Fault-Tolerant Operating System for Many-core Processors

Amit Fuchs
Fault-Tolerant Operating System for Many-core Processors

Research Thesis

In partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Amit Fuchs

Submitted to the Senate of the Technion — Israel Institute of Technology
Tevet 5778 Haifa December 2017
The research thesis was carried out under the supervision of prof. Avi Mendelson, in the Faculty of Computer Science.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xiii</td>
</tr>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>Glossary</td>
<td>3</td>
</tr>
<tr>
<td>Abreviations</td>
<td>5</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Achievements .......................... 10
1.2 Related Work .......................... 12
  1.2.1 Distributed Operating Systems .... 12
  1.2.2 Parallelization Techniques ...... 13
    1.2.2.1 Hardware Parallelism .......... 13
    1.2.2.2 Native Multi-threading Libraries ... 14
    1.2.2.3 Message Passing ............... 14
    1.2.2.4 Transactional Memory .......... 14

### 2 Architecture Model

2.1 Clustered Architecture .............. 17
2.2 Distributed Memory .................. 19
  2.2.1 Local Memories ................... 20
  2.2.2 Globally Shared Memory ........... 20
    2.2.2.1 Scalable Consistency Model .... 21
    2.2.2.2 No Hardware-based Coherency ... 21
    2.2.2.3 Distance Dependent Latency (NUMA) ... 22
    2.2.2.4 Address Translation ........... 22
  2.3 Processor Faults .................... 23
    2.3.1 Threats Terminology .............. 23
    2.3.2 Sources for Processor Faults ..... 24
    2.3.3 Implications of Transistor Scaling ... 25
    2.3.4 Treated Faults ................... 26
      2.3.4.1 Permanent Faults (a.k.a. Hard-faults) ... 26
3 Operating System Goals
  3.1 Fault-Tolerance Goals
  3.2 Performance Goals
  3.3 Additional Considerations

4 Design Principles
  4.1 Application Programming Model
    4.1.1 Dataflow Flavored Execution
    4.1.2 Implicit Synchronization
    4.1.3 Explicit Parallelization
    4.1.4 Building Programs
    4.1.5 Parallelization Example
  4.2 Partitioned Memory Layout
    4.2.1 Local Memory Role
    4.2.2 Shared Memory Organization
      4.2.2.1 Partitioned Global Address Space
      4.2.2.2 Low-Level Memory Management
  4.3 Fault-Tolerance and Resiliency
    4.3.1 Tolerating Hard Faults
      4.3.1.1 Detecting Permanent Faults
      4.3.1.2 Recovery Process Overview
      4.3.1.3 Fault Detection and Recovery Scenario
    4.3.2 Tolerating Transient Faults
    4.3.3 Low-Cost Fine-Grained Checkpoints
  4.4 Additional Principles
    4.4.1 Software-only approach
    4.4.2 Everything is Wait-Free
    4.4.3 Software-Managed Consistency
    4.4.4 No Atomic Conditionals
    4.4.5 Completely Decentralized
    4.4.6 Easily Fit Existing Systems

5 Functional Overview
  5.1 Inter-node Communication
    5.1.1 Block Transport Layer
      5.1.1.1 Behavior on Faults
      5.1.1.2 Poor Man’s Transactions
      5.1.1.3 Extending Transactions
      5.1.1.4 Transport Backplane
      5.1.1.5 Channel Operation

2.3.4.2 Transient Faults (a.k.a. Soft-faults)
6.4 Experiments and Results ........................................... 94
  6.4.1 Application Timeline ........................................ 95
  6.4.2 Recovery from Faults ......................................... 98
  6.4.3 Speedup Through Parallelization ............................ 99
  6.4.4 Overhead of Parallelization and Reliability ............... 102
    6.4.4.1 Effects of Parallel Workload ........................... 102
    6.4.4.2 Impact of Cores Partitioning ........................... 103
  6.4.5 Impact of Parallel Workload ................................. 105

7 Hands-on Experiment ................................................ 109
  7.1 Goal of the Experiment ......................................... 109
  7.2 Instructions to Start ........................................... 109
  7.3 Expected Output ................................................. 112

8 Concluding Remarks ................................................. 117

9 Appendix ............................................................... 119
  9.1 Experiment Output of 512 Cores ............................... 119

References ............................................................ 123

Hebrew Abstract ....................................................... 136
# List of Figures

1.1 On-the-fly recovery from a hard fault on a 1024-core processor 11

2.1 Heterogeneous cluster architecture for a many-core processor 18
2.2 Top-level organization of the many-core processor 19

4.1 High-level roles given to the architectural components 41
4.2 Node memory layout 42
4.3 Hard-fault recovery process 47
4.4 On-the-fly hard-fault recovery - 128 cores 48

5.1 Shared memory layout 54
5.2 TaskWrite message structure 57
5.3 Structure of a task identifier (UFI) 80
5.4 Structure of a TaskDescriptor 81
5.5 Binary layout of a dataflow thread 83

6.1 Full-system many-core simulation 89
6.2 Throughput and parallel workload during run-time 96
6.3 Workload versus task duration 97
6.4 Workload versus nodes - 4 cores per node 97
6.5 Workload versus cores per node - 4 nodes 98
6.6 Workload versus cores per node - 8 nodes 98
6.7 Workload versus nodes - long tasks duration 98
6.8 Increased hard-fault detection delay 99
6.9 Excessively long hard-fault detection delay 100
6.10 Speedup versus the number of cores 101
6.11 OS dominated cycles on different workloads 103
6.12 OS dominated cycles on different configurations 104
6.13 Run-time with varied tasks duration and cores partitioning 105
6.14 Varied application workloads 106
6.15 Impact of cores partitioning on workload 107

7.1 Simulation screenshot 112
7.2 Recovery status screenshot 114
7.3 Double Execution in action 115
Listings

4.1 The famous recursive definition for Fibonacci series . . . . . . 39
4.2 Explicit dataflow version of the Fibonacci series . . . . . . 40

7.1 Example output trace of the dataflow engine . . . . . . . 112
7.2 Hard-fault detection and recovery . . . . . . . . . . . . . 114

9.1 Node trace output on a 512-core experiment . . . . . . . 119
Abstract

Creating operating systems for many-core processors is a critical challenge. Contemporary multi-core systems inhibit scaling by relying on hardware-based cache-coherency and atomic primitives to guarantee consistency and synchronization when accessing shared memory. Moreover, projections on transistor scaling trends predict that hardware fault rates will increase by orders of magnitude and the microarchitecture alone could not provide adequate robustness in the exascale era. Resilience must be considered at all levels; operating systems cannot continue to assume that the processors are error-free.

A fault-tolerant distributed operating system is presented, designed to harness the massive parallelism in many-core distributed shared memory processors. It targets scale-out architectures with 1,000-10,000+ fault-prone cores on-chip and waives traditional hardware-based consistency over the shared memory. The operating system allows applications to remain oblivious to hardware faults and efficiently utilize all cores of exascale systems-on-chip without performing explicit synchronization.

To scale efficiently and reliably as the number of cores rapidly increases while their reliability decreases, the new operating system provides fault-tolerant task-level parallelism to applications through a coarse-grained data-flow programming model. A decentralized wait-free execution engine was created to maximize task parallelism, scalability, and resiliency over unreliable processing cores. It combines message-passing and shared memory without strong consistency guarantees. Fine-grained checkpoints are intrinsic at all levels, enabling on-the-fly recovery of application-level tasks in the case of hardware faults, automatically resuming their execution with minimal costs.

A prototype implementation of the new operating system was experimentally evaluated on a many-core full-system simulator, the presented results exemplify the characteristics and benefits of the new approach.

Keywords— exascale, many-core, fault-tolerance, resilience, operating system, data-flow, wait-free
Glossary

**Cluster** In the context of this work, a cluster is the set of tightly coupled symmetric or heterogeneous nodes and their shared resources.

**Core** A computation element within a node, capable of performing calculations and execute threads. A core can execute instructions independently of other cores but may share hardware resources with them, such as caches and buses.

**Data-flow** Execution model that helps fragmenting large operations to small tasks that may execute in parallel according to their dependencies, see Section 4.1.1.

**Dataflow Task** Task, in short, is a reference to executable instructions and their accompanying details, such as the required inputs and conditions for execution. A task is associated with a thread that may only begin execution when all of the required inputs are ready. Once started, the thread of a task may run uninterrupted and does not produce any system-wide side-effects until its completion.

**Error** See Section 2.3.1

**Failure** See Section 2.3.1

**Fault** See Section 2.3.1

**Hard fault** Permanent and complete core failure, see Section 2.3.4.1.

**JIT** Just-in-time compilation, to compile application code during its runtime, not only in advance to its launch.

**Mean Time to Error** A statistical measure of components reliability. It is the averaged length of time that the device is working without faults.

**Node** A group of tightly coupled symmetric cores and their nearby hardware resources. They share communication buses to other nodes and have their own private main memory. A node-local kernel governs over the resources of the node and acts as its façade.
**NUMA** Non-uniform memory access. On such architecture, some memory regions may take longer to access than others. For a certain core, this work considered three types of memory regions that may have significantly different access latencies: those to the local memory of the node, those to the shared memory region associated with the local node, and those to shared regions of other nodes. The difference in latencies may be because some regions are physically and/or logically further away from the core, so operations need to travel through additional buses.

**Processor** Contains multiple, possibly heterogeneous, cores or nodes and their supporting hardware resources, packaged together to a single integrated circuit. Another name for a processor is CPU (Central processing unit).

**Ready task** In the context of this work, these are dataflow tasks that have received all of their inputs and are ready to run.

**Scheduler** In the context of this work, refers to the operating system module that manages the life-cycle of dataflow threads at a high-level. It assigns memory and computing resources to threads and manages their state until completion. Note that short-term scheduling of ready threads within a node is the responsibility of a different module.

**Thread** A list of instructions that execute sequentially. A thread can belong to a dataflow task or any normal code, it can start execution as soon as it is created.

**Thread frame** Memory space that is created for a task to hold its received input values. An analogue for it in common conventions is the stack space of subroutines that receives parameters from the caller.

**Transient fault** Temporary core condition that results in computation errors, see Section 2.3.4.2.

**Wait-free** No operation is ever delayed or blocked by another.
Abreviations

**API**  Application programming interface

**CMOS**  Complementary metal-oxide-semiconductor

**DE**  Double Execution

**DF**  Data-flow. *Glossary: DF*

**DSM**  Distributed shared memory

**DSP**  Digital signal processor

**FIFO**  First in, first out

**GPGPU**  General-purpose graphics processing unit

**HPC**  High-performance computing

**ILP**  Instruction-level parallelism

**ISA**  Instruction Set Architecture

**JIT**  Just-in-time. *Glossary: JIT*

**MCA**  Machine-Check Architecture

**MPI**  Message-Passing Interface

**MTTE**  Mean Time to Error. *Glossary: MTTE*

**NUMA**  Non-uniform memory access. *Glossary: NUMA*

**OS**  Operating System

**SPSC**  Single producer single consumer

**UFI**  Unique Frame Identifier

**VFP**  Virtual Frame Pointer

**VLIW**  Very Long Instruction Word
This thesis presents the new exascale operating system that was designed, implemented to a working prototype, and experimentally tested a many-core full-system simulator, as well as evaluation results that illustrate the behavior and performance of the design. The operating system includes fault-tolerance mechanisms that allow keeping the running applications oblivious to hardware errors caused by transient, intermittent, or permanent faults. To support these reliability requirements while scaling to thousands of cores, the new operating system is distributed, completely decentralized and uses only wait-free structures on the shared memory. Strong coherency guarantees on the shared memory are not required; compared to existing multi-core architectures, only a weak consistency model is assumed on all global address space. The proposed system incorporates dataflow principles to assist the reliable distribution of parallel user applications while minimizing the overhead of fault-tolerance and delays on fault recovery.

As frequency stopped scaling and extra performance is still required, the common way of achieving more performance out of a computational system is by increasing the number of computational units at all levels, and to increase the integration with their peripherals, such as memory, DSPs, GPGPUs and various other controllers. A single processor will soon contain hundreds or even thousands of cores, composed of 100-1000 billion of transistors, and a single data center will be made out of millions of processors [1]. Taking the advantage of such vast amount of parallelism requires the redesign of the CPU cores organization, the memory hierarchy and its semantics [2], [3]. In order to exploit the hardware scaling with the best utilization of resources, many SW/OS/HW interfaces require major changes. New interfaces may vary from exposing the CPU organization to the programmer, similarly to the case of GPGPU, or provide an abstracted environment for SW that can expose maximal parallelism out of sequential code, as in the case of dataflow programming. The operating system has a critical role in guaranteeing the correctness of the execution as well as for creating uniform interfaces between the applications and different types of hardware architectures so that the system can be easily adapted to scale out large computations on evolving machines.

The continued scaling down of transistor feature size adversely affects the lifetime and reliability of processor die, from increased thermal densities,
accumulated defects and process variation [1]. It is therefore expected that future many-core devices will be orders of magnitude more prone to various types of transient and permanent faults. As indicated by [4], with the scaling down of transistors size, core faults will no longer be considered as extremely rare events. Processor faults are expected to gradually become common even on the best hardware designs [5], and core transient and hard-faults could not be avoided any longer. As hardware designers acknowledge that faults become a noticeable issue in modern systems, they must also be treated as first-class citizens in the eyes of software developers. Robust systems today usually maintain fault-tolerance at the macro scale, at the system level, rather than at the micro scale of the processors. When considering the tremendous amount of resources in many-core processors and their increased susceptibility to faults, this work assumes that the probability of hardware failures is significant. Therefore, design of future systems should treat faults at all levels, including the operating system that needs to answer these issues with fault detection and recovery.

As core count rapidly increase, having nearby low-latency resources such as registers, caches and memory remain critical for performance. Therefore, in order to balance between hardware growth, performance requirements and the need for reliability, the massively parallel operating system presented here is based on a clustered architecture that allows continued scaling from multi-core to many-core processors. To support such architecture, an additional level of the hierarchy is needed for cores, interconnects and memory. Thus, this work assumes a hardware model that is comprised of nodes with several tightly coupled cores and private memory, and interconnect to a globally accessible shared-memory. Each node resembles an existing off-the-shelf multi-core processor with strong cache-coherency over its main memory, but the shared memory is assumed to only be weakly coherent, since strong coherency protocols limit significant scaling toward future exascale processors.

This work focuses on creating a new operating system for general-purpose massively parallel processors, many-core devices that may be so highly integrated as to fit into a single die. Such an operating system needs to allow applications to achieve the best utilization of resources with low overhead, to support efficient scaling with increased number of cores, to employ load-balancing in order to improve overall performance, to avoid creating thermal hot-spots, and more. Imbalance in the utilization of resources may not only severely degrade performance, but it can also increase power consumption and long-term chip wear due to uneven thermal cycling [6], [7]. Another reason to avoid excessive concentration of resource utilization on a fault-prone hardware is that the effectiveness of fault-tolerant mechanisms may
depend on the system being distributed to isolate faults. Thus, the risk to physically concentrated data is increased in case of faults, i.e., lots of eggs in one basket.

In order to sustain continued scaling in performance as the number of cores increases, this work assumes that future processors will not only scale vertically, i.e., scale up with increased core count and memory capacity, but they will also scale horizontally, i.e., scale out to form several nodes in a single processor, as supported by [8]. In other words, it is assumed that scaling out is needed to efficiently utilize thousands of cores on a single processor, creating an additional level in cores and memory hierarchy. This work then suggests partitioning the hardware resources into nodes; each node contains a set of cores, local memory, caches, and interconnect to neighboring nodes and shared resources. A node may resemble a current off-the-shelf multi-core processor that is combined with its own private main memory (RAM). Besides the local memory in each node, additional memory domains are addressable and accessible from all nodes. The node-local memories are assumed to guarantee strong cache-coherency, as provided by hardware in current commodity processors, but the shared memory may only be a weakly coherent domain. Compared to the strong coherency on private memories, where writes are implicitly propagated to readers by the hardware, explicit instructions are required to propagate data and maintain consistency over the weakly coherent shared memory. Weak consistency on the shared address space is assumed because it allows future processors to break the cache-coherency wall [9], that would otherwise limit the scalability of the system. To allow for further scaling regarding possible shared memory hardware implementations, this work considers that the globally shared memory may be distributed on the processor die and have separate partitions with affinity to each node, i.e., non-uniform memory access (NUMA). Therefore, each node benefits from faster access times to data that is physically close to it in the distributed shared memory. Additional details on the assumed hardware architecture are described in Chapter 2.

