execution of tasks.

FFT is a kernel with Blocking Transpose taken from the SPLASH-2 [51] benchmark suite. We used 16,777,216 Complex Doubles resulting in 28,864 tasks that have strided `inout` arguments that correspond to array tiles, causing overhead due to multiple memory copies per argument.

Cholesky is a linear algebra kernel uses 128×128 matrices, resulting in 5,984 tasks. The tasks have strided `inout` arguments corresponding to tiles of the total array, resulting in several copying operations per argument. Moreover, the tiles are large, and so the checkpoints have a large memory and time overheads due to copying.

Jacobi is a kernel using the Jacobian method for solving linear equation systems. We used a 7168×7168 matrix tiled into 128×128 blocks for 30 iterations, resulting into 94,080 tasks. Jacobi uses two matrices to avoid in-place computation; its tasks read from one and write to the other array, swapping the arrays for the subsequent iterations of tasks. Even though Jacobi uses strided arguments to describe array tiles it incurs no checkpointing overhead, because there are no `inout` arguments that require copying.

Multisort is a task-parallel version of Mergesort similar to Cilksort [7]. We used 20MB of data for 836 tasks. Multisort does not have `inout` arguments, each task uses read-only inputs and a write-only buffer to output its results. Effectively there are no checkpoints for Multisort tasks, and the only overhead corresponds to retrying failed tasks.

4.4.1 Time Overhead

To compute the overhead of checkpointing, we measure the running time of each benchmark using the original BDDT runtime without any support for fault tolerance and checkpointing. We also measure running time using RelyBDT for zero permanent faults, while varying the probability of transient faults.

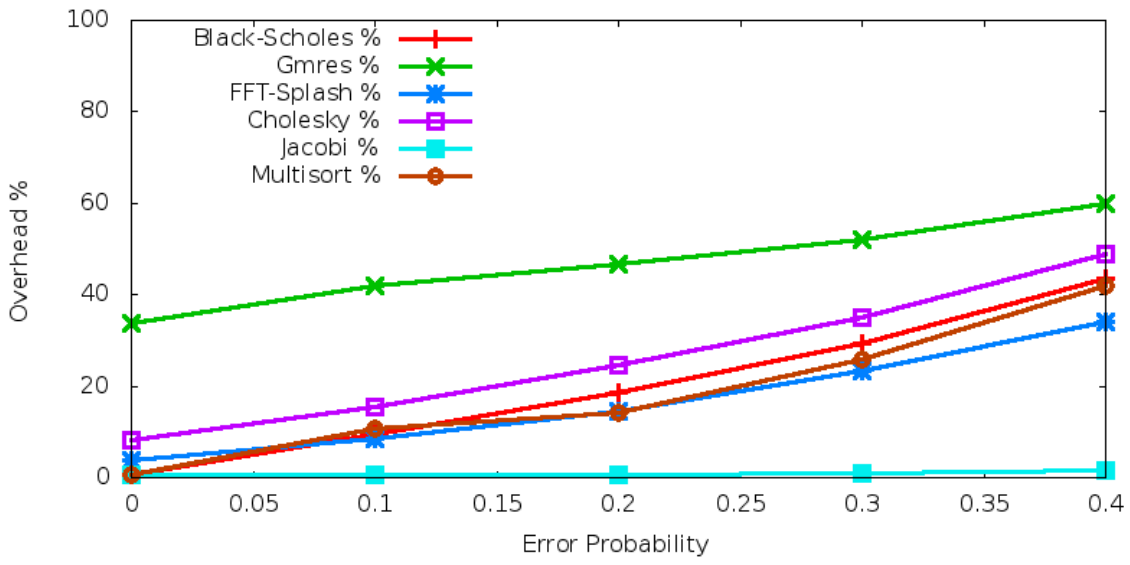


Figure 4.2: Time overhead percentage over no-checkpointing with transient error probability ranging from 0.0 to 0.4 and no permanent errors.

Figure 4.2 shows the overhead incurred by checkpointing as a percentage over the baseline run. Note that the leftmost data points correspond to a transient error probability of zero, in effect measuring the "protection" cost of checkpointing all `inout` arguments in an application, even in the case where no checkpoint is ever used. Benchmark performance varies depending on the cost of allocating space and creating the checkpoint, relative to the cost of the task. As expected, benchmarks that do not require checkpoints (have no `inout` arguments) do not show any checkpointing overhead. The overhead is similarly low on computationally-heavy benchmarks like FFT and Cholesky, because the cost of allocating the checkpoint and copying the data is dwarfed by the time spent in the actual task. On the other hand, applications like GMRES that have small, fast tasks operating on large data footprints suffer significant overhead, because creating the checkpoint takes time comparable to the actual task execution.

Figure 4.2 also shows three other data points per application, for executions with probability of transient error being 0.1, 0.2, 0.3 and 0.4 (10% to 40% error rate). Note that

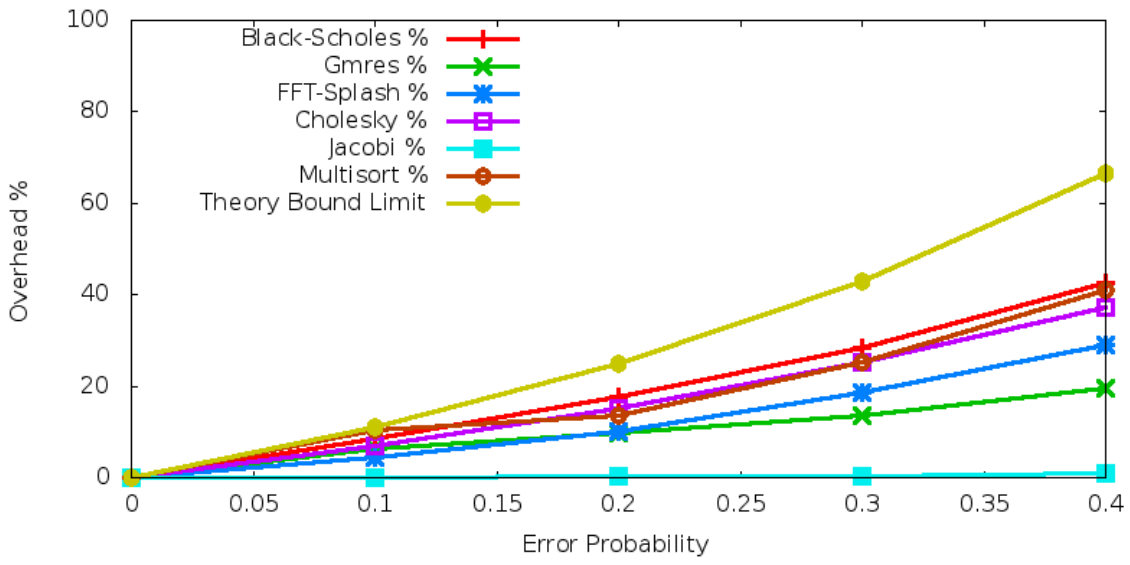


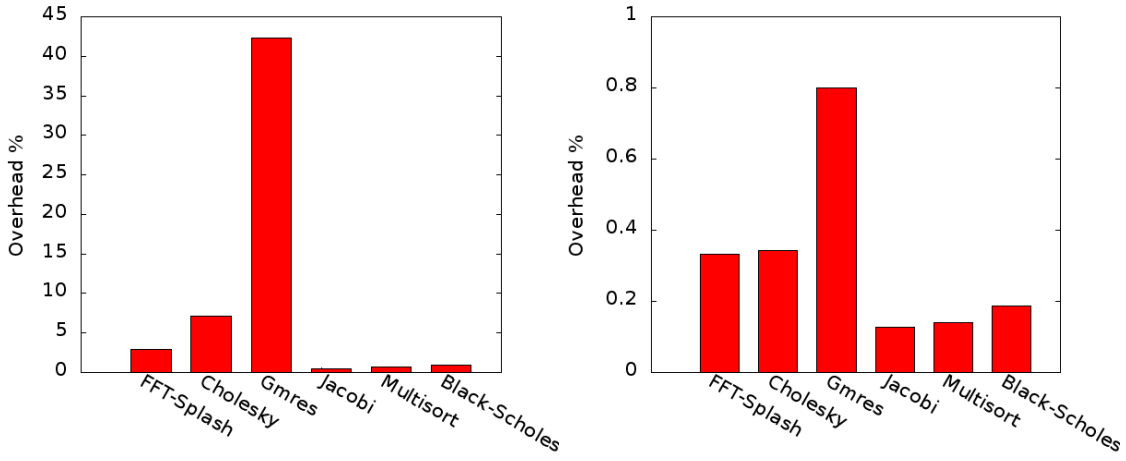
Figure 4.3: Time overheads due to retrying failed tasks: comparison of total execution time overhead for various probabilities of failure with total execution time of checkpointing without failure.

a probability of 0.1 for transient errors means that 10% of all tasks will fail and rerun, whereupon 10% of the reruns will also fail, etc.