This thesis presents a new distributed operating system prototype that supports the reliable execution of applications on multi-core and many-core (1,000-10,000+) processors using a new scalable SW/OS/HW co-design. The proposed task-dataflow distributed operating system aims at advancing existing execution models to a higher level, by introducing of new interfaces between the applications and operating-system, together with new HW interfaces. The user applications are written in existing programming languages such as C/C++ and can use the new run-time framework to define their desired work as a directed graph of dataflow tasks. Tasks are executed by the presented operating system with respect to their declared data
dependencies to achieve maximum parallelism and avoid using unnecessary synchronization mechanisms.

The foci of the work are creating a new distributed operating system for future exascale clustered architectures, a scalable, decentralized, and resilient dataflow execution engine and a fault-tolerant run-time environment that is wait-free and relies only on weak consistency guarantees from the shared memory.

The thesis is structured as follows: Section 1.1 mentions the main contributions and achievements of this work. In Section 1.2, a short survey of the existing works is done with emphasis on works that relate to the stated goals and problems. Chapter 2 presents the assumed baseline architecture that the proposed system targets. Top-level goals for the work are discussed in Chapter 3. Chapter 4 introduces the main design principles of the new operating system and Chapter 5 provides a functional overview of its modules. Chapter 6 describes the framework used to simulate and evaluate the system and Section 6.4 presents the experimental results of behavior and performance on different architectural configurations. Chapter 7 describes how the reader can configure and experiment with the operating system. The thesis is concluded in Chapter 8 that presents final remarks.

1.1 Achievements

A fault-tolerant many-core operating system was created. It assumes unreliable cores and does not require strong hardware-based consistency over the shared memory. The design has the following characteristics:

- Scalable to thousands of cores on a single processor, oriented toward clustered architectures with a distributed memory.
- Software-managed consistency on shared memory, neither hardware-based consistency nor atomic primitives are assumed.
- Resilient in the face of permanent and transient hardware faults.
- All operations on shared memory are wait-free.
- Completely decentralized microkernels.
- Hybrid mechanisms use both message-passing and shared objects.
- Software-only solution, assumes off-the-shelf cores.
- Exposes a programming interface for task-parallelism that allows applications to remain oblivious to faults, architectural details, and synchronization.
The new operating system has been implemented from the ground up to a working prototype as a distributed library OS. All of its run-time mechanisms described in this thesis were implemented from scratch, that is to say, using only standard libraries. The latest source code will be made available in a public repository [10].

The implementation was experimentally evaluated in an existing full-system simulator for many-core architectures. The tests included:

- Up to 1024 cores with various microarchitecture and macroarchitecture configurations.
- Behavior and performance results for many faulty and non-faulty cases.
- Extensive level of details on functional and timing behavior.
- Flexible configurations of the memory and cache hierarchy, their policies and latencies.

Preliminary versions of the operating system produced in this work were presented in [11]–[14].

![Figure 1.1: On-the-fly recovery from a hard fault on a 1024-core processor](image)

The shades of color correspond to 32 nodes running on the full-system simulator, each having 32 cores. A single application was utilizing all cores during the fault, it remains unaware as the system detects the fault and automatically recovers and redistributes the abandoned tasks to the remaining cores.
1.2 Related Work

Considering that a very broad range of different types of functionality was created in this work, covering the operating system design, a working implementation for it, and its evaluation framework, only a short survey of the related work is be presented in this section. Additional literature, related work, and alternative solutions to specific objectives and problems treated in this work are mentioned in the body of the thesis on the relevant sections, as well as opportunities for future research.

This work relates to the framework of European Union FP7 Future and Emerging Technologies (FET) project Teraflux [15], [16] in the area of Teradevice Computing, its partners contributed in other ways with other methods to the problems treated in this thesis. While this thesis presents an independent solution that was created from the ground up during the research period, some of the top-level assumptions on future many-core architectures are biased toward those adopted by the Teraflux consortium.

1.2.1 Distributed Operating Systems

The emphasis in this section is on other work with a similar top-level goal: an operating system for a single-chip exascale processor. Other related types of operating systems exist, such as those for super-computers or cloud services. Those systems target thousands of cores on clustered architectures, but their hardware is composed of separate machines and a network that connects them [17], not a single processor and shared memory. Therefore, although some of those reviewed during the research have similarities to the presented operating system, they are also fundamentally different when addressing inter-node communication and fault-tolerance.

Hive [18] is a distributed operating system for shared-memory systems with many similarities to this work. It assumes a single-chip multiprocessor (albeit with only a few cores), it provides fault detection and recovery mechanisms within a shared memory system, and it uses a combination of message-passing to communicate and shared memory objects to achieve its goals. It does, however, relies heavily on non-standard hardware features and cache-coherency that do not scale well, and it does not provide recovery for applications that were running on cores that suffered a fault.

Other distributed operating systems, such as Barrelfish [19], Helios [20], and fos [21], achieve scalability on heterogeneous many-core processors not only by abandoning the requirements for consistency on the shared memory, they abandon direct sharing altogether and rely only on message-passing.
The proposed system, however, does use direct memory sharing, but manages the consistency explicitly in software. Moreover, those mentioned systems do not inherently provide fault-tolerance, which is a critical requirement for the run-time environment created in this work. [22], [23] suggest software-managed consistency as a solution to the scalability problem of traditional shared-memory systems, but do not provide an overall operating system solution, nor address reliability issues.

1.2.2 Parallelization Techniques

1.2.2.1 Hardware Parallelism

For the last three decades, many different techniques were proposed in order to achieve more instruction-level and thread-level parallelism, such as:

1) VLIW (Very Long Instruction Word) [24] tries to achieve this goal by using simple HW and sophisticated compilers.

2) OOE (Out-of-Order Execution) tries to achieve this goal by using an extensive amount of HW that tries in a dynamic way to expose the dependency graph among instructions within an execution window of nearby instructions [25].

3) Simultaneous Multithreading (SMT) [26] permits to distribute instructions of several independent threads among several super-scalar functional units in a single processing core.

4) Chip-level multiprocessing (CMP) [27] integrates several independent processing cores into a single processor.

These former techniques are widely used in current processors, focus on maximizing the utilization of hardware functional units, but do not deal with automatically improving coarse-grained parallelized execution of multiple dependent threads, e.g., a producer-consumer pair of threads. Those techniques are limited to provide parallelism at a very low level. Considering they use a micro-scale view of computation tasks, they are too nearsighted by themselves to schedule several coarse-grained dependent threads, let alone if they span to other computation units [25], [27], [28]. Moreover, those mentioned techniques do not have a concept of fault-tolerance, and consequently do not provide the OS and applications any assistance to deal with possible faults. It is then a harder task for SW to make sure that the execution of dependent threads survives faults of single cores and even of a set of cores and their nearby resources, a.k.a. node.
1.2.2.2 Native Multi-threading Libraries

These libraries, such as pthreads [29], provide interfaces to create coarse-grained parallelism and synchronization. Native libraries are usually used at the lowest level in a multiprocessing software stack. They are versatile and can be used to implement various parallel programming models. Like hardware-level parallelization, these libraries do not inherently provide services for fault-tolerance and distributing executions beyond a single multi-core processor.

1.2.2.3 Message Passing

Message Passing Interface (MPI) is the de-facto standard for inter-process communication in current high-performance computing (HPC) systems. MPI has no inherent support for fault-tolerance [30], but considering it became a common API for large-scale parallel programs, a lot of attention has been given to extend it with fault-tolerance [5], [31], [32]. Although there is continuous progress, the current standard techniques for fault-tolerance on high-performance computing use various types checkpoints that are complicated and costly [33], [34].

1.2.2.4 Transactional Memory

Transactional operations on shared data structures simplifies the programming of concurrent and possibly dependent operations [35], it makes groups of operations on shared data appear to be performed atomically. This property is valuable on a shared-memory system, where concurrent accesses to shared data might conflict and leave the result in an inconsistent state.

Transactional memory [36] operations have several qualities that resemble the execution of dataflow tasks, which is the focus of the runtime engine created in this work. In both methods, there are no side-effects until an operation completed successfully, and if an error/conflict occurred during processing, the thread/transaction can be safely re-executed. Deferring side-effects is also required in both paradigms, since the intermediate results must be kept in a private location before they are committed to the globally observable state. The similar traits between paradigms encourage isolation between concurrent operations and provide inherent support for parallelization, error detection, and safe retry, all at the level of the run-time environment, instead of user code.

Transactions are easy to grasp as a means to work around synchronization problems, but optimistic implementations still have widely inconsistent
performance on various scenarios that are highly dependent on specific workloads [37]. In addition, both software and hardware implementations of transactional memory have significant efficiency deterioration when scaling transaction counts or sizes [38]. Software transactional memory (STM) implementations have large overheads from the mechanisms that provide conflict detection and resolution, while current implementations of hardware transactional memory (HTM) are limited by a small capacity for concurrent transactions [39], [40]. The better-performing hardware implementations depend on strong consistency of the shared memory, but in this work, this is only assumed to exist over the private memory within nodes, not over the global shared memory (see Chapter 2).
While future massively-parallel devices may have all of the processing cores on a single die, it is unlikely that they can all be efficiently managed using the same hardware and software designs as the ones used in contemporary processors [3], [8], [41]. The organization of hardware elements such as processing cores and memory hierarchy needs reconsideration, and so does the interfaces that expose them to operating systems and applications. One of the main reasons for the next evolutionary step is that current hardware support for multi-core processing does not scale well as core counts increases, mainly since cache-coherency mechanisms and shared buses become noticeably inefficient with more than several cores [9]. Many applications and the matching operating-system support that was designed for multi-core processors rely on the traditional strong guarantees from hardware and so they do not efficiently scale as core number increases.

An overall solution to reliably harnessing future massively parallel devices must, therefore, involve changes in all levels of the control hierarchy. Both hardware and software need to adapt to efficiently and reliably span out computation resources on a more distributed architecture. To allow better scalability, i.e., efficient utilization of an increasing number of computational units, this work suggests using a clustered hardware architecture within a single processor, and with it, a distributed operating system is proposed that provides additional layers of abstraction for applications.

This chapter describes the assumption and delimitations for this work regarding future many-core architectures and their constraints. Section 2.1 and Section 2.2 describe assumptions on the additional partitioning of the hardware compared to existing multi-core architectures. Section 2.3 describes the fault model that is assumed on future many-core processors.

### 2.1 Clustered Architecture

In order to efficiently scale toward future architectures, it is assumed in this work that future many-core processors will use a clustered architecture that will contain many, possibly heterogeneous, cores. Cores are assumed to be based on off-the-shelf components as in current x86-64 hardware. The
cluster is partitioned into nodes with several tightly coupled cores, which share communication links and hardware elements, such as caches, main memory, and interconnect to the rest of the chip. Concordantly, the memory hierarchy is assumed to be partitioned to have node-local memories and an additional distributed global memory. Section 2.2 provides additional information.

Thus, each node resembles a multi-core microprocessor (CMP [27]) that contains low-latency caches and maintains a strong cache-coherency on its main memory, as current processors do. The nodes can all be symmetric or heterogeneous [2]; a heterogeneous system may contain several types of nodes with different hardware capabilities. Specialized nodes can be made for excellent serial execution performance, other nodes might be specialized toward communication interfaces, storage, or graphics, and the rest of the nodes can be specialized in low-power execution of large numbers of small computational tasks.

Each node runs its own kernel/microkernel that is responsible for managing its resources, such as cores, local memory, and other resources such as a network or disk access. Each kernel can schedule new tasks and collaborate with the other nodes over interconnect channels or non-coherent shared memory. The assumptions of this work can fit various possible heterogeneous architectures [42], some of which could be similar to the high-level architecture of contemporary super-computers [17]. An example of a possible architecture that meets the assumptions in this work is illustrated in Figure 2.1, it may include:

- Front-end nodes that are responsible for user interactions. Their cores are designed toward the best single-thread performance and they can
run a general-purpose operating system such as Linux. They delegate the computational and communication workload to the other nodes.

- I/O nodes responsible for communication with the network, storage or other hardware devices that may run a specialized kernel for that purpose.

- Compute nodes that run only computational tasks in a power-efficient manner. These nodes can have a very basic kernel and have no access to the outside world. All communication and system operations are performed by the nodes running a full-blown kernel.

## 2.2 Distributed Memory

The memory in a future exascale device is assumed to be partitioned, either physically or logically, into private per-node local memory regions and an additional distributed shared memory (DSM) with on-chip global accessibility. A **NUMA** behavior is assumed in several aspects: accesses to the partitioned global memory may incur higher latencies than accesses to local node memories, and in addition, the same global location may have various latencies when accessed from different nodes. It is assumed in this work that the hardware implicitly maintains strong cache-coherency over the private memory of each node, similarly to current processors, but no hardware-based consistency is assumed over the shared memory, which requires the software to explicitly request the propagation of data.

**Section 2.2.1** and **Section 2.2.2** give additional details about the different memory types. **Section 4.2** will later describe the main roles that were assigned to different memory regions by the proposed operating system.

**Figure 2.2:** Top-level organization of the many-core processor

The assumed architecture is comprised of multi-core nodes with their local memories, and a partitioned globally addressable shared memory.
2.2.1 Local Memories

Since this work assumes that each node behaves like the current commodity multi-core processors (Section 2.1), it is expected that the nodes implement a hardware-controlled cache-coherency protocol over their private local memories. A strong consistency model is therefore assumed on the node-local memories as implemented in hardware by off-the-shelf x86-64 architectures, such as current Intel or AMD multi-core processors. The term strong consistency can be ambiguous, and furthermore, the memory consistency specifications for Intel and AMD processors are also ambiguous [43], [44]. The goals of this work (Chapter 3), however, can be achieved without depending on precise specifications of the microarchitecture.

In the context of this work, the strong consistency of memory within a node means that the hardware automatically enforces many additional constraints on the ordering of memory operations compared to what is subsequently referred to as a weak consistency model (Section 2.2.2). For example, programs designed to use a memory with the stronger model assume that the hardware implicitly and automatically propagates memory writes to the caches of cores within a node to be available on any subsequent reads, so multiple writes are observed in a consistent order by all cores in the node. These consistency guarantees cannot efficiently scale from multi-cores to many-cores with the current generation of coherency protocols. This work therefore assumes a weaker consistency model on the global memory in order to support efficient scalability to large number of cores, the model will be further described in Section 2.2.2. The assumed consistency model on local memories of nodes is similar to the model presented in commonly available x86-64 processors, those are generally considered strongly consistent. More specific guarantees are similarly described in both the architecture manuals for Intel-64, IA-32 and AMD64 (Mar. 2017) [45], [46]. As previously mentioned in this section, the general description of the consistency model on local memories is sufficient for this work since the focus is on resource management on the higher levels of the global system.

2.2.2 Globally Shared Memory

This work assumes that the hardware infrastructure provides a globally-addressable shared-memory domain, which is accessible from all nodes. This memory is assumed to exist in additional to the private local memory each node has (Section 2.2.1). The following sections describe the main characteristics of the memory model.
2.2.2.1 Scalable Consistency Model

Strong consistency models are considered intuitive and conceptually simple, but the ease of programming they provide imposes strict restrictions on hardware, which increases its complexity and hinders performance [47], [48]. Therefore, in order to support efficient scaling to many-core processors, the consistency model assumed on the shared memory between nodes is weaker than that on the local memories. Relaxation of the strong constraints that are guaranteed on the local memories is assumed because the current generation of cache-coherency protocols inhibit scalability to large numbers of cores [9]. The overhead of traditional interconnect architectures for cores and memory becomes significant even on 8 and 16 cores [49]. Weakening the consistency model on the shared memory allows decreasing the overhead associated with cache-coherency protocols over it. Applying different memory models for local and shared memory domains unlocks significant limitations on scaling without much consequences at a node-local level. The programming paradigms used at the local node scale can remain unchanged since this work assumes that nodes use existing components for their private memory and caches, which provide traditional memory consistency as contemporary processors. Therefore, it is believed in this work that future processors are likely to present both a weak consistency model at a global scale, on a memory shared between all nodes, and a strong consistency model at a local scale, on private memories within nodes.

2.2.2.2 No Hardware-based Coherency

The consistency model of the shared memory architecture assumed in this work is based on acquire and release operations. Under the assumed semantics, a reader must explicitly acquire a region of the memory to access its latest version, and a writer must release a region it modified so the changes will propagate and become observable on a subsequent acquire. These two memory fences are needed to be issued explicitly by software in order to exchange data on the shared memory and trigger consistency between nodes. The hardware is not assumed to automatically propagate data between nodes, not to mention enforce any ordering constraints between memory accesses.

This relaxed consistency model demands much less from the hardware compared the stronger model on local memory (Section 2.2.1), it is therefore believed in this work that it is the key to significant optimizations on hardware implementations that will allow efficient scaling to many-cores on a single die. Other works also argue that managing consistency by software
can be a suitable solution to the coherency wall [9] in many-core processors, and have proposed different semantics [22], [23]. The assumptions of the semantics described here were found sufficient in this work, the implementation of the operating system can provide the required synchronization for the chosen programming model (Section 4.1).

2.2.2.3 Distance Dependent Latency (NUMA)

Regardless of the consistency model on shared memory, this work assumes that the shared address space may have different access times depending on the originating core and physical location of the referenced memory location. In addition, accesses to any location on the shared memory are assumed to be slower compared to accesses on the node-local memory. An additional assumption in this work is that the shared memory may have separate partitions with affinity to particular nodes, i.e., provide consistently lower access times when partitions are accessed by their affined node, due to the applicable measures of distance to the data. The measure of distance can be physical or logical, e.g., the number of hops between memory controllers. Therefore, an implementation can leverage the hardware-based advantage a node has when accessing certain memory regions.

2.2.2.4 Address Translation

This work assumes that each node has hardware support for virtual memory translation similar to existing multi-core processors. This feature allows software to access the physical memory using addresses in a virtual address space and rely on hardware support to map virtual memory addresses to the physical address. In order to allow completely independent operating system kernels to run in the nodes of the distributed system, virtual memory is assumed to function in each node independently from the rest, as if they were on separated processors. As illustrated in Figure 2.2, each node may access the globally shared memory and its own private local memory, the local memories of other nodes are assumed inaccessible. The physical memory can be mapped correspondingly in each node, virtual addresses can be translated by hardware to the private local memory or the shared memory.

To maintain consistent addressing to the shared memory from all nodes, it is preferable that all nodes can map the physical address of the shared memory into the same virtual address in their private virtual space. This way, pointers to the global shared memory can be transferred between nodes as-is and dereferenced without requiring extra translations. If the
assumption is not met, it means that each node might map the shared memory into a different virtual address in its private space, and pointers transferred between nodes would not remain valid. There is an easy method to overcome this in software by the proposed operating system: instead of having pointers contain absolute addresses in shared memory, they contain offsets (i.e., distances) relative to the beginning of the shared memory. Offset pointers remain valid when copied to any address space, even if the shared memory was mapped to a different base address. Using offset pointers incurs negligible additional overhead compared because translations are normally only needed when starting and completing a task, as will be described in Section 4.1. This simple solution overcomes the requirement to have the same virtual mapping of shared memory on all nodes without additional hardware assumptions.

A kernel tweak was made in the evaluation platform (Section 6.1) in order to enforce consistent mapping of the shared memory across all nodes. The change simply disables a kernel feature that performs address space layout randomization (randomize_va_space kernel parameter on Linux), so booting symmetric nodes results in the same virtual address of the mapped shared memory in all of them.

2.3 Processor Faults

This section describes the assumptions made in this work regarding the expected faults in future exascale processor architectures. Section 2.3.2 presents a brief review on the causes for the expected increase of fault rates in future microprocessors. Section 2.3.3 shows the importance of introducing new fault-tolerance mechanism and Section 2.3.4 classifies the fault types assumed in this work.

2.3.1 Threats Terminology

This section provides brief definitions for the basic concepts that are often used in this thesis when referring to threats to the dependability of a system. The terms used in this thesis are identical to those generally used, they are listed here with their context for this work. Detailed definitions and classifications for various types of impairments are documented in [50], [51].

Fault A physical phenomena, a defect in the material or an incorrect internal state of a hardware component in the system. When a fault affects the
external state of the component and its manifestation becomes visible in the system, the deviation from the correct state is called an error.

**Error** A deviation from the correct state of a component in the system. Errors are considered to be the manifestation of faults, and are liable to lead to failures when they affect the delivered service that the system is meant to provide.

**Failure** This event occurs when the system does not perform its intended function and does not deliver the correct service. A failure is caused by an error in a hardware or software component of the system.

Additional definitions and specific assumptions regarding the faults treated in this work are described in Section 2.3.

### 2.3.2 Sources for Processor Faults

As it seems that Moore’s law continues to apply [1], transistors count rises and more of them are fitted into processors, allowing for the increasing number of cores per processor. Scaling more cores into a single integrated circuit naturally increases its overall power consumption and the power density per a unit of area [2]. The dynamic power consumption in contemporary CMOS processors is a significant limiting factor for the increase in processing power not only for low-power battery-operated devices but also for high-performance computers [32]. The following paragraph mentions several methods that hardware designers use to reduce the power consumption, along with their implications on processor reliability.

Considering only the increase in transistors count and neglecting other factors, the effect of switching more transistors has a proportional effect on the dynamic power consumption, since:

$$P_{\text{switching}} = C_{\text{eff}} \cdot V^2_{\text{core}} \cdot f \cdot N_{\text{trans}}$$

Where:

- $P_{\text{switching}}$ = Power from switching digital states [W]
- $C_{\text{eff}}$ = Effective load capacitance of each transistor [F]
- $V_{\text{core}}$ = Core voltage [V]
- $f$ = Core frequency on active mode [Hz]
- $N_{\text{trans}}$ = Number of switching transistors [−]

The increased power consumption from increasing $N_{\text{trans}}$ can be compensated for by lowering $f$, but this should not be done excessively because
operating frequency is critical for performance. Although it is possible to lower $C_{eff}$ using different process technologies [52], which is desired to lower both power consumption and switching latencies, one could also suggest lowering $V_{core}$ since its effect on power dissipation is squared. Designers are working to be able to operate microprocessor cores at ultra-low voltages to minimize current density and overall power consumption, which also lowers short-circuit and static power losses. However, lowering $V_{core}$ is more difficult than the other mentioned parameters [1] since it requires pushing manufacturing process verification corners to apply on deeper sub-threshold levels. Working near the threshold voltage, however, limits the maximum operating frequency and lowers noise immunity, which increases soft-failure rates [4], [32]. This mentioned limitation is one of the reasons that the dynamic power consumption (per a unit of area) does not scale down as fast as transistor density scales up. The increased power density combined with the reduction in transistor feature size directly affects the lifetime of the processor die.

These mentioned issues and additional implications of transistor scaling that adversely affect processor reliability were studied extensively [4], [53], those include manufacturing process variation, repeated thermal cycling, electromigration, mechanical stress, cosmic rays, dielectric breakdown and more.

### 2.3.3 Implications of Transistor Scaling

In order to produce many-core processors, semiconductor manufacturers will have to squeeze the verified fabrication process corners (i.e., design margins) so much that the robustness of the resulting die will significantly decrease and processors could not endure the same operating conditions for the chip lifetime expected today [1]. In other words, processor reliability under extreme conditions in terms of fabrication process variations, temperature, voltage and speed, could not be guaranteed as it is today through verifications on hardware fabrications. Therefore, it could be expected that such a device will be prone to various types of faults during its lifetime [2], [4], [32], [53]. If ignored, errors resulting from these faults may quickly cause the entire system to fail, at best; but in unlucky cases, sneaky errors may silently corrupt data and cause seemingly strange malfunctions, mysterious crashes or even incorrect application behavior after arbitrarily long delays [33].

---

1^Please note the terminology defined in Section 2.3.1.
2.3.4 Treated Faults

The overall reduction in processor die reliability can be liable to many types of faults in its components. Besides the processing elements, faults may also affect communication buses, on-chip memories and other controllers. The fault-tolerance mechanisms of the operating system proposed in this work are focused on treating reliability issues of the processing cores and their associated components that are local to the affected node, such as private memories, caches, and buses between the cores. The globally shared memory and its communication buses to the nodes are assumed to be protected from errors by hardware redundancy and error-correction codes.

It will be later described in Section 3.1, that the proposed operating system is expected to continue to function even assuming the possibility of the following faults, along with all running applications, much like a hard-drive occasionally suffering from a bad sector but the system continues to function.

The following sections summarize the assumptions and delimitations of the fault types treated in this work that are considered likely on the target hardware architecture.

2.3.4.1 Permanent Faults (a.k.a. Hard-faults)

Cores in future exascale devices are assumed to become increasingly prone to unexpectedly suffer hard-faults, as mentioned in Section 2.3.3. Such faults are caused by physical damage in the processor die and are therefore irreversible.

In the context of this work, a hard-fault may occur at any point in time and is assumed to immediately cause a critical node failure. Hard-faults are assumed to result in the immediate termination of code execution and loss of all private resources, such as memory and local caches. This work assumes that when suffering a hard-fault, a faulty node stops executing instructions, or at the very least, it does not run corrupted operating system code. Existing processors employ Machine-Check Architecture (MCA) [54], which is assumed in this work to detect this unrecoverable node state and stop the faulty cores. Architectures that exist today can detect errors in many of the processor subsystems and automatically recover from them, or report that they cannot [45], [46], [54].

In this work, this fail-stop case is often referred to as node failure, although a verbose description could be: the cores and all private resources of the node instantly and permanently vanished from existence without prior warning.
Further assumptions:

1. In a case of a hard-fault, all caches and private memory of the node are also instantly lost without any warning or flushing. Hence the caches and private main memory of the node are considered physically damaged and forever inaccessible as a part of the failed node.

2. The global shared memory is assumed to be unaffected by cores failing and remains accessible. It is assumed, however, that any on-going operation on it might stop part-way if the originating core has failed. It is explained later, in Section 4.2, that segments of shared memory are assigned to each node, but note it is only a logical partition, so it is reasonable to assume that those segments remain functional if the node failed. Those segments may, however, become corrupted by unfinished operations. A possible hardware-backed advantage a node has when accessing its assigned shared memory segment is utilized by the system but should not break this assumption.

2.3.4.2 Transient Faults (a.k.a. Soft-faults)

The continuous reduction in transistors feature size toward future exascale hardware will gradually increase the susceptibility of processing cores to intermittent and transient faults, also called single-event upsets (SEU) [55]. It is then expected in this work that cores may suffer temporary faults that induce errors in the results of machine instructions. Transient and intermittent faults are assumed to take effect for very short periods of time. Their effect can be limited to a single flipped bit in a computation result or corruption a group of successive instructions. Transient faults do not cause permanent damage to the affected core but may result in undefined behavior on running code, which may be disastrous if undetected.

This work separates the possible transient faults to several types that are described in the following paragraphs.

**Transient faults that are detected and resolved by hardware**

Existing architectural features for resilience are able to contain some types of soft-errors automatically, such as those mentioned in [54]. Additional hardware mechanisms for reliability may exist in future exascale processors and partly shield software from transient faults. Therefore, all levels of software can be oblivious to some types of transient faults, and consequently so is this work. This work proposes a software-only solution for reliable many-core distributed execution, hardware-supported features for resilience are not required.
The faults that are considered throughout the rest of this thesis are those for which no hardware mechanisms, such as MCA, are assumed to be able to detect and recover from.

**Transient faults on user code**  This work assumes that any type of soft error may occur when running user code. Such errors cannot affect the operating system’s kernel memory space, since its protected from direct user-level access (through existing privilege control mechanisms). Thus, software errors resulting from these faults only pose a threat to running applications. Specifically, they will not corrupt critical kernel structures, that might result in a complete node failure, similarly to a hard-fault. Even if an error of this type is ignored, the affected node and the global system can potentially continue to operate following the fault, but the affected applications will remain in an undefined state.

**Transient faults on kernel code**  A delimitation of this work is that the local node kernels on each node can be trusted, but if not - they halt. This is reasonable to assume in most systems that do not consider hardware-detectable transient faults. Therefore, if a node’s kernel is compromised by a transient fault that did not happen on user code, it is assumed to either recover from the fault or immediately stop as in the case of a hard-fault, without corrupting the global state. It will not, for example, uncontrollably write nonsense to the shared memory. The proposed operating system considers any unrecoverable non-user error as a hard-fault. It has mechanisms to detect and recover from hard-faults occurring at any point, but transient faults are only monitored on user code, not on kernel code. Section 5.3.5 provides additional details on detection and recovery from transient-faults.

**Prolonged temporary faults**  Although transient and intermittent faults are usually considered to appear for a short time, errors resulting from these faults may last for much longer. In this work, temporary soft-errors that halt correct node operation and remain in effect for a prolonged period are treated similarly to hard-faults. For example, a transient fault may cause an exceptionally critical error on the kernel of the node it affected and cause it to crash and reboot. Since a node takes a significant time to restart, the global system considers it to have suffered a hard-fault during that time.
Operating System Goals

The operating system created in this work aims at efficiently and reliably achieve high task-parallelism using the massive computational power of exascale processors, under the assumption that some resources may become faulty during their lifetime (Section 2.3). A required goal for this work is that the operating system will be able to overcome unexpected faults in hardware processing cores and automatically continue correct execution of all running applications, even those executed on the faulty cores. User applications may be allowed to remain unaware of the faults, although performance may degrade when fault rate exacerbates and resources are lost.

To allow software to efficiently and reliably scale to a large number of unreliable cores, the operating system should expose a programming interface for applications that allows them to safely harness the parallelism in many-core architectures while abstracting the architectural details and its flaws.

The following sections provide more details on the main goals of this work.

3.1 Fault-Tolerance Goals

When a massively parallel system is assumed, and faults cannot be avoided with a reasonable cost (Section 2.3.3), one may take faults as a first-class citizen on the system and should consider them at all levels of the hierarchy; i.e., hardware, operating system and applications. Thus, the operating system should achieve not only the best utilization of resources, low power, and balanced utilization, but also to detect and recover from different types of hardware faults. Traditional operating systems commonly used today are not resilient to processor faults; a hard-fault in any of the cores will quickly cause a full-system failure. Therefore, a design goal of the proposed many-core operating system presented in this work is to allow overall resiliency facing potential hardware faults at all levels. The anticipated faults range from causing temporary soft errors and up to permanent core failures that may gradually onset, mainly due to wearing out of the die material, as described in Section 2.3.4.2 and Section 2.3.4.1.

Additional details about the reliability aspects of the desired goals for this work are noted in the following paragraphs.
**Global resilience**  The proposed system should continue correct operation of all the operating system services and all running applications even in the face of faults. This work does not advocate that the OS is free to consider user applications as a necessary casualty to ensure its survival, a goal for this operating system is that all running applications obliviously continue their operation even when the system is met with the assumed hardware faults. A potential reduction in performance is however expected when faults occur.

**Minimize reliability overhead**  Reducing the cost of reliability is an important concern since faults are considered common phenomena [32]. Traditional checkpoint-restart fault-tolerance approaches do not fit well into the assumed architecture because they are complicated and inefficient to implement for large distributed systems [5]. The solution should strive to minimize the overhead of fault-tolerance on execution time and memory usage.