In theory, a probability of $0 \leq x \leq 1$ of failure and rerun will execute a total number of tasks equal to

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

meaning that at probability of failure 0.1, 0.2, 0.3 and 0.4 we expect to perform respectively 11.11%, 25%, 42.86% and 66.67% more work due to retrying failed tasks. To test this hypothesis, we use a different configuration comparing the total execution time for these probabilities of failure, with the execution time using zero probability, to avoid counting the cost of checkpointing together with the cost of retrying failed tasks. Figure 4.3 presents the results, along with a curve showing the theoretically expected overhead just by rerunning the failed tasks. Note that in all benchmarks the cost is considerably lower than expected. We believe this to be because each worker core that detects a task failure immediately repeats the task execution, thereby taking advantage of warmed caches and locality. Al-



(a) Total overhead over BDDT

(b) Overhead of a permanent fault

Figure 4.4: Permanent faults

most consistently, the second time a worker core executes a task it is faster, as tasks are sufficiently small to fit in the processor cache.

We measure the overhead of permanent faults on total execution time similarly. As a baseline, we measure the total execution time of each benchmark using (vanilla) BDDT with three worker threads. We compare this to a RelyBDT execution with four worker threads, in which the first task always causes a permanent fault, causing one of the worker threads to stop. We set the probability of transient faults to zero. Note that even at a zero probability of transient faults, RelyBDT still entails the overhead of creating checkpoints. Figure 4.4a presents the overhead in total execution time. This includes the cost of checkpointing plus the cost of recovery from exactly one permanent fault. Note that this overhead is not directly comparable with the overhead of checkpointing reported in Figure 4.2 as it refers to executions with three worker threads, whereas Figure 4.2 refers to four worker threads.

To estimate the actual cost of a permanent fault, we perform another set of measurements. We measure the total running time for each benchmark executed using RelyBDT with three worker threads and zero probability of either transient or permanent faults; this baseline includes only the application time plus the overhead of creating checkpoints. We

Benchmark	RelyBDT 3 Workers	RelyBDT 1 Permanent	Recovery Time Overhead
Black-Scholes	605.27	606.4	1.12
Cholesky	980.56	983.77	3.21
FFT	669.2	671.41	2.21
GMRES	261.07	263.16	2.08
Jacobi	2492.87	2496.04	3.16
Multisort	162.24	163.08	0.84

Table 4.1: Permanent fault recovery time overhead in ms. Comparisson of RelyBDT with three starting workers vs RelyBDT with four starting workers and one permanent fault in the first task.

compare this against the total execution time using RelyBDT with four starting worker threads that immediately become three due to a permanent fault at the first task. Figure 4.4b shows that the overhead of recovering from one permanent fault is negligible, at worst 0.8% of execution time. In addition, to further demonstrate the performance overhead of permanent faults, Table 4.1 shows the timings of the above configuration and the required permanent fault recovery time, which on average is 2.1 ms. The recovery time can be dwarfed in comparison to the total execution time, hence we used the same configuration of the benchmarks as described before to minimize this effect. Again, GMRES has a higher percentage overhead mainly because it creates a lot of small tasks, meaning that recovering from a permanent fault introduces more task-steals from the faulty core to the remaining workers.

4.4.2 Space Overhead

Task-based checkpointing incurs little space overhead overall: only one task can run per worker at any given time and task semantics allow the system to only maintain checkpoints for the arguments of running tasks. We measure the space overhead of checkpoints for each application per worker, that is, the maximum space used by any given worker at any given time to store a checkpoint. Figure 4.5 shows the maximum space required to checkpoint a task in each benchmark. Note that although GMRES creates many tasks, each

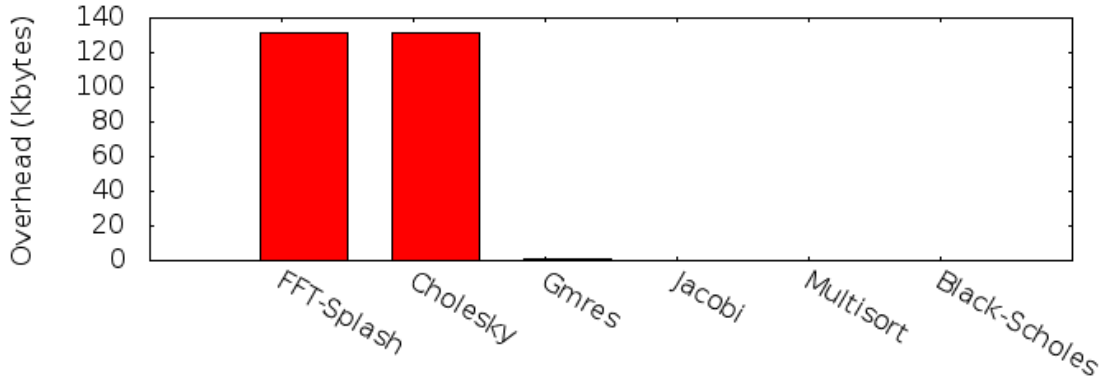


Figure 4.5: Worst case space overhead per application for each worker.

Benchmark	Tasks	`malloc()`		`memcpy()`	KBytes Restored / Probability			
		Calls	KBytes	Calls	10%	20%	30%	40%
Black-Scholes	100,000	0	0	0	0	0	0	0
Cholesky	5,984	4,459	566,467	570,726	65,851	142,475	223,085	339,706
FFT	28,864	46,679	1,104,229	1,397,305	123,529	268,415	45,381	674,288
GMRES	249,724	39,442	39,372	39,442	4,444	9,817	16,539	24,842
Jacobi	94,080	0	0	0	0	0	0	0
Multisort	836	0	0	0	0	0	0	0

Table 4.2: Space overhead and checkpoint memory operations.

task has a very small memory footprint (1 KByte), resulting in minimal space overhead for checkpointing all running tasks at any given time. On the other hand, FFT and Cholesky spawn tasks that operate on large tiles of large arrays, requiring a lot of space to store their checkpoints.

Table 4.2 shows the memory operations performed by checkpointing for each application. The second column shows the number of tasks per application. The third and fourth columns show the number of calls to `malloc()` and total size of allocated memory that stores checkpoints. The fifth column shows the number of `memcpy` calls needed to create the checkpoints. Note that this may be higher than the number of allocated checkpoints, due to strided arguments like array tiles that require multiple copies per task argument. The last four columns show the amount of memory that had to be copied back before rerunning a failed task, for four probabilities of fault.

Benchmark	Runtime Space	Data	Checkpoints	Overhead%	Total
Black-Scholes	404,111,456	358,400,000	0	0.00%	762,511,456
Cholesky	404,111,456	134,217,728	+524,288	0.39%	538,460,256
FFT	404,111,456	805,437,512	+524,288	0.07%	1,209,680,040
GMRES	404,111,456	73,453,000	+4,096	0.01%	477,565,480
Jacobi	404,111,456	881,852,416	0	0.00%	1,285,963,872
Multisort	404,111,456	167,772,160	0	0.00%	571,883,616

Table 4.3: Memory footprint (bytes) and overhead per application. The runtime space is always the same because the runtime reserves a fixed amount of space, regardless of the application.

Table 4.3 shows the memory footprint of the application and runtime, as well as the overheads due to keeping checkpoints. In cases where the memory footprint changes throughout execution, we present the high watermark: the maximum space required at any given point during execution. The first column shows the benchmark name. The second column shows the amount of memory reserved by the runtime system to contain task descriptors, the dependency graph, etc. The runtime takes a fixed amount of space, regardless of the application requirements. This is because it reserves a predefined maximum amount of memory that depends on the available hardware resources. The third column shows the amount of memory allocated by the application to store data processed by its tasks and the fourth column shows the maximum amount of memory used to store checkpoints at any given point during the execution. This amount is sensitive to the number of worker threads used. The number shown corresponds to four worker threads, meaning that, for instance, each worker thread in FFT needs 131072 bytes of memory to checkpoint the arguments of the task with the largest footprint. The fifth column presents the required checkpoint state as an overhead over the application data size. For all benchmarks it is much less than 1%. The last column shows the total maximum space required for checkpoints and corresponds to instances of the largest task running on all worker threads.

Chapter 5

ASGUARDIAN

5.1 Outline

This chapter presents ASGUARDIAN, a lightweight hardware mechanism that is able to reliably manage the memory transfer needs of the runtime system. The rest of this Chapter is organized as follows. Section 5.2 presents the architecture of the proposed mechanism and Section 5.3 covers the evaluation of our system.