**Non-intrusive to applications**  Running applications should be oblivious to fault events, transient or permanent, even those that occurred in their own code. Moreover, no custom user-level code is required to enable and maintain overall reliability.

### 3.2 Performance Goals

Achieving maximum parallelism in a massively parallel architecture is a challenging design task for both applications and their supporting operating systems. Considering that contemporary operating systems are not able to efficiently scale to the many-core era [21], future systems will require special run-time frameworks and complementary application interfaces in order to efficiently maximize parallelism. Even if some software was perfectly designed to decrease its execution time proportionally according to the number of available compute cores, it is up to the operating system to maximize performance by harvesting as much of the available parallelism as the hardware can deliver. More specific performance goals are mentioned on the following paragraphs.

**Maximize parallelism and scalability**  The operating system should allow applications to harness the maximal parallelism available from the hardware. To maximize performance, application workload should be distributed among the processing cores and balanced, so cores are not left idle when
user tasks are waiting to be executed. The operating system should efficiently scale as the number of cores increases and achieve the best utilization of resources. To meet these challenges, the operating system should also be able to adjust the characteristics of its task placement and scheduling algorithms to dynamic applications behavior.

**Fast on-the-fly recovery**  The ideal objective is, simply put, to provide superfine-grained fault-tolerance with minimal recovery penalties. Traditional fault-tolerance approaches that use checkpoints are not only costly to maintain, but also during recovery. Recent message-passing technologies that use various types of message logging, coordinated and asynchronous checkpoints or replication at the framework or invasively at user level are also very expensive in many aspects [32]. The complication and cost to save, roll-back and recover on a massively parallel system is also high for existing message-passing algorithms [31]. This limitation will become even more prominent as parallelization increases.

The proposed system should quickly recover applications and minimize the penalties involved in recovery. Moreover, only the user tasks with instructions that were directly affected by a fault may be disturbed, unaffected tasks should not be disturbed, even those that belong to the same application.

**Minimize shared memory accesses**  Accessing the shared memory is expensive in many aspects, its use should be tightly controlled. Reasons to minimize redundant accesses include:

- Memory access latency increases as data moves further away from the core, see Section 2.2.2.3.

- Considering a fault can happen at any time, special care must be taken when designing and accessing all shared structures, so they will not be vulnerable to corruption. This is rarely possible to achieve for free.

- The assumed shared memory supports a weaker coherency among nodes compared to the local memory within a node, and requires explicit instructions when storing and loading data. Even simple data structures become complicated or even impractical to implement.

- Additional snags as those that are ubiquitous on multi-core systems when accessing shared resources. Those are expected to become more difficult and expensive to handle considering faults and memory semantics.
3.3 Additional Considerations

This section will expend the discussion to some additional fundamental objectives and constraints. This should help point out recurring considerations that affected the proposed design. To simplify the initial discussion, let us first assume a device with the described clustered architecture and memory model but completely fault-free. A distributed run-time framework for the simplified device does not require any fault-tolerance to hard or soft failures. To communicate between nodes, the imagined system uses the shared memory as a communication channel and run-time scratch storage.

Even if the distributed system uses the most efficient data structures and (magically) avoided any performance deterioration from concurrent access to them, the solution will need to overcome or avoid possible bottlenecks from frequent accesses to the shared memory. Obvious sources for these bottlenecks in a naive solution may be from the tempting simplicity of keeping task management and scheduling related lists and queues in the shared space, since keeping this data shared will greatly ease both load distribution and fault-tolerance. Doing so carelessly, however, can introduce the need for very frequent accesses to the shared memory, and possibly a significant overhead from synchronizing concurrent accesses and wasteful load on the hardware.

Therefore, to minimize frequent and possibly redundant shared memory accesses, nodes could keep a backlog of their pending and active work in their node-local main memory instead of the global one. Nodes will be able to work more efficiently with their faster private memory if they can stay busy for long periods of time without depending on accesses to the slower and inconsistent global memory. This principle is not unlike keeping per-core thread queues at the multi-core level, until faults are added to the game. Vital information such as the existence and evolving state of pending threads cannot reside solely in node-local memories because they are vulnerable to hard-faults. To recover from hard-faults, the system needs to have a way to restore and resume all tasks after a fault even when node memory is lost, preferably with minimal setback, so essential information on the active threads must always be kept fresh in the relative safety of the shared memory. Nodes should then have some independence from the shared memory and avoid excessive operations on it, for the sake of performance, while maintaining fine-grained checkpoints in it to allow minimal penalty on faults and small recovery costs.

This general review might fit the roots of many possible works, for similar or even very different architectures. The solution presented here chose a certain balance between many competing goals such as optimizations for
performance and scalability, storage requirements, granularity, and effort to maintain reliability checkpoints and the cost of recovery from faults.
Design Principles

This chapter describes fundamental design choices that shaped the overall solution and some of the main concepts of the mechanisms created in this work.

Since faults are considered common phenomena in the assumed many-core system (Section 2.3), a design decision in this work is that the responsibility to maintain reliability at the software level lies on the operating system, not on the applications, as mentioned in Section 3.1. Therefore, the proposed operating system is designed to be the last line of defense that protects applications from hardware faults. It aims to efficiently maximize the parallelism of applications while allowing them to remain oblivious to processor faults by taking the responsibility for detection and recovery from hardware errors.

4.1 Application Programming Model

The distributed execution engine in the proposed operating system can execute any legacy code, but the design of the run-time environment was mainly focused on support for parallelization and resiliency for applications that use a certain programming model. The leading programming model chosen in this work is coarse-grained task-parallelism according to dataflow principles. The dataflow model was chosen because it fits very well to the expected horizontally scaled-out structure and limitations of future general-purpose many-core processors. Dataflow programming [56] allows to efficiently overcome scaling difficulties that traditional parallelism techniques have since they require explicit synchronization and stronger guarantees from memory semantics. Complex operations described in a data-flow manner directly reveal the possible parallelization. The implicit synchronization between parallel dataflow tasks is what allows the operating system to overcome weakened guarantees from future memory systems without user intervention, as assumed in Section 2.2.2. Since the model decouples memory accesses from the user application, it allows the underlying system to manage load, hide memory latencies and provide resiliency. Another main reason for choosing dataflow for programming exascale systems is that the model helps to overcome the expected deterioration of reliability in future...
hardware, which was mentioned in Section 2.3.4.1. Isolation is the main tool that helps complex systems to identify and recover from faults, dataflow encourages it by decoupling the software from the memory accesses and by limiting the possible side-effects of tasks, which allows controlling the possible cost incurred on faults. Transactional memory is similar in that regard, as previously mentioned in Section 1.2.2.4. Additional valuable attributes of dataflow tasks were chosen in this work to further assist with answering the goals. Details on the dataflow model applied in this work are discussed on Section 4.1.1.

Specialized framework support is therefore targeted toward distribution and reliable execution of the dataflow portions of application code. This fundamental design decision was key to answering the desired goals.

4.1.1 Dataflow Flavored Execution

The dataflow model helps to separate large operations to small tasks that may execute in parallel according to their dependencies. To maximize parallelism, the execution of complex operations is divided into many tasks. The execution of each task is governed by the dataflow model, i.e., a task can begin execution only when all its required inputs are available. Thus, a complex operation can be expressed as a directed graph of tasks dependencies. This execution model promotes parallelism and reduces synchronization overhead because the graph of dependencies directly exposes the possible parallelization among the tasks. Once started, a task does not need track any external state and therefore no synchronization is needed during its execution.

The new operating system provides an interface that gives users the ability to exploit task-level parallelism through coarse-grained dataflow tasks. The () allows the creation of tasks that are executed according to the dataflow model by the distributed run-time environment presented in this work. A task has the following fundamental properties:

- It is scheduled to run only when all its inputs are ready.
- During execution, all operations with global side-effects are deferred until the task completes.
- When completed in a non-faulty manner, the side-effects are published and results are propagated to dependent tasks.
- If a fault occurred at any point, the operating system automatically restarts the task.
In addition to helping maximize parallelism, the dataflow tasks concept is also leveraged in this work to increase resiliency facing potential hardware faults (Section 2.3). In this work, tasks are made to only publish their output variables after their execution has completed without errors. Thus, faults occurring during task execution cannot contaminate the system with possibly corrupted side-effects. In addition, all side-effects are made to be idempotent, so tasks can be safely restarted even if they suffered a fault during the publication of results.

Another desired (but not required) property is that once it has all inputs available and scheduled to run, a task can run uninterrupted to completion. Continuous execution without blocking running threads helps to minimize the total resources required by the operating system when managing huge amounts of tasks, and it reduces their vulnerability to faults, which is increased once they started running. It will be further explained in Section 5.2 that tasks in this work have different stages in their existence compared to traditional threads. Here, only tasks that have received all their inputs and started execution require a full context like traditional threads. This is very significant when managing huge amounts of threads as assumed for future systems. The proposed operating system does not keep a full thread context for tasks that have not begun to run yet, and so they only consume a small amount of resources compared to threads than began to run but then blocked.

Allowing tasks to run uninterrupted to completion is also beneficial for reliability. Until a task is ready to run, the information required to start it is stored in the relative safety of the shared memory (as will be detailed in Section 5.2), but when scheduled, a more vulnerable node-local thread context is created to support its execution and to contain the side effects until completion. The node-local thread context is analogous to a thread descriptor most operating systems would keep. The longer the thread lives in a node’s private memory without checking back to the shared memory, it is more exposed to faults and there is more information to lose in case a fault occurred.

4.1.2 Implicit Synchronization

The dataflow application programming interface that the proposed OS exposes to applications allows the run-time environment to take over all synchronization responsibilities, and thereby completely relieve the users from them. Therefore, from the user’s perspective, the synchronization of dependent or concurrent tasks is entirely implicit. In other words, applica-
tion programmers do not need to (explicitly) write code that synchronizes between operations of their parallel algorithms. This property should not be mistaken for not having synchronization requirements on user code, it instead means that the synchronization requirements are expressed indirectly by the semantics of the programming model (Section 4.1.1). Expressing the synchronization requirements to the operating system is very valuable in this work because it allows transferring the responsibility of synchronization from the users to the run-time environment.

Please note that although the proposed OS exposes a programming interface that provides implicit synchronization on the user's side, nothing is implicit at the OS layers created in this work. The implementation of the dataflow engine and the related modules use legacy synchronization primitives explicitly. Parallel execution is synchronized using mutexes and counting semaphores that build on hardware-supported atomic processor instructions, and shared memory is synchronized using explicit acquire and release operations (see Section 2.2.2.2).

### 4.1.3 Explicit Parallelization

From the user's perspective, the synchronization of dependent tasks is entirely implicit, as inherent from the data-flow-based programming interface that the proposed OS provides, the parallelization, however, is not. Parallelization of user applications is demanded to be explicit, it means that the dataflow runtime created in this work requires explicit definitions of the dataflow tasks, i.e., their thread binaries a list of expected inputs. If the reader feels this is very limiting, then to put into perspective, please note that these tasks are defined at the level of the user algorithm, they are a relatively high-level programming construct. At the lower levels, e.g., small instruction windows of cores, dataflow principles are expected to be employed by compilers and hardware to resolve dependencies and improve Instruction-level parallelism (ILP), at least as well as they do in today's out-of-order microprocessors.

### 4.1.4 Building Programs

The required specifications for tasks can be supplied directly by the programmer, as will be demonstrated in Section 4.1.5, or by intermediate offline/JIT compilation tools that may assist in the future with tasks dependency analysis, as investigated by [57], [58]. When the programmer supplies the tasks definitions explicitly, as in Listing 4.2, no special compilation tools or
extensions are needed to create a dataflow-enabled application that can run on the presented operating system. Standard compilation tools can be used to build programs for this operating system. A single C header file supplies the interface for the dataflow run-time services. Legacy (non-dataflow) application code can be of any type but dataflow tasks are compiled in a way that prevents the users from using any external dependencies besides the programming interface created in this work.

Although programs can be built with unmodified compilers (GCC was used during the evaluation), the compilation of the dataflow tasks needs special treatment. Once the explicit tasks definitions are specified, the build process compiles the designated code to isolated binaries of dataflow threads. The threads are built in a way that allows the created OS to execute them in an isolated environment, independently from each other and from the rest of the application. During run-time, the threads binaries are transferred over the shared-memory between nodes independently from the main user application. Besides the thread binary, only a small amount of additional information about the specific task is required to execute them, which makes the overhead on work distribution very low. Additional details on the mechanism are described in Section 5.5.3.

4.1.5 Parallelization Example

An example is presented next in order to explain how to explicitly define a dataflow thread to be compiled as detailed on Section 5.5.3 as a part of a task. A recursive definition to calculate the Fibonacci series is shown in Listing 4.1, while an explicit dataflow implementation that the demonstrated system can accept is presented in Listing 4.2.

Listing 4.1: The famous recursive definition for Fibonacci series
```c
uint64_t fib(uint64_t n) {
    if (n <= 1)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}
```

The code in Listing 4.2 is can be used as-is in the implemented operating system. In fact, it is exactly the one used for evaluating and demonstrating the system, more details on it are mentioned in Section 6.4.

The presented implementation is based on a similar variation that was found to scale well in [59]. Note, however, that this is only a minimal example
Listing 4.2: Explicit dataflow version of the Fibonacci series

```c
TASK_ENTRY(fib) {
    uint64_t dest = d_frame(0); // Destination for the result.
    uint64_t n = d_frame(1); // fib's n to calculate.
    if (n <= 1) {
        d_write(dest, n); // Write result to the waiting adder.
    } else {
        uint64_t tfib1 = d_schedule(fib, 2); // Spawn fib1.
        uint64_t tfib2 = d_schedule(fib, 2); // Spawn fib2.
        d_write(tfib1 + 1, n - 1); // fib1 calculates for n-1.
        d_write(tfib2 + 1, n - 2); // fib2 calculates for n-2.
        uint64_t tadder = d_schedule(adder, 3); // Spawn adder.
        d_write(tadder + 0, dest); // adder.out -> self.out.
        d_writev(tfib1 + 0, tadder + 1); // fib1.out -> adder[1].
        d_writev(tfib2 + 0, tadder + 2); // fib2.out -> adder[2].
    }
}

TASK_ENTRY(adder) {
    uint64_t n1 = d_frame(1); // Result from fib1.
    uint64_t n2 = d_frame(2); // Result from fib2.
    uint64_t dest = d_frame(0); // Destination for the sum.
    d_write(dest, n1 + n2);
}

// Equivalent adder with direct access to the full frame:
TASK_ENTRY(adder) {
    struct frame { uint64_t dest, n1, n2; } *fp = (frame*)d_framep();
    d_write(fp->dest, fp->n1 + fp->n2)
}
```

that does not demonstrate all of the supported capabilities of the dataflow threads in the proposed system. The implemented dataflow tasks are not limited to only use the services that the dataflow runtime provides, nor they are bound to the scope of a single routine. Besides being wrapped by the dataflow framework, the threads are not very different from normal user-level threads. They can call sub-routines, use common constant structures or tables and even use libraries, provided they accept the responsibility to stay free of side-effects. The reader is invited to modify and experiment with more advanced dataflow tasks. Instructions how to start using the implementation of the operating system are presented in Chapter 7.

4.2 Partitioned Memory Layout

Figure 4.1 illustrates a logical high-level view of the roles given to the architectural components in the distributed system while running a parallel user application. Please note that the specific system attributes in the figure,
such as cores and memory partitioning are presented only as an illustration. Introduction to the roles of the local and shared memory regions is presented in Section 4.2.1 and Section 4.2.2 respectively.

![Diagram showing memory partitions and their roles across multiple nodes.]

**Figure 4.1:** High-level roles given to the memory partitions

DFx desc. stands for data-flow task descriptor, see Section 5.5.2. DFx binaries are described in Section 5.5.3. Message queues are a part of the inter-node communications mechanism described in Section 5.1.1

### 4.2.1 Local Memory Role

Each node has a private memory region used at its sole discretion, it is only directly accessible from the cores within the node. This local memory region is analogous to the volatile main memory (RAM) of contemporary computers. Since the assumed hardware maintains a strong cache consistency within a node, as current multi-core processors do, this memory can support standard kernels that exist today without modification. The local memory holds the kernel structures for managing the local hardware resources, as most common kernels would, and additional memory space needed by the new distributed run-time services. These additional services use node-local data structures to support reliably executing distributed dataflow tasks, e.g., tracking task inputs, temporary stacks and heaps, buffering results, scheduling queues, multiplexing communication with other nodes, caching dataflow threads binaries, gathering information for load balancing, fault detection and recovery. The shared memory is considered expensive in every way (see Section 3.2), so data that does not need to be shared is kept in the local memory.

### 4.2.2 Shared Memory Organization

In the proposed system, the shared memory (Section 2.2.2) contains data structures that serve two main purposes: inter-node communication and
fault-tolerance. This memory is used in a very different manner compared to the local memories since only a weak consistency model in assumed over it. As mentioned in Section 2.2.2, this work assumes that the shared memory requires explicit fencing operations to propagate changes. Compared to the local memories, users must adhere to additional constraints as described in Section 3.2 to Section 4.4.5.

The organization of the memory segment with affinity to a node is illustrated on Figure 4.2. Section 5.1.1 describes the global layout of the shared memory.

### 4.2.2.1 Partitioned Global Address Space

Shared memory can be an efficient and relatively easy platform to use when parallelizing algorithms, but traditional multi-core implementations have major scalability issues on both hardware and software, as mentioned in Section 2.2.2. A distributed memory management approach was employed at the operating system level in order to avoid traditional scalability issues, it allows to achieve the goals of this work considering the assumed clustered architecture and weakened shared memory consistency.

The operating system proposed in this work creates logical partitions in the shared memory. These declared partitions have considerable significance for software in themselves, but it is possible that the partitions will also be supported by the physical components in future architectures (NUMA, also mentioned in Section 2.2.2).
At the top level of the memory layout, each node has a particular segment of the shared memory assigned to it by the operating system. The shared memory is globally addressable and accessible, which allows any segment to be read and written by all nodes, but the proposed design allows the owner node special privileges over its segment. For example, only the owner node of a shared segment may perform dynamic allocation or deallocation of space in the segment. The shared segment of each node holds the communication channels to the other nodes (Section 5.1.1), and essential information on the tasks assigned to be executed on that node, which is needed for recovery in the case of node failure (Section 5.3).

4.2.2.2 Low-Level Memory Management

The limitations of the assumed shared memory architecture, mainly those mentioned in Section 2.2.2.2 and Section 4.4.4, encouraged the solution to use static allocations for shared structures wherever possible. When dynamic allocations are needed for a temporary shared structure, such as a dataflow TaskDescriptor (Section 5.5.2), only the owner node of the shared segment is allowed to allocate and deallocate memory from it. This constraint helps to minimize vulnerability to faults during critical sections and allows to completely avoid any overhead from synchronizing multiple nodes that attempt to perform memory management operations on the same shared segment. Moreover, depending on traditional multi-core synchronization mechanisms would impair the ability of the global mechanisms to remain wait-free (see Section 4.4.2).

4.3 Fault-Tolerance and Resiliency

The main goal of this work is to provide a self-contained operating system solution for future many-core devices by relying on software mechanisms and existing processor architectures. Since no hardware features for resiliency are assumed in this work, all software mechanisms were made to detect and recover from the assumed faults (Section 2.3).

Having the application code broken up to relatively small tasks helps to limit the effects of intermittent and transient faults, making it possible to contain the problem without the rest of the running application even noticing. Provided the application was parallelized enough, its unaffected tasks can continue in parallel to the fault and recovery process, only suffering a possible delay. The mechanism helps the proposed operating system to conform to the specified goals of this work (Section 3.2).
If the user code uses the dataflow framework (Section 4.1), resiliency for transient-faults is provided at the level of a single thread. On hard-failures however, all of the pending and partial work of the failed cores needs to be recovered and completed by the remaining ones. Therefore, the fault-tolerant run-time mechanisms created in this work handle the two main fault types mentioned in Section 2.3.4.2 and Section 2.3.4.2 very differently. The top-level concepts for fault-tolerance are introduced in this section, further details on the mechanisms are presented in Chapter 5.

4.3.1 Tolerating Hard Faults

Consideration to hard-failures is inherent in all aspects of the proposed operating system and embedded in every one of its modules, see Section 5.1, Section 5.3 and Section 5.2. Fault detection in itself is a rather isolated mechanism in the system but the synergy between all modules in the system was required in order to allow efficient execution and recovery facing hard-faults.

4.3.1.1 Detecting Permanent Faults

In order to detect permanent faults that lead to node failures, a distributed watchdog mechanism was made to continuously monitor the health of the nodes in the system. The watchdog is comprised of software agents running in each of the nodes. Each agent monitors activity signals from its neighbors and alerts the local kernel if a failure is detected in the system. The created mechanism is further described in Section 5.3.4. The design also allows to integrate other hardware features for fault detection, such as those mentioned in [54], however, no such no hardware support is assumed in this work in order to provide a self-contained software solution (see Section 4.4.1).

4.3.1.2 Recovery Process Overview

When a failed node is detected, a node is assigned to take over and rebalance the abandoned work of the failed node. To improve recovery performance, the chosen node should be physically close to the failed one (see Section 2.2.2.3), to benefit from low latency when repeatedly accessing the shared segment during recovery. To simplify the implementation of the decision process, the implementation picks the living node having the shared memory segment with the lowest address, which was not previously
assigned for recovery (to spread the recovery effort on cascaded failures). Since all nodes use the same logic, they do not need to communicate to eventually reach the same decision. Note that in case that unfortunate faults also caused the designated recovery node to fail, the second fault would also be detected by the watchdog (Section 4.3.1.1), and trigger the recovery process for both failed nodes.

The recovery node loads the information about the pending tasks from the shared segment of the failed node and verifies its consistency, details on the process are described in Section 5.3.1 and Section 5.5.2.1. The process then starts a message pump thread to handle the pending messages from the orphaned input queues. The recovery node maintains control over the shared memory affined to the failed node until all recovered tasks are redistributed and all other nodes acknowledge their awareness that the node is no longer available to accept new tasks.

4.3.1.3 Fault Detection and Recovery Scenario

This section illustrates the top-level behavior of the created operating system when a hard-fault is detected and recovered from. As mentioned earlier, in order to maintain resiliency facing hard-faults, all operating system mechanisms presented in this work were built from the ground up to allow automatic recovery of user applications with minimal overhead. The various mechanisms involved are described in detail in Chapter 5. The performance metrics presented here were collected in real-time from the operating system running on a full-system many-core simulator [60], as a part of the evaluation process described in Chapter 6.

The architecture parameters for the experiments presented in this section were chosen in order to emphasize the effect of the fault and the recovery process. The user application is launched on a relatively small architecture of 128 cores with 4 nodes and 32 cores per node in order to allow each node a significant contribution to the total throughput. The chosen user computation is short, and the fault detection latency was adjusted to have the different states of the execution clearly visible in the graphs.

Figure 4.3 shows real-time performance measurements during an execution with an injected hard-fault after roughly 40% of the computation was performed. Marked on the graph are the following key points:

1. A hard-fault is injected into one of the nodes, its cores instantly halt all code execution and its local memory vanishes. These experimental results are from a 4-node simulation, so the failure immediately results in 25% drop in overall throughput.
2. Following the fault, the remaining nodes still have plenty of work to do that does not depend on results from the abandoned tasks of the failed node. Although the amount of ready tasks quickly dwindles, the nodes can continue at full utilization and throughput stays relatively flat.

3. After a while, the nodes complete all the tasks that were independent of results from the failed node. Throughput begins to fall until it reaches zero when there are no ready tasks left to execute. The remaining tasks may not begin execution until all their required inputs are ready (see Section 4.1.1), but those are outputs of tasks that belonged to the faulty node.

4. The distributed watchdog decides that too many heartbeats were missed from the faulty node (see Section 5.3.4), so its agents alert the remaining nodes to initiate the recovery process. A node is selected to recover and rebalance the work from the failed node (the bottom node in the graphs). Throughput begins to rise as the recovered tasks are redistributed and executed, which also allows previously blocked tasks to become ready.

5. Throughput approaches 75% of that before the fault and the computation continues with the remaining cores at full utilization. The throughput of the recovery node is slightly lowered due to the recovery process, which continued until the end of the computation in this short experiment.

6. The computation is close to completion and the number of tasks that become ready for execution is not enough to sustain full utilization, throughput falls until all work is complete.

As mentioned, the parameters in the first example were chosen specifically to demonstrate the recovery process. The delay from the fault occurring to its detection can be controlled by parameters of the watchdog (see Section 5.3.4). Regardless, hardware mechanisms that can detect core faults can also be used on a future system, as investigated by [54]. Hardware support could reduce the response time to hard-faults without increasing the overhead of the watchdog.

The second experiment presented in this section illustrates more typical behavior on faulty cases, where utilization does not fall before the fault is detected and the recovery starts. The results shown in Figure 4.4 are from an experiment similar to the previous one, except that the duration of tasks was doubled. The main features of the behavior in this experiment are similar to those from larger experiments, as presented in Section 6.4.2. The
number of ready tasks does not reach zero before the abandoned work is recovered from the failed node, it can be seen that throughput is not affected until the fault is detected. Once the hard-failure is detected, the throughput of the recovery node (bottom one in the graphs) slightly decreases due to the recovery effort.

Results from additional experiments on faulty hardware are presented in Section 6.4.2, and a demonstration of such experiment is shown in Section 7.3.0.0A.

4.3.2 Tolerating Transient Faults

For the detection of transient and intermittent faults on dataflow applications, the presented operating system can wrap user tasks with a Double Execution (DE) layer. Algorithms that use redundant execution and results comparison for fault detection and tolerance are wide-spread [32], [61], [62]. The use of Double Execution was found suitable to provide fault-tolerance for dataflow models because its application within a dataflow execution environment can be simple and light-weight, considering the isolated nature of dataflow tasks [15], [63]. Compared to the mentioned works that use architectural components, the suggested solution in this work is not based on an invented hardware and does not require data replication. The use of MCA mechanisms
Figure 4.4: On-the-fly hard-fault recovery - 128 cores

4 nodes, 32 cores per node, 86k total tasks, 1.6 ms average task duration. Recovery has started before the utilization of the remaining nodes was affected. Throughput then slightly decreases due to the extra effort during the recovery process.

[54] is suggested both in previous works and in this work to assist fault detection and recovery, but it is not required in this work to treat transient faults.

The method used allows the system to efficiently detect and recover from soft errors at the node level without disturbing the global system. Furthermore, taking advantage of the created mechanism does not require any user involvement, as specified by the goals for this work. Transient faults on user code are detected and contained at the scale of a single task, which minimizes their effect and allows quick response and low cost of recovery.

In this execution mode, two independent threads are created to run a task when it becomes ready for execution. Both threads share the same inputs and code and are therefore expected to produce the same outputs, unless a fault occurred. When both threads complete, their outputs are compared and the side-effects of the task are either committed to the global state if the results match or the task is restarted if there is a mismatch.

This redundant execution mechanism is further described in Section 5.3.5, and it is demonstrated in an experiment presented in Section 7.3.0.0B.
4.3.3 Low-Cost Fine-Grained Checkpoints

Capability for recovery is maintained at the level of a single task. This means that if a hard-fault occurred, only the actively running task on the failed cores will need to be re-executed. This constraint limits the penalty that running applications might suffer, and the size of each checkpoint. The solution does not rely on additional persistent storage and it does not require any custom user-level assistance. These guidelines are crucial to achieving the goals of the work mentioned in Section 3.1 and Section 3.2.

Achieving a very low overhead for the declared reliability goals (Section 3.1) is an unrealistic expectation with traditional methods if no user involvement is assumed and a general application programming model is used [5]. To provide reliable service with low overhead, the mechanisms in this work rely on the dataflow application execution model that allows decoupling applications from memory accesses, task synchronization issues and faults. The reliability considerations are an inherent part in all levels of the global and local execution dataflow engine of the operating system, they provide fault-tolerance capability with low overhead on non-faulty cases and fast on-the-fly recovery when faults occur.

4.4 Additional Principles

4.4.1 Software-only approach

While other approaches for run-time environments and fault-tolerance solutions on future many-core system reply on inventing new hardware mechanisms [64], the proposed solution is implemented entirely in software. This approach increases flexibility and generalizes the solution to only depend on a high-level set of assumptions and requirements from future many-core architectures, those described in Section 2.1 and Section 2.2.

Independence of hardware support also applies to the fault-tolerance aspects of the solution; no hardware-assisted fault detection and recovery mechanisms are required by the OS. This constraint applies both at a node-local level and at the global level. Software-only mechanisms are made to detect and recover from both transient and permanent faults. That being said, the modular design is ready to accommodate hardware features for fault detection and tolerance that can improve resiliency and efficiency. Examples of such hardware features can be watchdog timers that are extensively used in most embedded systems and MCA that exists on modern processors [45],
Additional architectural features for reliability that are focused toward future many-core processors are investigated in [15].

### 4.4.2 Everything is Wait-Free

To improve performance, the new operating system is designed so that all data structures that reside on shared memory and all inter-node operations are wait-free. This means that any single operation on a shared resource or interaction between nodes is never blocked by another. All operations a node performs on shared structures are non-blocking and can be completed without synchronizing access with other nodes. Not a single synchronization lock or atomic conditional primitives such as read-modify-write instructions is used (see Section 4.4.4), nor higher-level mutual exclusion mechanisms or hardware supported workarounds, such as transactional memory [35].

Note, however, that this design choice does not preclude using data structures that reside on the private node memories (Section 2.2.1), since the involved algorithms only require synchronization within a single node, between only a few cores. The architecture of individual nodes is assumed to use commodity components, which allows node kernels to use traditional programming models that may rely on strong memory consistency guarantees on the local memory.

### 4.4.3 Software-Managed Consistency

Traditional operating systems rely on hardware-implemented cache-coherency protocols to provide system-wide consistency of the shared memory. Since these hardware mechanisms prohibit significant scaling, such capabilities are not assumed in the architectural model described in Section 2.2.2.2. This approach is supported by several other works that address the scalability issues of shared memory systems [22], [23]. Therefore, the presented operating system does not rely on the hardware to provide consistency guarantees on shared memory accesses, and implements it in software using new machine instructions that trigger the consistency. Section 6.1.2 describes how the Instruction Set Architecture (ISA) extensions were implemented in the simulator.

Therefore, all run-time mechanisms created in this work explicitly manage the global consistency of their shared data, considering that the coherent domains within nodes are otherwise isolated.
**Decoupling memory from applications** Although the implementation of the shared objects and communication mechanisms in the operating system trigger consistency explicitly by issuing acquire and release instructions, the applications are not expected to do so. The run-time environment created in this work decouples the application software from memory accesses by abstracting the architectural details and the memory semantics. The new environment exposes a dataflow programming interface to users, which enables it to do so, as described in Section 4.1.

### 4.4.4 No Atomic Conditionals

Considering that the hardware does not maintain consistency over shared memory across nodes, and software must issue explicit instructions to propagate data on it (acquire/release instructions, see Section 2.2.2.2), this work further weakens the hardware requirements such that no atomic conditional primitives such as compare-and-swap are assumed from the microarchitecture. This constraint means that conventional mutual exclusion mechanisms that are used by modern operating systems and applications to synchronize concurrent accesses to data between parallel threads, like spin-locks and semaphores, cannot be placed in the shared memory.

Note, however, that this constraint does not preclude using atomic primitives on node-local memory locations, since traditional cache-coherency is assumed on the private memory within nodes, as mentioned in Section 2.2.1.