Table 5.1: Task Structure

Task Descriptor	Functionality
ID	Identifies the current task
Input A	Pointer to the starting address of argument A
Input B	Pointer to the starting address of argument B
Output C	Pointer to the starting address of argument C
Size	Arguments size

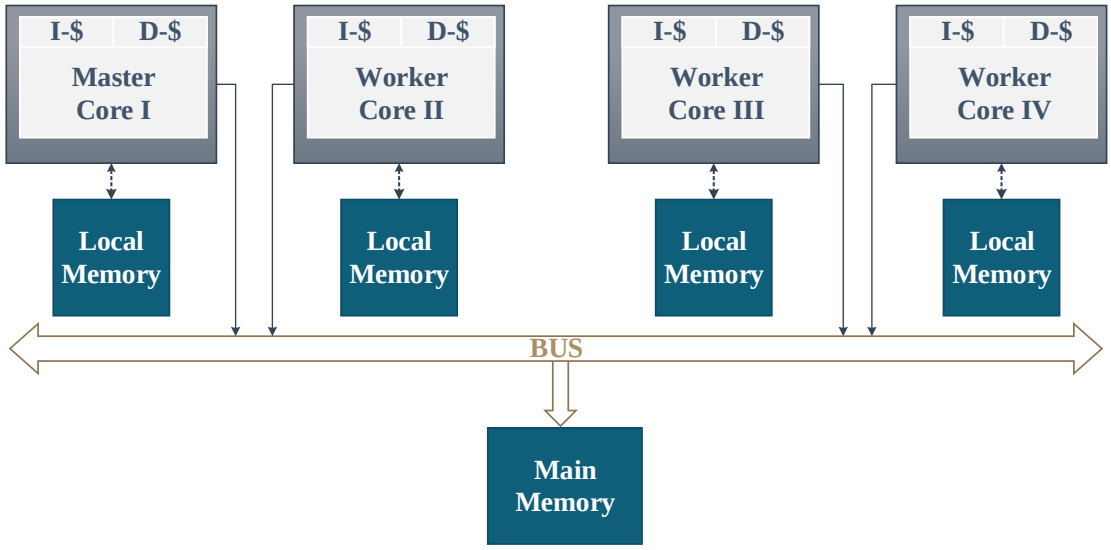


Figure 5.1: System Global Access/Memcpy: In this setup the the ASGUARDIAN is not included in the system. In the Global Access case the worker core accesses the main memory directly through the available interconnection module. In the Memcpy case each worker transfers all task descriptors and arguments to its local memory (hence keeps a task copy), executes all assigned tasks, and then commits back all results to the main memory.

5.2 System architecture

Fault Model: A basic multi-core setup is presented in Figure 5.1. Each worker accesses the main memory directly, in order to read all task descriptors and arguments (and store them to its local memory if available), and as soon as processing is done, write back all task outputs. However, an important issue that arises is that an unreliable worker core may corrupt vital memory contents, hence even cause an entire system breakdown.

Nowadays, there are many chip-hardening techniques [19] that can selectively protect vital hardware components (instead of the entire chip), thus leading to lower production costs. Hence, in the context of our work *we take as granted any mechanisms to detect transient and permanent faults*; we assume that each worker has integrated reliable modules that are capable to detect such errors. As we also describe in the next sections, when a permanent error occurs, the ASGUARDIAN tries to recover the system. When a transient

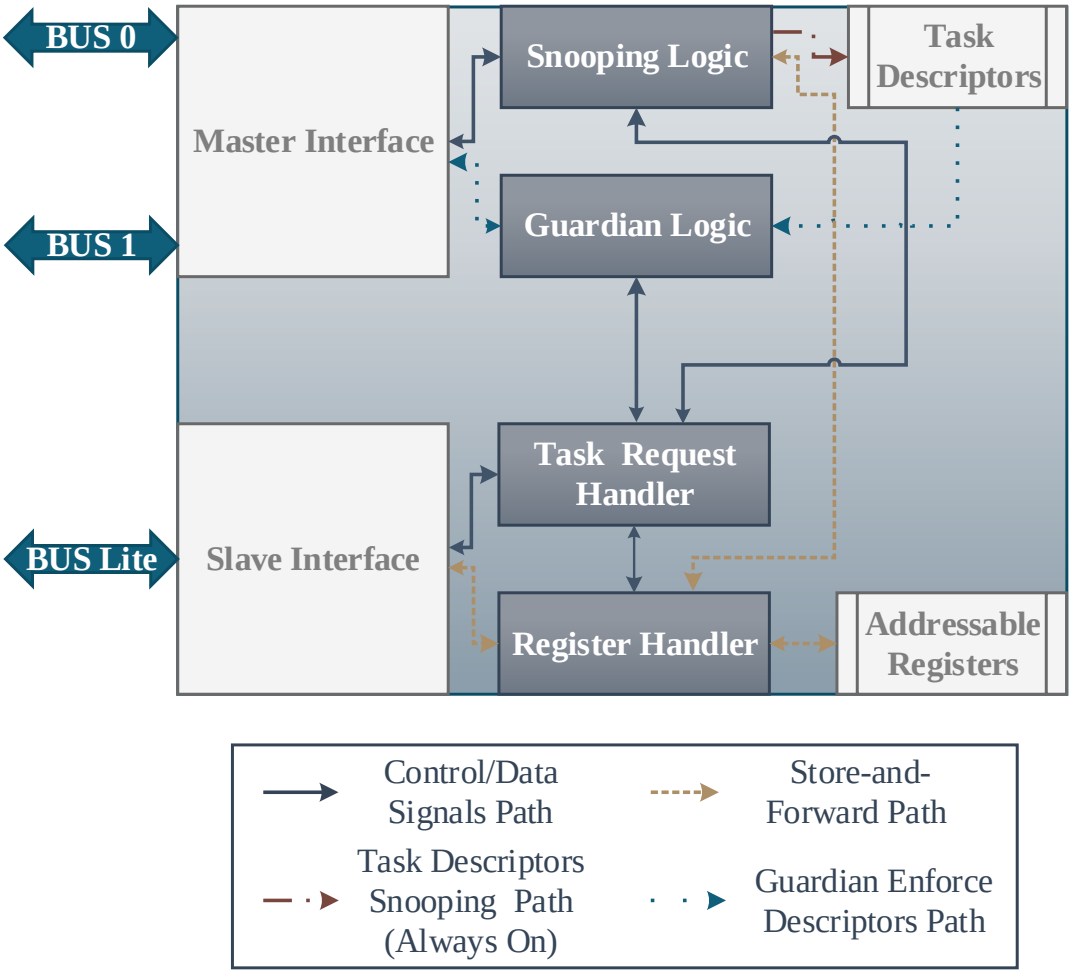


Figure 5.2: ASGUARDIAN Microarchitecture

worker fault is detected, it protects data corruption by utilizing memory address isolation (based on the task descriptors and TMR modules), and also migrates all pending tasks from the faulty worker to others.

The ASGUARDIAN overview: In our work we focus on MPSoCs that utilize a master processor interconnected with a set of worker cores, and a shared memory space. The main purpose of the ASGUARDIAN is to support reliable memory transfers of task descriptors and arguments. As an example, Table 5.1 shows the descriptor of a task with two input and one output arguments, which all have the same size. Two modes of operation are available, namely cut-through and store-and-forward. In both modes the transfers are split into two

steps, (a) read the task descriptors and (b) fetch all task input data from the memory to the workers, and once processing is done, store all task outputs back.

During the first step, the ASGUARDIAN keeps a copy of the descriptors, in order to initiate the transfers of all task arguments. During the application execution, when a new task is identified, the runtime notifies the ASGUARDIAN. During the second step, in the cut-through mode (illustrated in Figure 5.3) the ASGUARDIAN transfers all task data from the main memory to the worker local memory in sort bursts, while it also snoops necessary task metadata and stores them locally. However, in a store-and-forward configuration (shown in Figure 5.4), the ASGUARDIAN does not have access to the local memories of the processors. Hence, it first stores all task inputs data from the main memory to addressable registers, which are accessible by the worker cores. In case the task inputs size is large, all addressable registers can be switched to normal on-chip BRAM blocks.

Figure 5.2 presents an overview of the ASGUARDIAN microarchitecture. The basic components are (1) the Master and Worker interfaces, (2) the Snooping and Guardian logic and the (3) the Task Request and Register Handlers. The task descriptors and addressable registers are simple memory units. In the next paragraphs, we discuss these components in more detail.

Master and Worker Interfaces: Each worker communicates with the ASGUARDIAN by sending requests to the slave interface. In order to minimize the communication overhead between the two units, and most importantly to ensure the protection of the system, any interactions of the unreliable worker core with the runtime system had to be reduced to a bare minimum. The available API requests (implemented in C) are presented in Table 5.2. The worker interface is responsible for identifying an incoming request and pass it along to the appropriate Task Request or Register handler (described in the next paragraph). The runtime issues requests to the worker cores via a dedicated pattern register. All requests can be grouped in two major categories, namely task requests and register requests. The worker interface detects detects the category of the request and notifies the Task Request