### 4.4.5 Completely Decentralized

All operating system mechanisms are made to be completely decentralized across nodes. There is no master node or any centralized services or data structures. The solution avoids dependency on any centralized facility in order to prevent obvious performance bottlenecks that inhibit scalability, and major vulnerabilities to faults from having a single point of failure. Design considerations for performance issues that are usually addressed on small-scale concurrent systems are included in addition to those that emerge in many-core systems, relating to NUMA, bus contention, synchronization mechanisms and more.
4.4.6 Easily Fit Existing Systems

The assumed node architecture is based on contemporary multi-core x86-64 processors (Section 2.1), so it is reasonable to expect that existing user applications should only require minimal or no adjustments in order to be executed in the new environment. More specifically:

• The operating system is able to run existing (legacy) code without modifications. If the new OS services are not employed, parallelization of unmodified code may be limited to the bounds of a single node.

• Unmodified system libraries (e.g., libc) may be used on nodes running a full (e.g., Linux) kernel. System libraries support may be limited or require adjustments on some nodes, depending on their architecture. For example, nodes targeted mainly toward computational tasks, as illustrated on Figure 2.1, may have limited I/O support.

• Compiled libraries and executables may run without modification. As mentioned in a previous point, some nodes may run a kernel compatible with existing multi-core systems but support may be limited on other types of the nodes.

Note, however, that in order to fully utilize the parallelism capabilities in the system, beyond the multiple cores of a single node, an existing parallel application is expected to explicitly request the presented operating to do so. This is later described in detail on Section 4.1.3.
5.1 Inter-node Communication

This section describes custom data structures and methods that were created in this work to allow fault-tolerant inter-node communication over the assumed shared-memory (Section 2.2). The hybrid solution comprises a combination of message-passing and shared state between nodes to provide a fused environment in the proposed operating system.

Considering the assumptions detailed in Section 2.2.2, standard (library) data structures could not simply be placed in the shared memory to share information between nodes because it is assumed to be weakly consistent between nodes and require explicit acquire/release operations to synchronize access to it. The new structures that are detailed on Section 5.1.1 and Section 5.1.2 consider both the assumed consistency models in the memory hierarchy and possible faults, as mentioned in Section 2.3. The new mechanisms embrace the acquire-release semantics to optimize parallel performance by allowing completely wait-free accesses to all shared data. In addition, the required shared-memory synchronization operations, i.e., acquire and release, are used in ways that gracefully fail in any event of a hard-fault; i.e., the main objective is to leave all data structures in a consistent state without requiring any roll-back operations (see Section 3.2).

5.1.1 Block Transport Layer

This software layer forms the backplane on shared-memory for unicast communication between nodes, it is used to pass fixed-size blocks between nodes and provides a wait-free interface. This is a low-level layer of the OS, it is used as a transport layer for the messaging layers that will be described in Section 5.1.2. The global mechanism this layer represents is actually a composite of independent data structures, called channels throughout this thesis. Each internal element is a first in, first out (FIFO) queue for raw data, that was permanently designated to be a unidirectional link between a single pair of nodes. The combination of all the data queues forms a complete graph of communication channels between the nodes. Since each channel is designated to only be written by a single node and read by a
single node, i.e., **Single producer single consumer (SPSC)**, it was possible to make the transport layer implementation very efficient and *wait-free* using the assumed shared-memory semantics (*Section 2.2.2*). *Figure 5.1* shows how the operating system partitions the shared memory.

![Figure 5.1: Shared memory layout](image)

Where the buffer noted \( x \to y \) contains the queue of messages routed from node \( x \) to node \( y \). If \( x = y \) then the buffer is referred to as *loopback*, which is a special case because it is both read and written by the same node.

Arena \( x \) is a continuous region of shared-memory, each node may only dynamically allocate/de-allocate from its own arena, *Section 4.2.2* explains why.

### 5.1.1.1 Behavior on Faults

Each channel was designed to inherently survive a hard failure happening during any single operation *and* during sequences of operations that may span many consecutive blocks, without any effort for recovery. Surviving a failure, in this context, is that evidence of the incomplete sequence, whether it was adding or removing data, will be *gracefully* lost. Incomplete sequences cannot corrupt the channel structure nor data from other sequences on the channel (either previous or concurrent) and they do not leave behind any partially added or removed data. This is true by design, regardless of failures that may happen at the other end of the channel. Seems like the most that could be hoped for, basic transactional memory for almost free. The mechanism that delivers this guarantee is described in *Section 5.1.1.5*.

### 5.1.1.2 Poor Man’s Transactions

Automatically undoing invalid or incomplete sequences provides the users of channels a trait that similarly exists for transactional memory operations in case of conflict (*Section 1.2.2.4*). Protective coverage for sequences of operations is possible although the transport channels have no notion of sequences themselves, and do not assume any structure on the underlying data. Instead, they expect the user (the message-passing layer, in this case, see *Section 5.1.2*) to provide segmentation for groups of operations. The messaging layer utilizes such a protected sequence to guard against a failure case when reading or writing one or more messages that span several transport blocks.
Sequences of operations on a channel are inherently undone if they were not completed because the mechanism does not publish them to the global system until the last operation completed. The mechanism relies on the assumed consistency model of the shared memory (Section 2.2.2.2), that requires explicit software instructions to propagate writes on the shared memory. The runtime does not release changes on channels until the whole sequence of messaging operations and their surrounding processing is completed by the upper layer.

5.1.1.3 Extending Transactions

Each protected segment may also cover the user's post-processing of the sequence. A common usage for this is when a reader wishes to pull data from a channel, then process and check in the information it got somewhere safe (e.g., somewhere in the shared memory), and doing so without being vulnerable to failures causing the data to be lost after already being removed from the channel. Virtually extending the sequence is also available on the dual example for a writer so it can extend a sequence of messaging operations to cover its post-processing which allows performing additional operations safely on other channels.

5.1.1.4 Transport Backplane

At the beginning of the shared region of each node, a set of buffers are statically allocated to store inter-node communication data. Those buffers hold data structures called channels throughout this thesis, they contain messages received from every other node, and from the node itself. A single buffer is allocated for each node, each buffer contains a cyclic FIFO queue of small memory blocks of fixed size. The optimal size of a block can be chosen to fit the most common message sizes, currently 32 bytes. Each queue is the channel dedicated for unidirectional communication between two nodes (SPSC), so data can only be added by a single remote node and removed by the local node. Since each node has one input buffer for every other node and one loop-back buffer to itself, there are a total of \( n^2 \) such buffers in the shared memory. Their combination creates a complete graph of non-blocking bi-directional communication channels among all nodes.

5.1.1.5 Channel Operation

The implementation of each channel was optimized toward minimizing the number of acquire and release operations on shared memory while meet-
ing the fault-tolerance and usability requirements mentioned in the previous sections. Only a single acquire or release operation of one machine word is performed per access sequence (see Section 5.1.1.2), in addition to those that read or write the user’s payload. This efficient mechanism leverages the weak shared memory semantics (Section 2.2.2.2) by using a private (low-latency) cache of a partial state of the channel from shared memory. The channel allows the user to perform a series of read or write operations, including their required acquire or release for the payload, while only updating the metadata of the channel in the local cache, effectively keeping the changes unobservable by the global system. Intermediate operations on the channel issue acquire instructions to detect an overflow or underflow while operating on the private cache of the state. The metadata of the channel is only published globally (a single release instruction) when the user has completed the sequence of operations in a non-faulty manner.

5.1.2 Message-Passing Layer

Messages are packetized to fixed size blocks and sent over the inter-node transport layer (Section 5.1.1) to their destination nodes. Each node frequently polls on the messaging queues from the other nodes and processes the received messages. Section 5.1.3 later describes how nodes reliably process messages from the queues.

The following paragraphs describe several primary types of messages that are used by the distributed run-time environment. Please note that additional types of messages are exchanged between nodes but the following three are the main ones that support the dataflow execution engine.

5.1.2.1 TaskLoad

These messages bundle all information required to construct a task. It contains the thread binary to execute (Section 5.5.3) or its identifier, the latest inputs information and the user requirements for shared regions. This message becomes a TaskDescriptor that the receiver will keep in shared memory, bit for bit. The structure of the message is therefore identical to a TaskDescriptor which is described in Section 5.5.2 and illustrated on Figure 5.4. The procedure performed when receiving this message is described in Section 5.1.3.2.
### 5.1.2.2 TaskWrite

Contains the destination address (combination of the Unique Frame Identifier (UFI) and input offset) and the value to write. The steps taken when processing this message are described in Section 5.1.3.1. Bit 8 is reserved with a value of 0, Section 5.2.2.3 explains how it is used when this message is created.

![TaskWrite message structure](image)

**Figure 5.2:** TaskWrite message structure

### 5.1.2.3 Heartbeat

Notifying that the source node is alive and contains state information used for load balancing, such as workload, temperature, fault rate and other metrics. The purpose of this message is further described in Section 5.4.3.

### 5.1.3 (Safe) Messages Processing

Considering that permanent faults may cause the private memories of nodes to become inaccessible, as mentioned in Section 2.3.4.1, the proposed operating system keeps inter-node messages in the safety of the shared memory until their processing is complete. The shared memory is assumed to function regardless of node failures, so the messaging channels in it were designed to maintain a consistent state of the structures even if a hard-fault occurred during an operation, or their affiliated node suffered a hard failure. Continuous sharing of the messaging queues allows to recover the pending messages of a failed node and handle them on a different node (see Section 4.3.1.2). Regardless of the specific type of message, the following methods are generally used to enable recovery from hard-faults when processing received messages:

- When a node processes a received message, it does not remove it from the input queue until it completed performing the required operation. If a hard-fault occurred during processing, a different node could still process the message.
- The processing of messages that are critical for the correct operation of the application, such as TaskLoad and TaskWrite, involves updating the shared memory region of the node by checking-in the new information before disposing of the message. The process ensures that no
critical information will be lost along with the local memory of the node in case of a fault. Further details on how this is performed for several messages is described in Section 5.1.3.2 and Section 5.1.3.1.

- The above processing is done in such a way that a core failure occurring at any point in time will always keep the information recoverable from the shared memory, either from the messaging queues or the relevant TaskDescriptor, as described in Section 5.3.1.

The combination of these steps allows the recovery of all messages that were pending in the failed node, and of the message that was being processed at the time the node failed. Special care must be taken when processing the first recovered message, as it might have already committed its full or partial results to the backup store before it was removed from the message queue. The backup region and all objects in it (Section 5.3.1) were designed to overcome problems resulting from incomplete operations and avoid corruption if operations are repeated during recovery.

5.1.3.1 Processing Task Inputs

Writing inputs to waiting tasks frames with TaskWrite messages must also involve an immediate update of the TaskDescriptor (Section 5.5.2) in shared memory so information is not lost in a case of a node crash. In addition to keeping the written input values, the TaskDescriptor also keeps track of the locations of writes made to it. The state is updated for each written input and marks when all of the inputs are ready and the thread can be executed. As always, the message processing assumes a hard-fault can occur at any point, so the process first writes and publishes the new data to the thread frame in shared memory, and only after doing so it sets the corresponding bit in the TaskDescriptor that indicates that the specific location was written. Keeping track of individual inputs is needed since duplicated messages may be received after a fault, more details regarding this safeguard are in Section 5.3.3.

5.1.3.2 Processing Scheduling Messages

Processing of TaskLoad messages involves modifications to several shared structures which are done in several steps, this makes the message more complex to safely process than other types of messages. Safely processing this message means, as always, to carefully account for the possibility of a hard failure happening at any point in time, on any instruction before or even during the processing of the message. The procedure must never leave
the involved shared structures in an inconsistent state even in case of faults, and specifically, it must not lose or corrupt information that is vital to the correct execution of the task.

The procedure described below ensures that the latest state of the new task can always be recovered from either the newly created TaskDescriptor (Section 5.5.2) or the pending TaskLoad message, no matter at which point a node has failed. Moreover, the procedure considers that if interrupted, it can be safely restarted. The steps involved in processing a scheduling request are:

1. Verify that this is a new task, ignore the message if it isn't, see Section 5.3.2.
2. Allocate space for the TaskDescriptor (see Figure 5.4) in the shared memory of the node and initialize it using the information from the scheduling request.
3. Add an entry in the assigned tasks collection of the node, which is also stored in the shared memory (see Section 5.3.1). The entry in the collection links the UFI of the new task to the memory location of its TaskDescriptor. Modifications of the structure consider faults and aim to keep the collection consistent even if an ongoing operation has failed. If the structure became corrupted on a fault, it can be rebuilt with no information loss by scanning the entire memory region to gather valid TaskDescriptor objects, see Section 5.3.1.1.
4. Publish (release) the new descriptor and map entry. From this point, the TaskLoad message can be safely removed from the messaging queue.
5. Add a pointer to the task descriptor into the list of pending tasks in the local memory of the node. The task will remain there until all its inputs become available, Section 5.2.1 describes the execution process.

**Single-Copy Scheduling** Depending on the chosen assumptions for shared-memory access latencies (Section 2.2.2.3), some modifications on the process described in Section 5.1.3.2 that handles scheduling requests may be preferable. The mentioned process transfers the information on a new task in TaskLoad messages to the executing node over the shared memory, which is only a good idea under some sets of assumptions. The main assumption that led to this design is that the memory access times of a node to its affined shared memory region are significantly better than to remote shared regions, so much so that it is worthwhile to copy information to
a faster region before the task is ready to be executed. This tactic was designed to mitigate longer shared memory latencies when starting a task by bringing the data nearby before it is needed. However, under different assumptions, the nearby copy may not be beneficial enough to justify the extra copy, if nodes can efficiently execute code located on shared-memory with affinity to remote nodes. In such case, the node that created the task can keep the TaskDescriptor in its own shared region, and the TaskLoad message merely contains the address of the TaskDescriptor, i.e., a single machine word. If applicable, this method avoids redundant data movements on shared-memory and reduce the load on the message queues because TaskLoad would be much smaller, but there is an additional cost. This mode requires an additional message (simply named TaskCompleted) sent to the originator of the task to inform it when the task was done executing so it can free the shared TaskDescriptor. The extra message is needed because nodes are not allowed to perform memory allocations or de-allocations on remote regions, as mentioned in Section 4.2.2.

This work assumes that physical distance to the shared location has a very significant factor in access latency (Section 2.2.2.3). It was therefore decided to place task descriptors on the shared region of the node that was assigned to execute them. There is an exception to this policy, however, where the variation described in the procedure is used and the TaskDescriptor is not copied to the executing node but only its address. The situation occurs when rebalancing recovered tasks from a failed node, as described in Section 5.3.1. In this situation, some scheduling steps were already done and the TaskDescriptor is already stored in the shared memory, but within the abandoned shared region of a failed node. In such case when the task is rebalanced to a node other than the one recovered it, only the address of the recovered TaskDescriptor is sent to the new node. This mechanism avoids the cumbersome transfer of the actual TaskDescriptor from the recovered region to the newly assigned node. Once the rebalanced task is executed, a TaskCompleted message is sent to the recovery scheduler (see Section 4.3.1.3) so it can deallocate the memory.

5.2 Task Execution Procedure

5.2.1 Life of a Task

The process of tasks execution can be described in the following high-level stages:
• A scheduling request is created, specifying thread code by name or by attaching the actual binary to execute, the number inputs to wait for and the regions of global memory it needs to access.

• The request is submitted to the local scheduler, that decides which node should be assigned to execute the thread based on load-balancing and performance considerations, see Section 5.4 for more details. The scheduler assigns a system-wide unique identifier (UFI) to the new task.

• The scheduling request (Section 5.1.2.1) is sent to the assigned node (can be loopback to the node same node), as a message over the shared-memory communication channels (Section 5.1.1).

• The target node begins to process the request by first copying the thread information to a TaskDescriptor (Section 5.5.2) in its backup store in shared-memory, before removing the message from the queue, as described in Section 5.1.3.2. After doing so, the task is protected from being lost in case of a failure during its processing.

• The node creates a local realization of the task in its local memory and adds it to the list of pending tasks. Then, it will wait there until all its inputs are received by TaskWrite messages.