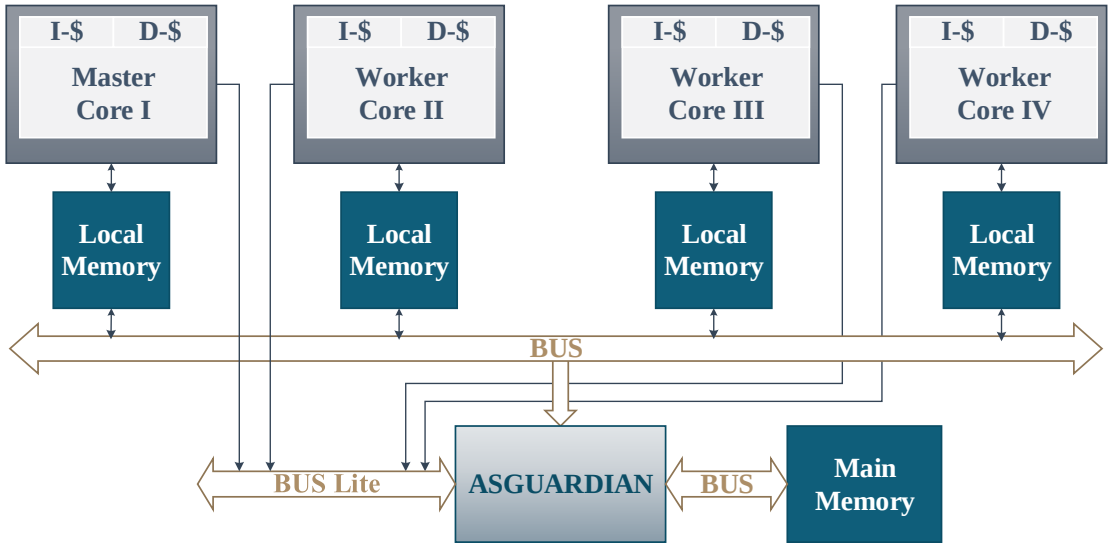


Figure 5.3: System cut-through: In this setup the ASGUARDIAN communicates with the cores through ”BUS Lite” and is connected to the local memories of the cores through ”BUS”. Memory transfers from/to the external memory to/from the local memories are performed in a cut-through fashion.

handler for task requests and the Register handler for register read/write requests. The worker communication interface between the worker cores and the ASGUARDIAN is very simple, hence does not impose any overheads.

The master interface provides the memory transfer operations of the ASGUARDIAN. Towards supporting both cut-through and store-and-forward modes, this interface is able to transfer both task descriptors/data between memories and between registers and a memory location. When the cut-through mode is enabled, data are transferred in bursts between the local memory of a worker and main memory. In the store-and-forward mode, the master interface transfers data between the local registers and the main memory. The address boundaries of the master interface are verified by the Guardian Logic (described in the next paragraph), in order to successfully avoid erroneous memory accesses from unreliable workers. Finally, the master communicates with the Snooping Logic and the Register Handler (described in the next paragraph), so the ASGUARDIAN can store all incoming data

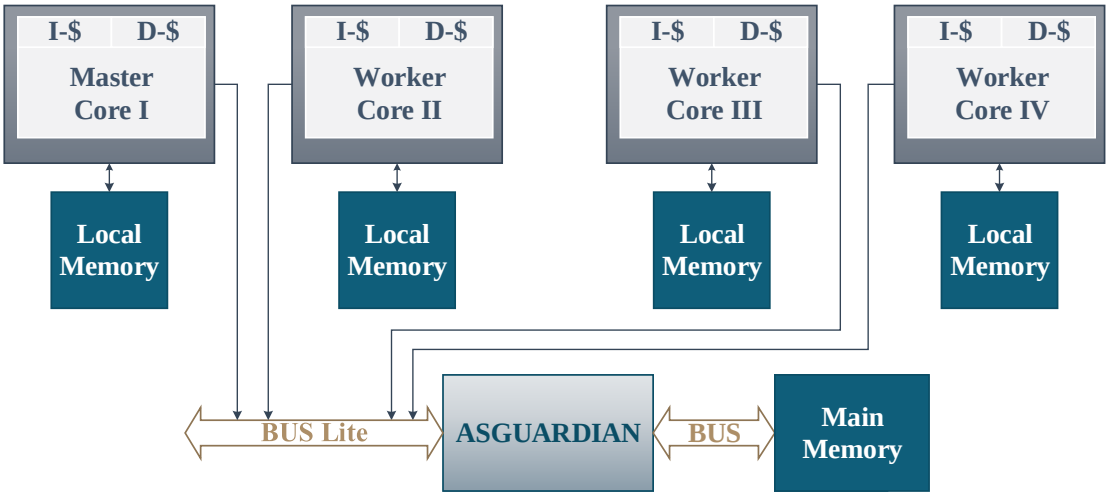


Figure 5.4: System store-and-forward: In this setup the ASGUARDIAN communicates with the cores only through "BUS Lite" and the ASGUARDIAN does not have direct access to worker local memories. Memory transfers from/to the main memory to/from the local memories are performed in a store-and-forward fashion.

from the main memory to the registers. During a data transfer in the cut-through mode, the guardian logic provides both the source and destination addresses; in the store-and-forward mode the guardian provides the memory address to be accessed, while and the Register Handler designates the correct register. To enhance memory access time, we consider that memory transfers are committed on an interconnection module with burst mode support.

Snooping and Guarding Tasks: The Snooping Logic is a module specifically designed to allow the ASGUARDIAN to *steal* data from the Master Interface. In both store-and-forward and cut-through modes, the ASGUARDIAN keeps a copy of the task descriptors and stores it on the Task Descriptors unit. The dash-dot line in Figure 5.2 represents the path of the task descriptors, from the Snooping Logic to the Task Descriptors unit. The task descriptors are required from the ASGUARDIAN in order to transfer the required task arguments to the worker cores. During the runtime initialization phase, the master processor notifies the ASGUARDIAN of the tasks starting address, the number of arguments in the tasks and the size of each argument. As a result, the Snooping Logic is able to know when

to store the task descriptors in the requesting workers memory space in the Task Descriptors unit.

On the store-and-forward mode the Snooping Logic is also responsible for forwarding incoming data to the Register Handler, which stores them in the Addressable Registers. As soon as a task execution is complete, the Snooping Logic forwards all task outputs to the Master Interface. The dashed line in Figure 5.2 represents the path of the task descriptors and arguments from/to the Snooping Logic to/from the Addressable Registers and to/from the Slave Interface. The main insight of the Snooping Logic is that by tracking the traffic on the Master Interface is able to identify incoming tasks descriptors or task data, which provide to the ASGUARDIAN the required task information to complete memory transfers reliably.

The Guardian Logic is a simple but vital unit of the ASGUARDIAN; its main purpose is to enforce the address space isolation of each task by checking the accessed memory addresses in each step.The doted line in Figure 5.2 highlights the path of the required task descriptors from the Task Descriptors Unit to the Guardian Logic and to the Master Interface. By using the information already stored in the Task Descriptors, the Guardian Logic provides the allowed address space to be accessed by the master interface during each memory transfer. In addition to the Task Descriptors, the Guardian Logic offers fault-coverage on operations by calculating the addressing of memories in a TMR fashion [31, 32, 1, 26].

Task Request and Register Handlers: The Task Request Handler is responsible for identifying all task requests. When a task request arrives, it notifies the Guardian and/or Snooping Logic units that a task transfer operation is on the fly, hence ensuring that memory accesses will be valid. The Snooping Logic is not required on the Cut-Through mode when the task arguments are transferred back to the main memory. In addition the unit passes information of the state of the ASGUARDIAN to the worker interface through the addressable registers, in order to acknowledge the worker cores that data transfers are com-

plete. The lines in Figure 5.2 highlight the control and data signals between the Handlers, the Interfaces and the Logic Units. Figure 5.6 gives an example of the task request control flow. In order to keep the guardian as lightweight as possible and the processor requirements to a bare minimum, the completion of events is performed in a busy-wait fashion rather than supporting interrupts. Consequently, the Task Request Handler needs to only write a specified register through the Register Handler, which is continuously checked by the master processor, in order to know the state of the current operation.

The Register Handler is responsible for providing access to the addressable registers. It can perform multiple writes in a single cycle, hence accommodate possible writes from different units, while also perform serial access to the registers, in order to support the store-and-forward ASGUARDIAN mode. For example, the Register Handler is able to provide a series of registers to the master interface to write incoming data of one task argument. At the same time, it can also give access to a set of registers to the worker interface, hence the worker can directly read another task argument.

5.2.1 Stand-alone Guardian Logic

An extended version of the Guardian Logic is able to provide a stand-alone protection mechanism for task-based runtime systems because of the ability of the Guardian logic to utilize task descriptors for fault tolerance purposes. Similar protection mechanisms exist in modern processors, like the paged virtual memory. Figure 5.5 highlights a simple paged virtual memory architecture where a page table maps virtual address to physical ones.

Paged virtual memory restricts an application to access only virtual pages that have been explicitly allocated to it. In a similar way, the main insight behind a stand-alone version of the Guardian Logic is that this module would verify that each worker core is allowed to access a specified memory location by comparing the requested address and the addressing provided from the current task descriptor. The Guardian Logic would not allow worker cores to access memory locations outside the current task descriptor hence, isolating the

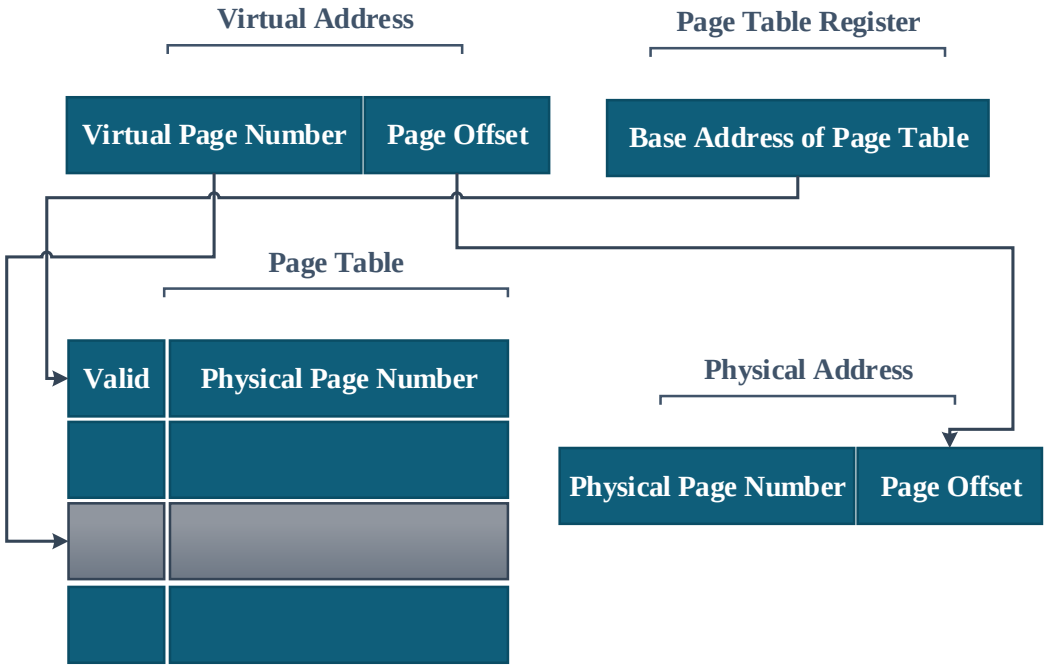


Figure 5.5: Virtual Memory Architecture: When a virtual address arives, the page table is used to map the virtual address to a physical address.

worker inside the allowed address space. The main benefit behind the Guardian Logic is that it requires minimum hardware resources and it does not require OS support because of the characteristcs of the task based programming model; making it an ideal stand-alone memory protection mechanism for an embedded task-based multicore enviroment.

5.3 Experimental Results

5.3.1 Runtime system

To evaluate the ASGUARDIAN module, we have developed a simple runtime system with basic task-distribution functionalities. In our current testing platform, we consider one master and three worker cores with shared (main) memory, however each worker has also its own local space. The master core initializes the system, creates tasks, initializes all runtime data structures and finally dispatches tasks to all workers. Each worker core is

Table 5.2: ASGUARDIAN API

Function	ASGUARDIAN Operation
Cut-Through Mode	
task_request()	Transfer the next task, descriptor and data, to a predetermined workers memory location
task_complete()	Transfer the output arguments from the workers memory, back to the main memory. This function is called after the calculation of the output arguments.
Store-and-Forward Mode	
task_request_des()	Store the next task descriptor in the ASGUARDIAN and make it addressable
task_request_data()	Store the task arguments in the ASGUARDIAN and make it addressable
task_complete_ready()	Transfer the output arguments from the ASGUARDIAN, back to the main memory. This function is called after the worker writes the output arguments on the addressable registers.
Both Modes	
read_reg(uint16 sel)	Read the data of the selected register
write_reg(uint16 sel, uint32 val)	Write the value on the selected register

responsible for a specified number of tasks which are identified from a unique ID. Without loss of generality, for demonstrative purposes we consider simple tasks with two input and one output arguments. As shown in Table 5.1, each task description is composed by five values; the task ID, the starting addresses of input arguments A and B, the starting address of output argument C, and finally the size of all arguments.

During the runtime initialization phase, the ASGUARDIAN receives all task descriptions, in order to know where the first task is and the memory patterns for accessing the tasks metadata and arguments. To investigate the ASGUARDIAN performance overhead, we implemented four different strategies when accessing the main memory to fetch task data, namely global-access, memcpy, cut-through and store-and-forward. The first two do not utilize any ASGUARDIAN protection mechanisms. In global-access, each worker accesses directly the main memory to find the task descriptors and execute the tasks, i.e. in other words, data are not copied to scratchpad memories. The memcpy version first creates a copy of the task descriptors and data to the local worker memory of and then starts the task execution. Once processing is done, the worker transfers back the results

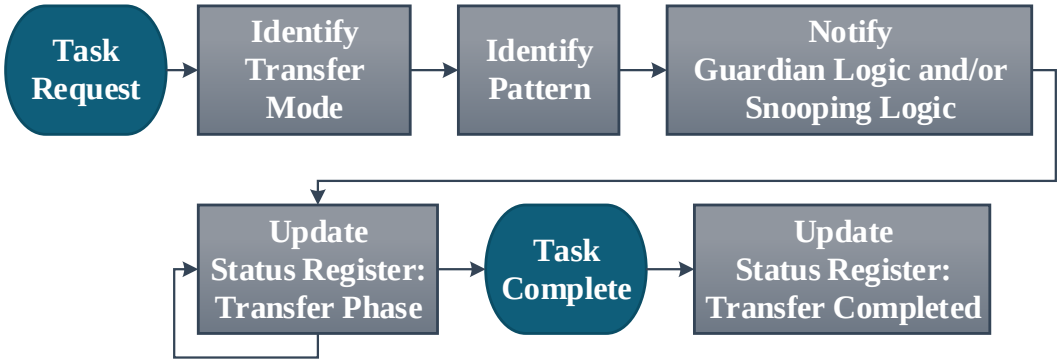


Figure 5.6: Task Request Handler Flowchart: The steps of the control logic behind the Task Request Handler is defined in 7 steps. The first steps require the identification of a new request, the current memory transfer mode and the request type by pattern recognition. The second part is the notification of the required units, based on the request and the update of the status register. The latter is continuously checked by the worker core, in order to know the transfer state.

to the main memory. As it can be observed, these two approaches are unreliable, since anytime a worker may corrupt the main memory contents. The last two strategies employ the ASGUARDIAN modes described in Section 5.2.

Table 5.2 presents the basic API of the ASGUARDIAN utilized by the runtime. In the cut-through mode, a worker requests a task by calling the task_request() function, which writes a specified pattern to the request register, and waits until the ASGUARDIAN is done by updating its status register. When all data transfers are complete, the worker cores reads the task descriptors from a predetermined location in his local memory and starts the task execution. As soon as the task processing is done, the worker calls the task_complete() function. This procedure is repeated by all workers until all tasks are executed.

In the store-and-forward mode, the worker core first calls the task_request_des(), which asks the ASGUARDIAN to begin transferring the task descriptors and task arguments. Once all task data are transferred, they are addressable to the worker core. After the busy-wait loop, the worker reads the task descriptors by calling the read_reg(uint16 select) function which reads the data from the specified register of the ASGUARDIAN. As

it was mentioned, in our current configuration, each task requires five descriptors, which are stored in the first five addressable registers. When this step is completed the worker calls the task_request_data(), which enables the ASGUARDIAN to re-use the addressable registers holding the task descriptors for task arguments if needed. Finally the worker reads all task data from the registers in the same fashion as before. The number of registers to be read is defined from the size of the task. In order to reduce the latency of the store-and-forward method, registers reading is overlapped with the data transfer from the main memory. After the task processing is complete the worker writes the results back to the addressable registers and notifies the ASGUARDIAN by calling the task_complete_ready() function.

5.3.2 ASGUARDIAN evaluation

We performed various experiments, in order to evaluate the ASGUARIDAN overhead in terms of performance and hardware resources utilization. Moreover, we implemented a hardware prototype of the ASGUARDIAN on a Xilinx ML605 FGA board using the ARM AMBA-based AXI and AXI-Lite protocols. In our experiments we used the runtime system described in the beginning of this section, and mapped it on a testing platform with one master and three Micrboblazes as workers. The Microblazes instruction and data caches were set at 8KB and the local memories at 32K. The main memory of the system was set at 64MB, and the operating frequency was 100MHz. Finally the operating frequency of the ASGUARDIAN was set at 150MHz in order for the module to be ahead of requests and to be able to operate in a TMR fashion in the cases described in 5.2.

Hardware Resources: Regarding the hardware resources utilization, we used the Xilinx Design Suite 14.7 to get the total system resources after placing and routing the design to the FPGA, which are shown in Table 5.3. Apart from the four Microblaze cores and the ASGUARDIAN module, the system also contains other necessary units like the SDRAM memory and its controller, hardware timers, an RS232 module, bus interconnects and other

Table 5.3: FPGA Resources Utilization

Unit	Flip Flops	LUTs	BRAMs
System	17947	23259	70
ASGUARDIAN	1870	2764	1
Reference Microblaze	2632	2843	14
Reference DMA engine	1177	1192	1

peripherals. Hence, according to Table 5.3, the ASGUARDIAN requires only 10.4% of the total FFs, 11.8% of the total LUTs and 1.4% of the BRAMs, yielding an average of 8% of the overall system resources. Moreover, the ASGUARDIAN utilizes 58.4% on average of the resources when compared to a typical Microblaze configuration. Finally, we also provide the resources required from a very simple DMA engine, in order to define a baseline for supporting burst memory transfers. As we can observe, the ASGUARDIAN yields an overhead of 47% of the resources required from the DMA engine, at the expense of offering task-management capabilities and fault-tolerance.