• When a TaskWrite message is processed, the destination TaskDescriptor in the shared memory is first updated with the new information. This, again, is done before the message is removed from the queue, so the new data is not lost even if the node crashed after a thread received some of its inputs but not yet executed. See more details in Section 5.1.3.1.

• When the thread received all of its inputs it is ready for execution. At this point, another scheduling decision can be made to forward the thread to be executed on another node. The time passed since the creation of the thread could affect the decision on which is the best node to run on. The ownership is easily transferred when the thread has all of its inputs, because no further TaskWrite messages (Section 5.1.2.2) are expected.

• Once all inputs are ready, the shared user regions that the task needs to read are acquired to local memory and the task begins executing.

• During execution, the side-effects of the task are buffered in local memory because they need to be deferred to the completion time, see Section 5.2.2.

• Once the task completed executing, it is added to the list of completed ones in local memory, pending its finalization process that ultimately
commits the side-effects to the global system. The task finalization procedure is further detailed on Section 5.2.3.

5.2.2 Deferring Side-Effects

All operations a thread performs that have side-effects are not immediately committed to any shared resource, they are buffered in local memory until the thread completes. If Double Execution is enabled for the task, it is launched twice on different cores of the same node, and each thread’s output is compared to the one from the sibling thread. If both executions complete with matching results, their outputs can be committed system-wide, if there is a mismatch in outputs then the thread is re-launched.

The side-effects buffer contains a trace of three types of operations that a thread may use to output its results. Descriptions of each of the buffered operations are in the following sections.

5.2.2.1 Deferring Memory Write Operations

Let us call speculative results those output values of a thread from the time they were generated to the time that the thread has successfully completed its execution. Those results should be stored and not be committed to the globally observable state before the thread was completed, and possibly also passed the Double Execution test. These writes are therefore not published until the execution succeeded. Depending on the application code, publishing these speculative results might not damage the user logic even if the thread then failed before completion, because the writes must not trigger a remote operation directly, by polling for example, until an appropriate TaskWrite will notify about it, which will only come after successful finalization of the thread.

5.2.2.2 Deferring Scheduling Requests

Submitted scheduling requests need to immediately return an identifier to the new task, which can be used as the target of subsequent TaskWrite operations or even their value. The task may use the returned value to allow the child tasks to communicate directly among themselves. Since it is still unknown if the running task will complete successfully, and will pass the Double Execution test, the scheduling requests are not published to the rest of the system until their parent task completed. It seems reasonable to delay, considering the difficulty to hunt down wandering orphaned tasks.
that will be left in the global system. Those would be forever missing their inputs if because their parent was re-launched after a fault and created other tasks. It is important that in addition to delaying the distribution of new scheduling requests, real UFIs are not even generated for them yet, as generating one leaves side-effects, since it involves invoking the scheduler (and its load-balancer) to create and assign a unique identifier and owner node. Immediate scheduling would also require to mirror or share the information between two sibling threads if Double Execution is used, which requires additional synchronization overhead.

5.2.2.2A Virtual Frame Pointers  When a new scheduling request is made by a task, an actual scheduling message cannot be sent immediately, as mentioned earlier, so a temporary task identifier is returned to the caller and the request is buffered. The returned value is a Virtual Frame Pointer (VFP), it is not yet the global identifier that is described in Section 5.5.1. The VFP is only valid in the context of the creating task, it could not be used system-wide. However, the creator can still use this virtual identifier in its following operations just like it would use any UFI it received as input, e.g. writes can be performed to it and other tasks it creates can depend on its outputs.

Using virtual identifiers also eases Double Execution because both siblings are simply made to produce identical identifiers, removing the need for any synchronization mechanisms. Another advantage when using VFPs is that with the wait until all scheduling requests from the task are generated, a more informed decision can be made on where the child threads should be placed, i.e., which node should run them.

After creating a task, the parent is likely to quickly write some input values to it. So even without sophisticated analysis mechanisms, the system can exploit the temporal and spatial locality of write operations to preceding scheduling operations. Keeping the scheduling request buffered in local memory for a short time without publishing allows to immediately merge following write operations to the new thread frame. The merges will be made to the fast local memory of the node without ever going through the message queues in the global memory, thus saving global communications costs. Note that the gain applies also if the following writes are sent to the loopback queue, because it is also stored in shared space for reliability. The write operation will be made almost effortlessly compared to the case that has the scheduling request is immediately published and writes to it are published soon thereafter.
The cost of using these virtual identifiers might not be negligible, however, because once UFIs are actually allocated for the requests, all of the data the thread has created needs to be reviewed and references to the virtual identifiers need to be replaced with the global UFIs, see Section 5.2.3 for details on this step.

### 5.2.2.3 Deferring Results Publication

Although sending duplicated writes will not damage the execution integrity (see Section 5.3.3), write to threads still cannot always be performed immediately for several reasons. When (optionally) using Double Execution, the value needs to be first verified against the sibling thread before considered verified and can be published. More importantly, these writes need to be buffered because the target thread that is written to and/or the value, may be the result of a previous scheduling request the thread has made, and are therefore only temporary VFPs when the write is made. The writes cannot be published before the temporary thread identifiers are replaced with their final values, which only happens when the thread is finalized and the scheduler has assigned global identifiers for them.

When buffering a write, bit 8 in the message structure illustrated on Figure 5.2 is reserved to specify if the value of the write is actually a VFP (Section 5.2.2.2A). Writes that contain a VFP in the destination or value will need translation to final UFIs when finalizing the thread (Section 5.2.3). When finalizing the thread, checking if the destination of a write is a VFP and not a UFI is easy since their structure is different, but an additional hint is needed to detect if the written value is a VFP. Since the values can be any data, the finalization process has no way of knowing if a certain value is a VFP that needs translation. To overcome the problem, a bit is reserved in the TaskWrite message to notify the scheduler that the value of the write needs translation when finalized.

There are quite a few simple tweaks that can immediately improve overall performance regarding the containment of side effects, most apparent ones are for the Double Execution case since only a simplistic implementation was made. For example, some writes to inputs of dependent threads can be published system-wide immediately on creation if:

- The write doesn't use a VFP, which means that it will not be modified during the thread's finalization process. Remember there is no harm in re-sending it later if there was a fault.
• The write is trusted, meaning that if Double Execution (DE) is used, the operation was already verified by the sibling thread or Double Execution is not used and every write is immediately trusted.

5.2.3 Task Finalization

Dataflow tasks have no global side effects during their execution but eventually, their outputs, i.e., memory writes and new tasks they created, do need to be published system-wide. The task finalization process is not an atomic operation, the operating system considers that the global state must remain consistent even if every stage in the process may fail part-way when a node suffers a hard-fault.

When the task execution completes, after passing the Double Execution test if it is enabled, the task is put in a list of completed task waiting to be finalized. There is some work to the finalization process since every side-effect that the task has made was deferred and stored in the local memory of the node (Section 5.2.2). The finalization process prepares the information the task has generated so it can be published to the rest of the system.

The deferred operations performed by the task are processed and committed from the local buffer in this order:

• Requests for direct writes to shared user regions are published system-wide as-is. This stage simply performs release operations (see Section 2.2.2.2).

• The local scheduler is invoked for each of the scheduling requests the task performed to generate valid UFIs that will replace the temporary VFPs the task was handed to during its execution. The scheduler chooses a destination node to run each task while considering performance optimizations and load-balancing (see Section 5.4). It then generates a UFI for the new task and sends the request to the destination node, possibly local loop-back. For details on the receiver side see Section 5.1.3.2.

• The buffered write requests are reviewed in order to map usages of temporary task identifiers (VFPs) to final addresses in UFIs generated in the previous step. The temporary task identifiers may either be a part of the the destination field of the write request or the value field. After the translation was made, the final requests can be sent to their destination nodes. For details how this was made fault tolerant see Section 5.3.3.
Section 5.3.1, Section 5.3.2 and Section 5.3.2 further describe the methods that allow safe tasks finalization even in the face of faults.

5.3 Fault-Tolerant Methods

This section describes the main principles and mechanisms that were developed in this work in order to achieve the global fault-tolerance as described in Section 3.1 while still doing the best for keeping the performance goals mentioned in Section 3.2. Considering that fault-tolerance is an inherent part of all modules in the presented operating system, most of the mechanisms that allow achieving the fault-tolerance methods described in this section were already mentioned in previous sections that describe the run-time mechanisms.

Example Fault Conditions  This paragraph illustrates some concrete contexts for the fault-tolerance mechanisms presented in the following sections. As faults can occur at any point in time (Section 2.3.4.1, Section 2.3.4.2), the system needs to be designed in order to recover from any single point of failure. Therefore, correct operation of the distributed system and all running applications should continue when a node responsible for some user task has failed, even when ¹:

- The task may have already written some results to its dependent tasks, those writes will be repeated when the task is recovered and re-executed after a failure.
- The node was holding the binary code of the task it was executing in its volatile private memory, which is assumed inaccessible after a hard-fault (Section 2.3.4.1). However, it is required not to lose user code running from memory even if a node crashed, since it is not assumed that the user would be able to provide it again.
- The thread might have already performed scheduling requests and even wrote data to the threads it created.
- The node might have died before it read the contents of the scheduling command, while the originator of the command may have long forgotten about it.
- During the initialization of the support structures of the thread in global shared memory.

¹The example fault conditions use concepts specific to this work, some of the concepts will be defined in subsequent sections in Chapter 5, most notably in Section 5.2.
• During the waiting time for the inputs of the thread, but after it already received some of them.

5.3.1 Recoverable State

Each node keeps essential information about the latest state of its tasks in shared memory, so in a case that the node failed, its assigned work can be recovered and executed by the remaining nodes.

A collection of TaskDescriptor (Section 5.5.2) objects for pending and running tasks a node is assigned to execute are stored in the shared memory, each descriptor contains the task identifier (UFI), binary to execute, inputs state etc., everything needed to describe the task as illustrated in Section 5.5.2. This collection serves as the backup in case the node failed and allows the recovery of all the task that the node was responsible for. A TaskDescriptor also keeps all the input values that were written to the task so they are not lost on faults. Each TaskWrite message updates the destination descriptor with the new information until all inputs are received (Section 5.1.3.1). This backup is simply a copy of the TaskLoad messages the node has received but did not yet complete. As mentioned regarding messages processing in Section 5.1.3, while processing a TaskLoad or TaskWrite messages, the TaskDescriptor in shared memory is updated first, before removing the messages from the channel (Section 5.1.1), so critical information is never lost if the node suddenly stops at any point.

The recovery process for abandoned work processes the recovered TaskDescriptor objects from the shared backup, as if they were just received from another node. The recovery process can also choose to forward the recovered tasks to other nodes, completely as-is, but can do so only if all their inputs were received already, see Section 5.4.2 regarding the specifics. Note that if the recovery node chose to forward a task elsewhere, this task will already have all its inputs available and is ready for execution, thus it is not expected that there will be any pending messages regarding it that the destination node could not handle. All of the messages that are still waiting in the queues of the failed node are processed by the recovery node as if it was the original node. Note that the first message in each queue is processed just like any freshly received message, even considering that the failure may have happened after already partially handling it.

These mechanisms ensure that the latest state of all tasks can always be recovered from a failed node, combining the information from the shared TaskDescriptor collection described in this section and the pending messages channel, no matter in which point a node has failed.
5.3.1.1 Verifying Shared State

When starting the recovery process, the shared collection of TaskDescriptor is checked for consistency. The check verifies that the metadata of the collection is in a consistent state and that the structure of all TaskDescriptor objects in the collection are valid (see Section 5.5.2). If the metadata of the collection is corrupted, the recovery process scans the entire shared segment looking for TaskDescriptor structures in the raw data and reconstructs the collection. The TaskDescriptor structure has a special field with a value that is unlikely to appear in memory by chance, it is used when scanning the memory to find candidates for recovery. When the field is found, a checksum verifies the validity of the TaskDescriptor, see Section 5.5.2.1.

5.3.2 Idempotent Side-effects

As mentioned about the dataflow execution model in Section 4.1.1, dataflow task execution has no side-effects until its execution completes, but the actual completion of a task is a process in itself, that might be interrupted and repeated in case of faults. Safe finalization process of a task is therefore critical to maintain air-tight fault-tolerance.

A task is considered complete only when the entire system sees it that way, not when its thread finished execution but only after all side-effects were submitted back to the safety of the shared memory. In order to prevent any chance of memory corruption and unexpected application behavior after a recovery from a hard-fault, special care is taken when handling the completion process of a task. This process takes place after the thread code was done executing but before the side-effects were committed to the shared memory and the task was marked completely completed. The finalization process accounts for the faulty case when the task is finalized a second time, if a crash happened during the finalization process that already released partial results to other nodes or even performed scheduling requests for new tasks. When the task is re-executed, the same results will be produced and sent similarly to the previous time, and repeated scheduling requests may also be re-sent if the node failed after starting to submit TaskLoad messages (Section 5.1.2.1) for new tasks when finalizing their parent (Section 5.2.3).

Solution overview  The operating system design ensures that no problems will appear as a result of repeated executions of tasks that previously failed. Problems from duplicated task results sent in TaskWrite messages (Section 5.1.2.2) are overcome by tracking not only the total amount of data
written to each task before starting it, but also where it was written. Section 5.3.3 explains the mechanism and its benefits.

To overcome conflicts from duplicated tasks that may be created after recovering a failed node, the operating system ensures that each task has a system-wide unique identifier (UFI). The mechanism also ensures that on the finalization of a recovered task (Section 5.2.3), each child task that was created is assigned the same identifier even on repeated executions if its parent. Moreover, the mechanism ensures that repeated scheduling requests are always sent to their original designated node.

With these mechanisms, it is possible to check that written task inputs and scheduling requests are not duplicates of ones received earlier.

**Reproducibility** The task finalization procedure (Section 5.2.3) ensures that repeated execution of recovered tasks will produce identical UFIs for their children as they did on the previously interrupted execution. To do this, the process publishes to shared memory the UFI that is being finalized and a list that contains the UFIs allocated to child tasks, which also stores their designated owner node. The list items are published (released) to shared memory before their matching scheduling requests are sent to the destinations. If the results publication process was interrupted and re-executed when the task was recovered, the scheduling information is recovered from shared memory and the new child tasks will be created with the same UFIs and owner nodes as on the previous execution.

**Duplicates detection** To verify a scheduling request is not a duplicate, the implementation utilizes the internal structure of UFIs (Section 5.5.1). Each node only needs to keep track of the single latest tasks identifier received from each of the other nodes. This small amount of information is enough because values for one of the fields in the UFIs are generated sequentially on each node, see Section 5.5.1.

### 5.3.3 Fault-Tolerant Inputs State Tracking

The correct order of execution for dependent tasks must be ensured even in the presence of faults, specifically faults that interrupted the process of communicating results between tasks. The issue addressed in this section describes the reliability aspects of the task execution stage described in Section 5.2.3. A task may only start executing once all its inputs are (fully) available (Section 4.1.1). Therefore, the proposed operating system
guarantees that duplicated or partial results that were written do not cause a task to start prematurely. This section describes how the solution was implemented.

The responsibility of detecting duplicated writes was assigned to the node receiving the written values, where the added logic was made simple and efficient. The solution described in this section also allows the TaskWrite messages to fit in two 64-bit machine words, as mentioned in Section 5.1.2.2. The method is to keep a bitmap of the offsets of inputs that were already written within the frame of a task (in units of 64 bits). A write of a 64-bit word to the kth position in the frame sets the kth bit in the map, but only after writing and publishing the new value to the shared memory. To know when a task can start executing, the number of its expected input values is specified and stored along with the bit-map when the task is created. Using this bit-map, the latest state of fully written input values of a task can always be recovered in any case of failure. The updated bit-map and the original number of required values are sufficient for safe scheduling of dataflow tasks. The mechanism prevents repeated writes of re-finalized interrupted tasks from corrupting the sequence of dataflow execution. Nodes receiving task results verify that the expected number of bits are set in the mentioned bit-map before starting to execute a task. Note that the bit-map implementation is now 64 bits in size, in order to fit it in one machine word, thus limiting the number of independent input values a dataflow task can wait for. Independent was noted because the task frame that contains the input values has no size limitation, and therefore each distinct input value can represent more data than a single word. Therefore, although 64 input words are a mere sum of 512 bytes, data beyond the first 512 bytes is also accessible to a task when it is ready to run. Regardless of this workaround, it is very easy to increase or completely remove the 64-values limitation by simply increasing the bit-map size, but there might not be a need to do so in future applications.