Performance: The performance evaluation of the ASGUARDIAN was based on the four previously described memory access strategies (global-access, memcpy, cut-through, store-and-forward). In order to measure the performance of each one under various conditions, we created simple tasks, which add two arrays under a varying number of input sizes, ranging from 300 to 12000 elements. Each task performs addition on 20 elements, hence the total number of tasks ranges from 30 to 1200 tasks, while each of the three workers is responsible for a workload of 10 to 400 tasks. Table 5.4 and Figure 5.7 present the timings of one worker, while Table 5.5 and Figure 5.8 show the total runtime timings.

As we can observe from the results, the store-and-forward ASGUARDIAN configuration yields an execution time slowdown of 1.7x and 2.3x on average, when compared to an unprotected scratchpad-based and global-access runtime system respectively. On the other hand, the cut-through mode shows a minimum slowdown of 1.2x on average in comparison to the unprotected global-access system, and a speedup of 6% on average when compared to the unprotected memcpy-based runtime, while providing sufficient fault-tolerance to the

Table 5.4: Worker Timings (msec)

Input Size - Elements	Store-and-Forward	Cut-through	Memcy	Global Access
600	0.424	0.159	0.209	0.077
1200	0.818	0.298	0.327	0.152
1800	1.212	0.436	0.482	0.221
6000	3.971	1.406	1.578	0.702
12000	7.913	2.792	3.147	1.395
18000	11.855	4.178	4.712	2.101
24000	15.797	5.564	6.296	2.806

Table 5.5: Total Timings (msec)

Input Size - Elements	Store-and-Forward	Cut-through	Memcy	Global Access
600	0.597	0.332	0.383	0.250
1200	1.157	0.637	0.666	0.492
1800	1.718	0.942	0.989	0.727
6000	5.641	3.076	3.248	2.371
12000	11.245	6.124	6.478	4.727
18000	16.849	9.172	9.706	7.095
24000	22.453	12.220	12.951	9.462

tasks management and graceful degradation of the system in case of permanent errors. The slowdown of the system in the store-and-forward mode comes from the fact that the task descriptors and arguments have to transfered from the main memory to the worker cores in two steps. On the other hand, the speedup of the Cut-Through mode when compared to the memcpy version of the system comes from the fact that memory transfer operations of the ASGUARDIAN utilized dedicated hardware with burst support to perform the transfers without the execution of instructions that the worker cores has to rely on for the transfer.

Permanent Faults Recovery: While the ASGUARDIAN ensures secured memory transfers by the Guardian Logic unit, which utilizes Finite State Machines in a TMR mode, we have integrated extra mechanisms, so the system can recover from detected permanent core faults. We have considered the following two scenarios where permanent faults can occur, before a task request and during a task request. In the first case, the worker core fails before requesting a new task, hence independently of the ASGUARDIAN transfer mode (cut-through or store-and-forward), the system continues uninterrupted because the

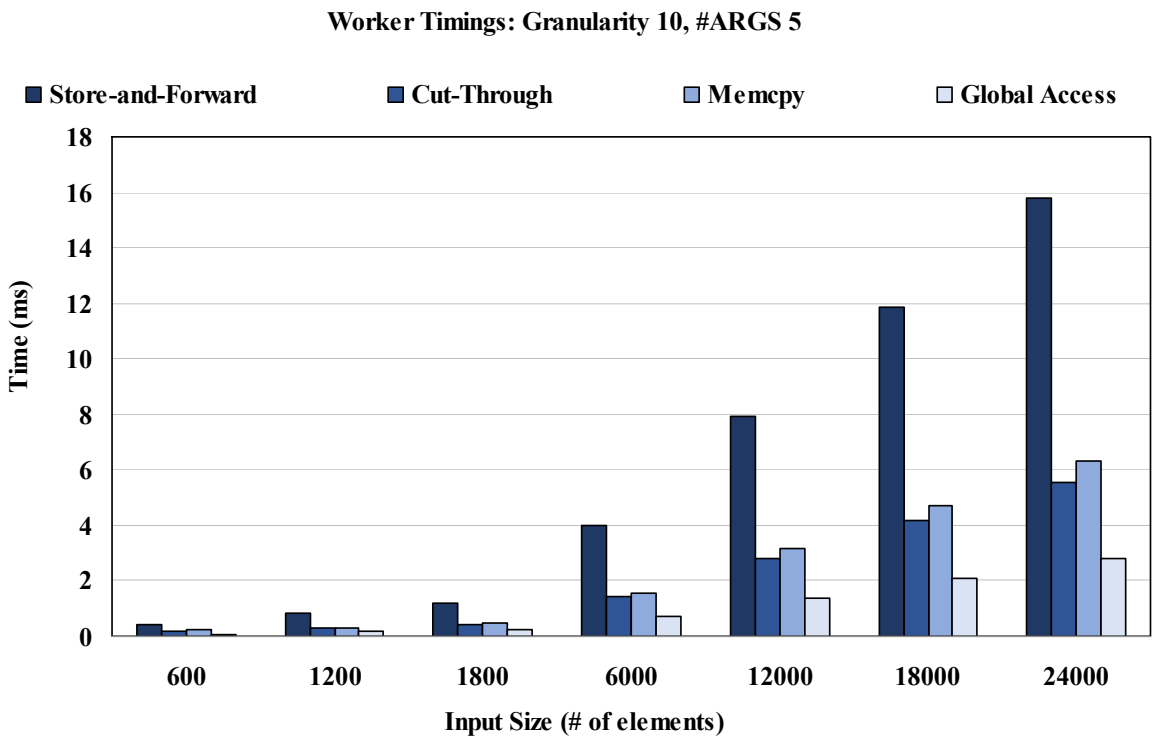


Figure 5.7: Worker Timings: Worker timings are presented here for different workload input sizes. The workload input size ranges from 600 to 24000 elements. Each bar represents a different memory transfer method, with the store-and-forward and cut-through being the reliable ones, and the memcpy and global access the unreliable ones.

Guardian Logic will continue servicing task requests to all remaining cores until all tasks are executed.

If an error is detected during a task request, the ASGUARDIAN tries to recover the system based on its current configuration. In the cut-through mode, the ASGUARDIAN transfers both task descriptors and arguments to the local memory of the worker core and then continues servicing requests. If a worker is detected as faulty, the ASGUARDIAN, may change the ownership of its task descriptors to another one, using a specific pattern written to the fault register. During this operation, the takeover worker will get the remaining task of the failed one, while the ASGUARDIAN will seamlessly re-execute the failed task transfer to to the local memory of the takeover worker.

In case of failure after a task-complete request, the ASGUARDIAN operates normally, because the unit is able to operate autonomously without further communications with the

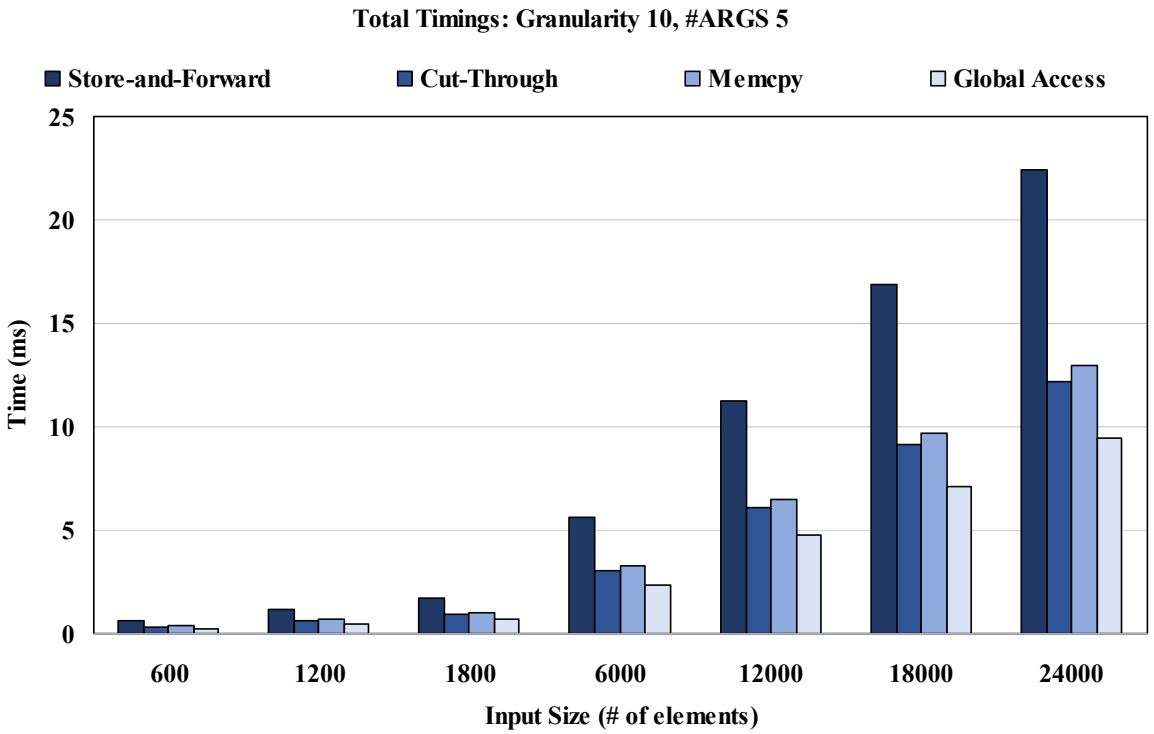


Figure 5.8: Total Timings: Total timings are presented here for different workload input sizes. The master timings is only dependant to the workload input size, hence the total timings are similar to the worker ones. The workload input size ranges from 600 to 24000 elements. Each bar represents a different memory transfer method; the store-and-forward and cut-through are the reliable ones, and the memcpy and global access the unreliable ones.

failed core and continues from this point serving the rest of the workers. The scheduling policy of tasks to available worker cores can be defined in the runtime, hence the AS-GUARDIAN can support different scheduling policies.

In the store-and-forward configuration, when a worker fails after a task descriptor or a task arguments request, the ASGUARDIAN will react in the same way as in the cut-through mode where the worker fails after a task request. More specifically, the ASGUARDIAN will transfer the task descriptors, make them addressable to the workers and then proceed to the task input data transfers. After each transfer, the ASGUARDIAN writes a specified pattern to the status register to signal a status update, which is acknowledged by the worker core. Once all data are transferred, the ASGUARDIAN waits either for confirmation that the worker has successfully read all arguments, or a notification from a possibly takeover

worker that the original worker failed. In the second case, the next task-descriptor request from a *new* worker will directly receive the task descriptors and the arguments without extra transfers. In other words, permanent faults do not require task description transfers to be re-executed.

The ASGUARDIAN blocking mode serves two main purposes; to make sure that all workers have successfully received their tasks' arguments, and because it serves as a indication to the worker cores that another worker may have failed. The detection is possible because the status of the ASGUARDIAN will remain busy when a worker tries to communicate with the ASGUARDIAN. In case a worker fault is detected after a notification has been sent to ASGUARDIAN, all task descriptors of the faulty worker will become addressable to the one that will resume its pending tasks. However, all pending task input data need to be loaded again, since the ASGUARDIAN will overwrite the original registers with new task data due to the "fake" received notification.

In addition, if a fault is detected after a task-complete request, the ASGUARDIAN operates as in the cut-through configuration. Figure 5.9 highlights a case were the first worker fails permanently at 50% of the workload and the remaining workers continue. The lines demonstrate the timings required for 3 workers. The results indicate that the ASGUARDIAN does not further impede the performance of the system in case of a permanent fault. This is also verified by Figure 5.9, where the extra time required from the remaining workers to complete all tasks, is almost the same as the time required from the system with three workers, plus the time required from the fault-free workers to compute the remaining 50% (25% each in the demonstrated case) of the faulty core workload.

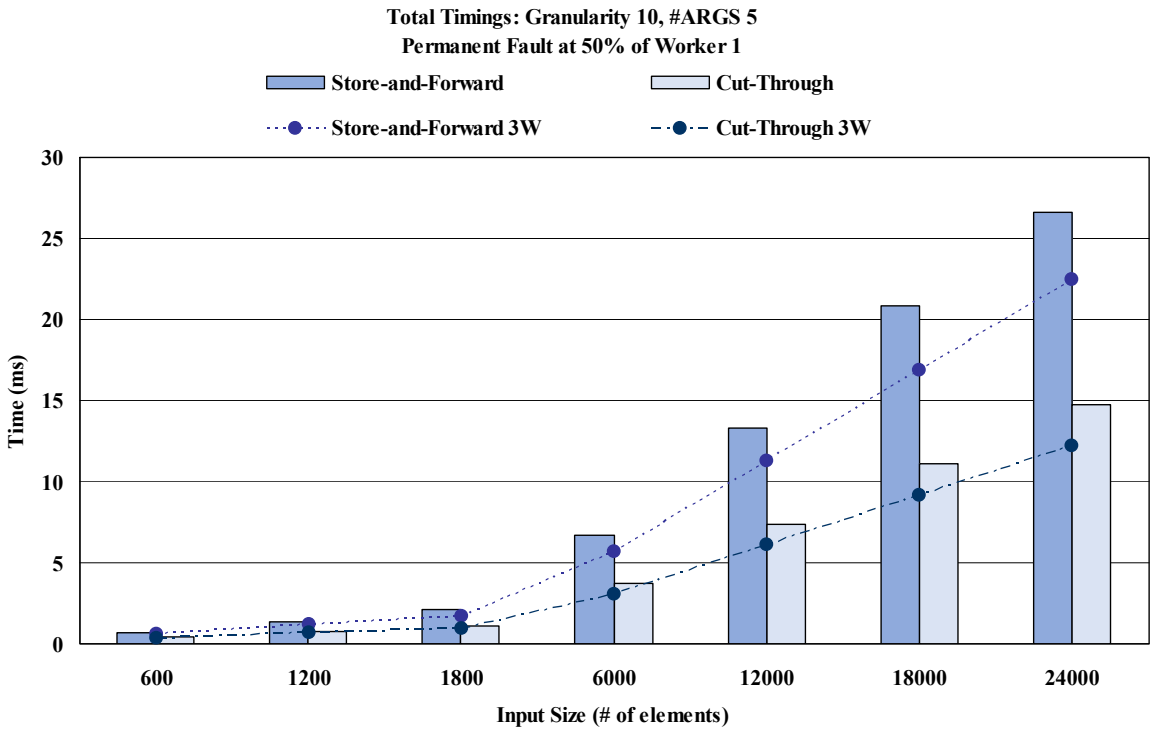


Figure 5.9: Total Timings with One Permanent Fault at 50%: The graph presents the total timing results when one worker core has a permanent fault at 50% of its workload. In addition, the lines present the total timings for three worker cores. The remaining 50% of the workload is distributed among the remaining workers evenly. The available memory transfer modes are store-and-forward and cut-through, since the unreliable versions would have failed.

Chapter 6

Validation and Features

6.1 Outline

This Chapter presents the features and, the verification and validation methods utilized to test and confirm the proper operation of both RelyBDT and the ASGUARDIAN. The rest of this Chapter is organizes as follows, Section 6.2 presents a general overview of the verification and validation process, Section 6.3 presents the Verification methods, Section 6.4 describes the Validation steps and finally Section 6.5 presents the implemented features of RelyBDT and the ASGUARDIAN.

6.2 V&V Overview

Modern systems have to pass multiple steps of Verification and Validation in order to confirm the quality of the product [20]. While the validation is as simple as confirming the required specifications of the system, verification is based upon verifying the design and the code. Both RelyBDT and the software part of the ASGUARDIAN passed through the same validation methods, extra steps had to be taken in order to verify the hardware part of the ASGUARDIAN.

6.3 Verification

Formal Verification of the system through formal methods of mathematics is a very difficult process especially for systems with great complexity like the ones presented in this thesis. As a result, as part of the Verification process of the RelyBDT and the ASGUARDIAN we used the features described in Section 6.5 as the formal specification of the system and verified the correctness of each function for a number of runs. In addition, for the hardware part of the ASGUARDIAN we verified the logic and FSM modules of the ASGUARDIAN with extended simulations.

6.4 Validation

The first step taken towards confirming the correctness of both RelyBDT and the ASGUARDIAN was to validate the results of our implementations. In RelyBDT because of the x86 architecture the validation was easier and was based on comparing the output dataset of the RelyBDT against the original results of the runtime without any modifications. In addition by using basic C functions like memcmp and assert [44] we were able to verify the correctness of our design during different steps of the runtime.

The FPGA implementation of the ASGUARDIAN required a number of extra steps in the validation process because of the time required to access the on and off chip memories and transfer their data. In addition to basic C functions and their Microblaze equivalents, we used the Xilinx Design suite Tools [13, 15, 14] in order to verify the output datasets by accessing the DDR memory and the BRAMs on the FPGA and comparing them to the original outputs. The memory accessing process takes place at different steps of the runtime after certain operations are complete, like task transfers. In order to automate the process, a validation script was implemented that first executes the original runtime without the ASGUARDIAN on the four Microblaze cores. By the end of the execution the script access a number of on- and off-chip memories and performs a checksum function

on the contents. A version of the runtime with the ASGUARDIAN is then executed and a new checksum is generated. The two results are compared against each other to verify the correctness of the results. While the checksum function is not ideal and may prove error prone, it provides a lightweight solution to be used during the development stages. Memory dumping of the results from the off-chip memory was used in order to verify the correctness of the augmented system after the aggressive development phase of the ASGUARDIAN mechanism. The main reason behind using memory dumping at later stages of the development was because of the time required to transfer the whole data set from the off-chip memory of the FPGA to the workstation, especially for large datasets.

6.5 Features

The basic specifications of RelyBDT are the following:

- Workers threads are able to create checkpoints based upon the task descriptor.
- The creation of checkpoints take place after acquiring the task from the workers personal task-queue and
- before the execution of task function.
- Checkpoints are created only for inout arguments.
- Checkpoints are not created for input or output arguments.
- Inout arguments may be array- or stride-based.
- Faults both permanent and transient are simulated with a coin-toss like function based on a uniform distribution and a specified fault probability.
- If the probability is set at zero no faults occur. If the probability is set at one faults will occur all the time and the program will not terminate.

- Transient faults are only detected after the execution of a task and before the submission of the results.
- Permanent faults are detected after acquiring a task and before the submission of the results.
- Permanent faults disable the worker thread.
- In case of a permanent fault the rest of the workers finish the remaining tasks in the workers task-queue through task stealing.
- One worker and one master must be operational for the successful completion of the program.

The basic specifications of the ASGUARDIAN are the following:

The ASGUARDIAN:

- Transfers task descriptors and data from one memory location to an other memory location based on addressing provided from the core.
- Transfers task descriptors and data from one memory location to an other memory location based on addressing provided from worker dedicated registers.
- Transfers task descriptors and data from one memory location to local addressable registers based on addressing provided from the core.
- Transfers task descriptors and data from one memory location to local addressable registers based on addressing provided from worker dedicated registers.
- Transfers data from addressable registers to a memory location based on addressing provided from the core.
- Transfers data from addressable registers to a memory location based on addressing provided from worker dedicated registers.

- Supports burst-mode transfers.
- Allows cores to write on addressable registers.
- Allows cores to read addressable registers.
- Identifies requests based on pattern recognition on specified registers.
- Identifies the current phase of a transaction.
- Enfornces memory access addressing based on data stored on local registers.
- Calculates addressing locations with a starting address, a step, and a size.
- Stores task descriptors on worker dedicated registers.
- Transfer task descriptors between worker dedicated registers.

The ASGUARDIANs Runtime Master core:

- Initializes the runtime system.
- Initializes the task descriptors.
- Initializes the task data.
- Initializes mutexes.
- Initializes mailboxes.
- Initializes the ASGUARDIAN.
- Notifies the worker cores.

The ASGUARDIANs Runtime Worker core:

- Requests task descriptors from the ASGUARDIAN.
- Request task data from the ASGUARDIAN.

- Write on addressable registers of the ASGUARDIAN.
- Read from addressable registers of the ASGUARDIAN.
- Request the completion of a task from the ASGUARDIAN.
- Access the main memory for task descriptors and data.
- Copy from the main memory to the local memory task descriptors and data.
- Copy to the main memory from the local memory task descriptors and data.
- Execute the task function.

Chapter 7

Conclusion

In this thesis we have presented RelyBDT and the ASGUARDIAN, a software checkpointing mechanism for the fault-tolerant execution of programs and a leightweight hardware mechanism that provides fault-tolerance at hardware level for reliable task data transfers to all available worker cores in MPSoC. Our proposed checkpointing mechanism can transparently re-execute failed tasks and achieve correct execution of the application at runtime. We found that for our benchmarks, saving enough program state to re-execute the tasks requires almost negligible storage space, while the performance overhead depends strongly on the benchmark behavior. Regarding the ASGUARDIAN, to enchance its programmability and ease of use, we have developed a simple API that abstracts away from the user memory management and transfer needs of the task-based runtime system. To evaluate our proposal we implemented a hardware prototype on a Xilinx ML605 FPGA board and attached it to a MPSoC with 1 master Microblaze and 3 Microblazes as workers. Using a lightweight runtime system we performed experiments with a varying number of tasks assigned to workers. Results suggest that the ASGUARDIAN reliability features offer an additional level of protection on memory transfers at a minor cost in terms of resources and performance degradation.

Bibliography

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital systems testing and testable design*. Computer Science Press, 1990.

[2] Andrea Acquaviva, Andrea Alimonda, Salvatore Carta, and Michele Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. volume 2008, 2008.

[3] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. Architecture of the ibm system/360. In *IBM Journal of Research and Development 8*, 1964.

[4] O. Arnold and G. Fettweis. Resilient dynamic task scheduling for unreliable heterogeneous mpsocs. In *Semiconductor Conference Dresden (SCD)*, 2011.

[5] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: A feasibility study. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, March 2006.

[6] C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.

[8] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5, January 1998.

[9] Tamer Dallou and Ben Juurlink. Hardware-based task dependency resolution for the starss programming model. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 367–374, Washington, DC, USA, 2012. IEEE Computer Society.

[10] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[11] Onur Derin, Emanuele Cannella, Giuseppe Tuveri, Paolo Meloni, Todor Stefanov, Leandro Fiorin, Luigi Raffo, and Mariagiovanna Sami. A system-level approach to adaptivity and fault-tolerance in noc-based mpsocs: the madness project. *Microprocessors and Microsystems*, 37(6-7):515–529, August 2013.

[12] OpenMP Documentation. Openmp application program interface - version 4.0. 2013.

[13] Xilinx Documentation. Microblaze proccessor reference guide.

[14] Xilinx Documentation. Platform studio help.

[15] Xilinx Documentation. Xilinx software development kit help.

[16] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[17] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[18] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[19] Yu Fa-Xin, Liu Jia-Rui, Huang Zheng-Liang, Luo Hao, and Lu Zhe-Ming. Overview of radiation hardening techniques for ic design. In *Information Technology Journal*, pages 1068–1080, 2010.

[20] Richard E. Fairley. Software project management. In *Encyclopedia of Computer Science*, pages 1634–1636. John Wiley and Sons Ltd., Chichester, UK.

[21] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ASPLOS*, 2010.

[22] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[23] Chun-Hsian Huang, Pao-Ann Hsiung, and Jih-Sheng Shen. Model-based platform-specific co-design methodology for dynamically partially reconfigurable systems with hardware virtualization and preemption. volume 56, pages 545–560, New York, NY, USA, November 2010. Elsevier North-Holland, Inc.

[24] Daya Shanker Khudia, Griffin Wright, and Scott Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 99–108, New York, NY, USA, 2012. ACM.

[25] Man lap Li, Pradeep Ramach, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Swat: An error resilient system.

[26] R.E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962.

[27] Intel Software Developer's Manual. Intel itanium architecture.

[28] Y. Masubuchi, S. Hoshina, T. Shimada, B. Hirayama, and N. Kato. Fault recovery mechanism for multiprocessor servers. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 184–193, June 1997.

[29] Vassilis Papaefstathiou, Manolis Katevenis, Dimitrios S. Nikolopoulos, and Dionisios N. Pnevmatikatos. Prefetching and cache management using task lifetimes. In *ICS*, 2013.

[30] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, August 2009.

[31] Dhiraj K. Pradhan, editor. *Fault-tolerant Computing: Theory and Techniques; Vol. 1*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[32] Dhiraj K. Pradhan, editor. *Fault-tolerant Computing: Theory and Techniques; Vol. 2*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[33] Dhiraj K. Pradhan, editor. *Fault-tolerant Computer System Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[34] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 119–126, 2001.

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

[36] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *MICRO*, 2001.

[37] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *MICRO*, 2001.

[38] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *CGO*, 2005.

[39] The sequoia programming language. http://http://sequoia.stanford.edu.

[40] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 297–306, Washington, DC, USA, 2007. IEEE Computer Society.

[41] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 37–48, New York, NY, USA, 2009. ACM.

[42] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS*, 2009.

[43] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The impact of technology scaling on lifetime reliability. In *International Conference on Dependable Systems and Networks*, pages 177–186, July 2004.

[44] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2Nd Edition)*. Addison-Wesley Professional, 2005.

[45] Dimitris Theodoropoulos, Polyvios Pratikakis, and Dionisios N. Pnevmatikatos. Efficient runtime support for embedded mpsocs. In *ICSAMOS*, pages 164–171, 2013.

[46] Dimitris Theodoropoulos, Polyvios Pratikakis, and Dionisios N. Pnevmatikatos. Enhancing fault-tolerant run-time support for embedded mpsocs. In *MULTIPROG-2014*, 2014.

[47] Antonino Tumeo, Marco Branca, Lorenzo Camerini, Marco Ceriani, Gianluca Palermo, Fabrizio Ferrandi, Donatella Sciuto, and Matteo Monchiero. A dual-priority real-time multiprocessor system on fpga for automotive applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 1039–1044, New York, NY, USA, 2008. ACM.

[48] George Tzenakis, Angelos Papatriantafyllou, Hans Vandierendonck, Polyvios Pratikakis, and DimitriosS. Nikolopoulos. Bddt: Block-level dynamic dependence analysis for task-based parallelism. In Chenggang Wu and Albert Cohen, editors, *Advanced Parallel Processing Technologies*, volume 8299 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2013.

[49] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *HotPar*, 2011.

[50] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[51] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[52] Jing Yu, María Jesús Garzarán, and Marc Snir. Languages and compilers for parallel computing. chapter Techniques for Efficient Software Checking, pages 16–31. Springer-Verlag, Berlin, Heidelberg, 2008.

[53] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime asynchronous fault tolerance via speculation. In *CGO*, 2012.

[54] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. Daft: decoupled acyclic fault tolerance. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 87–98, New York, NY, USA, 2010. ACM.