Other dataflow implementations keep a field named Synchronization Count (SC) along with the task descriptor [65], which indicates how many inputs values are needed by a certain thread. On thread creation, it is initiated to the number of inputs the thread is waiting for, and each write to the thread decreases the SC by one so when it reaches zero the thread can be executed. This simple mechanism is enough to support the dataflow logic on a non-faulty case but does not work correctly with the assumed architecture and constraints (Section 4.4.1, Section 4.4.2) once faults are considered possible. This synchronization count field alone is not enough to reconstruct the latest state of a task after a node failure. Take the example of a task waiting for two input values, in the case that a node failed during the commit phase.
described in Section 5.2.3 of another task that writes to the waiting task. In the unfortunate case that the writer crashed just after sending the first value and caused the target SC to be decreased by one. The almost-finished task will be recovered and executed again, entering the commit phase and sending the first value again and the second value for the first time. At the time when the first value is received again, the SC of the waiting task will be decreased to zero and the task will begin to execute even though it only received one of its inputs. This simple erroneous case cannot be solved with the SC aggregation alone, so additional information is needed to be able to track the state of individual inputs, to prevent prematurely launching a task.

5.3.4 Distributed Watchdog

A distributed watchdog was made to continuously monitor the health of the nodes in the system and detects hard-failures. It is composed of software agents running in each of the nodes, they monitor their neighbors and initiate the recovery process if a hard-fault is detected. Each node sends Heartbeat messages mentioned in Section 5.1.2 to all nodes periodically, itself including. Concordantly, a service running in each node periodically checks that a Heartbeat was recently received from all nodes. If watchdog agent in one of the nodes does not receive a Heartbeat from any of the other nodes for a long period of time, it triggers the recovery procedure for the presumably failed node, as described in Section 4.3.1.2.

The watchdog is an independent service from the main scheduler, it was made so it will be able to perform partial self-test of the local node and detect problems before the other nodes do. Subject for future work may be to add the ability to also recover from some hard node-local core faults without affecting the rest of the system.

In addition to the sign of life that any message generally signifies, the Heartbeat messages have more value. Each node adds instantaneous and cumulative metrics on its state to it, this information is used by other nodes to support better scheduling policies and load balancing, more on this subject is detailed in Section 5.4.3.

The frequency at which Heartbeat messages are sent controls the trade-off between detection latency and freshness of load-balancing metrics, and the mechanism’s overhead. An optimal interval for Heartbeat advertisements is not easy to find, even when ignoring the overhead of processing them. If on a very error-prone system one would want to have the shortest delay until a failure is detected and recovered, a very small watchdog timeout may not
work. The Heartbeat messages are transferred on the main communication channels between nodes and not in an out-of-band method, e.g., written to a fixed shared-memory location or using a dedicated hardware module. Having to travel through the message queues, their waiting time depends on workload; so if a node is busy, a very small watchdog timeout might cause a false detection even though there are Heartbeat messages waiting on the input queues. Having the possibility for such a catastrophic outcome, a limit is enforced by the schedulers on the amount of processing done between checks of messages from each of the nodes. With the strict boundary on the time-slice for checking each message queue, a reasonable watchdog timeout exists even for very busy workloads.

Although the work needed to process each Heartbeat is very small, it is expected that deeper tests and data analysis will be needed on an actual device, so a relatively significant reduction in computing costs with negligible sacrifice of reliability is having each watchdog agent only guard several other nodes and not all of them, so a similar safety level is maintained with much less computing effort.

This work suggests incorporating hardware support on future devices for more efficient core failure detection. If such a solution exists, it can be immediately utilized in the current implementation, triggering the existing watchdog starvation procedure. Informing any single watchdog agent that a failure happened is sufficient and avoids having to wait for a timeout.

5.3.5 Double Execution

As introduced in Section 4.3.2, Double Execution in this work is a lockstep algorithm performed on a per-task basis. The method applied in this work allows node-local detection and containment of soft errors with only a thin layer added over the execution procedure of a task. To detect soft-errors and prevent them from corrupting global state, a ready task starts its thread twice in independent environments on two different cores of the same node. The deferred side-effects buffers (Section 5.2.2) are compared once both threads complete, if they do not match then a fault is assumed and the task restarts. The process repeats until the outputs from both threads perfectly match.

Compared to the extensive system consideration for hard-faults, user-level transient faults are given attention in a much more localized area of the runtime. The distinction from hard-faults is because it was found possible to design the solution such that the transient and intermittent disturbances considered in this work can be detected and recovered from at the local node
level, without disturbing the global system. The guarded execution of tasks requires system-wide effort for recovery (Section 5.2), but the operation of the Double Execution mechanism is focused only on the thread creation and completion of a ready task.

Considering that the side-effects of tasks are deferred until their threads completes, as mentioned in Section 5.2.2, adding support for (pessimistic) Double Execution was straightforward. The developed dataflow runtime is built to contain the side-effects of a task while it runs and only commit the results on completion. Therefore, some of the reliability mechanisms that are needed to recover from hard-failures could also be directly utilized to easily add an efficient Double Execution overlay on user tasks. To perform Double Execution on a task, the additional effort on the operating system is only to create a second side-effects buffer for it and execute a duplicated thread with it. When both threads complete, the side-effects are committed only if both side-effects buffer match.

The OS implementations can, therefore, support use of Double Execution for selected tasks with very low overhead on thread management. The additional costs when using the implemented mechanism is the computation cost for outputs comparison of the two threads, and of course, the naive double execution, which adds significant delays if node utilization is high. Additional optimizations over Double Execution were not in the focus of this work, but the design allows other, more efficient, transient-faults tolerance methods to be easily incorporated.

Admittedly, a delimitation of the described fault model limited the recoverable soft-errors to happen only on user code (Section 2.3.4.2) because the Double Execution mechanism can only be directly applied to dataflow tasks. Double Execution could not be easily applied to the internal service threads that the presented operating system uses to support the user applications. Those OS services use long-lived kernel-level threads in each of the nodes that are written using traditional programming models and therefore do not qualify to be dataflow threads.

5.4 Task Placement

One of the unfathomable aspects to optimize in the design of such a massively parallel operating system is (wait for it...) task placement, i.e., the strategy to determine assignments of tasks to cores. These algorithms drastically affect the performance of applications running on the suggested operating system, as in most multiprocessing environments. With traditional multiprocessing
programming models on several contemporary cores, one could manually place tasks to processing cores and design the method for threading, synchronization and communication to reach acceptable performance. In a chip with thousands or more cores, however, optimizing task placement becomes too complex for a programmer to do manually. The help of a smart scheduler is required to optimize task placement in order to maximize parallelism and reduce data movements, communication congestion, memory access latencies, energy consumption, heat density and more.

Although the operating system suggested in this work assumes a memory architecture with several possible layers of NUMA in the hierarchy, the implemented prototype for the operating system does not yet fully consider it. The implementation models a simplified NUMA architecture of the memory: there are node-local domains that contain private memory and caches of each node, and a possibly partitioned shared memory. The model is further described in Section 2.2. In a future exascale system with more fine-grained memory access latencies, sophisticated scheduling methods will be required to optimize performance.

5.4.1 Load Balancing

Dynamic load-balancing is needed in order to optimize utilization of computational resources in the dynamic system, which improves performance and reliability. Imbalance in cores utilization may not only cause severe performance degradation, it may also increase power consumption and long-term chip wear due to uneven thermal cycling and hot-spots [6], [7].

Since the goal of this work is an overall design for a new operating system, the prototype implemented for it covers a wide range of functionality. In order to reach a fully working system (in simulation) during the course of the research, it was obvious that some aspects of the implementation would have to be simplified so only basic scheduling algorithms were implemented. Developing good load balancing algorithms for the similarly assumed architectures and fault models can be the subject of many works. The work described in [66], for example, considers several important factors for task placement with a fault-aware model similar to the assumptions of this work. Task placement algorithms that consider communication latencies and a distributed shared memory are suggested in [67], [68] and several thermal-aware techniques are discussed in [6], [69].
5.4.1.1 Placement Considerations

When a local scheduler is asked to assign a node to execute a task it can either be a new task, existing or recovered (see Section 5.4.2). The scheduler can use information from several sources to determine which node should execute the task. The considerations available in this work include:

- The immediate state of the local node, e.g., utilization of its hardware resources, temperature, failure rate, and more.
- State information about other nodes which is collected through heartbeats, see Section 5.4.3. Since heartbeats do not arrive frequently enough to provide a fresh view of each scheduling request, extrapolation is be used during the time intervals between heartbeats.
- Locality considerations that may apply to the task in relation to its ancestors or other tasks, such as communication, data, or code sharing.

Of course, additional sources of information should be used to improve task placement, such as those mentioned earlier in this section.

5.4.1.2 Task Placement

Although this work considers it is likely that the shared memory will introduce various possible latencies on accesses depending on the location of the originator and the data, as mentioned in Section 2.2, the demonstrated scheduling algorithms ignore some possible sub-partitioning of the shared memory in order to simplify implementation. The implemented task placement algorithms treat any location in the shared memory which is not owned by the local node as if has the same latency. With this simplification, each node has only two shared memory latencies for consideration - the latency on its own shared segment and the latency to segments of other nodes, thereby making the scheduling problem becomes somewhat simpler.

Local-first Placement Policy The basic scheduler first tries to assign a new task to itself, in the same node where a task request was generated. For a first choice this makes sense because it is likely to preserve the spatial locality of the referenced code and data by keeping parent and child tasks near each other. This is because without having additional insight on the optimal target, it is likely that tasks will communicate most of the results to their close family. Keeping coupled tasks physically close to each other will probably improve their access latencies. In addition, keeping the scheduling information in the same node requires less processing and shared memory space.
than sending it to another node over the shared memory communication channels (Section 5.1.1).

When a local node becomes over occupied with work it sends new work requests to other nodes using the placement policy described below. The implementation determines that a node is over occupied by simply setting a threshold on the amount of available space in the owned shared memory segment.

The effectiveness of this policy depends on application behavior, it can work well if all nodes have tasks that continue to generate additional tasks. However, if only a single node generates the main application tasks, then this policy would not initially balance the work well. Correspondingly, when running a single application for the evaluation of this work (Section 6.4), this policy was disabled and the following one was immediately used.

**Uniform Placement Policy**  When a node decides to forward a task, it can choose from several implemented ways to pick a target. In the experiments presented later in Chapter 6, a method that tries to distribute load uniformly among living nodes is used. The load balancer picks the living node with the smallest amount of pending work, as far as it can tell. To estimate which node is least occupied, each node keeps track of the number of waiting tasks on each of the others. The estimation is updated from gathered Heartbeat messages (see Section 5.4.3) and linear prediction is used for the time in between received Heartbeat updates.

### 5.4.2 Scheduling Junctions

There are several events that a node’s scheduler handles that require it to decide where to place a specified dataflow task, it immediately needs to assign a node to execute it. The decision can be adjusted for a certain task several times, the main decision points are the following:

- When a new dataflow task is explicitly created by legacy type (non-data-flow) user code at the beginning of a large parallel operation. Those first tasks of the operation can create child tasks to expand parallelization.
- New dataflow tasks can be created during the finalization process of their parent task (the processes is illustrated in Section 5.2.3). Considering that the dataflow framework is targeted to performing large computations, the vast majority of scheduling decisions are a part of the results of some parent task. The focus is then mostly given
to optimizing this case, although the same techniques can generally apply to all cases.

- After the task received all its inputs and became ready to execute, this is a bonus opportunity to reconsider the assignment. Note that the task could not be moved before it became ready to run, because the first scheduling decision is incorporated into the UFI of the task, it may be a target of results from other tasks (see Section 5.5.1). If the task is relocated before receiving all its inputs, the values it needs will never reach it. Once the task is ready to run, it can be re-assigned to any node, this might be a good idea if there was a significant change in load distribution in the system since the task was created, e.g., the original owner became over utilized. Note that re-balancing tasks at this point has additional costs compared to the distribution cost at the time of creation. The current implementation can either use the same algorithm that was used on the original scheduling, or simply assume there is no change in considerations.

- When a task is recovered from the shared memory of a failed node. The node assigned to handle the recovery process can immediately rebalance all of the ready tasks it finds in the abandoned memory. Recovered tasks that are not ready can only be rebalanced once they become ready, as mentioned in the previous point. Similarly to the case in the previous point, holding onto a recovered task and later re-balancing when it becomes ready has a cost, but non-ready tasks cannot be rebalanced because their pending inputs would be sent to the wrong destination.

All scheduling junctions can use mostly similar load-balancing considerations, except for having more recent state information.

### 5.4.3 Heartbeats Collection

In addition to the sign of life that the Heartbeat message represents to the distributed watchdog (see Section 5.3.4), each node also adds some local state information to it. Other nodes use this information to improve scheduling policies and load balancing. Metrics added to Heartbeat messages include an estimation of workload on the node, temperature, fault rate, message creation time-stamp, the time-stamp of when the last Heartbeat was received in the other direction and several other metrics on the local kernel. The information is sent to help other nodes decide on the best targets for new tasks. It is easy to imagine many more useful measurements and insights that can be gathered and considered for scheduling decisions. Of
course, adding many complex metrics has a cost, so for each, one should consider the benefit and how often it should be sampled and sent, and if those can be reduced by estimation or prediction. The frequency of published Heartbeat messages is therefore not trivial to decide on, having a fresh view of the global state, or even just of nearby nodes, is valuable but gathering some of the metrics may be costly.

5.4.3.0A Inter-node latency measurements  It may be beneficial in a future system to measure various delays in the system, latencies for each inter-node channel and workloads of each remote node separately, in order to optimize task placement and scheduling. Some of the latencies of interest, such as access times to different parts of the shared-memory, might be available through hardware interfaces on a future device, or be estimated accurately enough according to the architecture. The current OS implementation, however, does not assume much on the underlying memory hierarchy, and it also cares about delays accumulated on the software levels that hardware cannot measure, those also vary according to specific workloads. To avoid dependencies on future hardware and use indirect means to estimate important latencies, a direct method to calculate end-to-end response times is used. Its goal is to estimate the total response times between each pair of local node schedulers, in other words, how soon a node can be expected to process a command sent to it from a specific location in the system. To do this, each node simply adds a time-stamp to each Heartbeat when it is created, a receiver node collects recent time-stamps from all other nodes and can then estimate several factors it might depend on. These algorithms were not investigated in the work since the goal was to create a functional system. That said, several example statistics can be captured to supply the task-placement algorithms:

• Comparing the received time-stamps to local ones during processing measures the time it took to travel the shared-memory and the input queue from the sender. Accuracy depends on clock synchronization.

• Measuring timing jitter across a moving window of previous Heartbeat intervals measures the stability of the channel.

• Reading these timing values that the remote node captured in the other direction. This depends on the time it takes for each remote node to process their messages.

The current node implementation performs only simple processing of Heartbeat messages, so it was expected and experimentally supported that changing the update frequency between tens to hundreds of milliseconds did not have a noticeable effect on throughput. Still, building more useful insight on the
global state to perform smarter task placement would require more work to
gather the metrics to send and more sophisticated analysis of the received
messages. Considering the processing costs of the additional work, a few
easy changes can be done to fit different scenarios:

- Each node could keep a single Heartbeat on a fixed shared-memory
  location and periodically update (release) it so all nodes can read it.
  This method reduces the communication overhead and congestion on
  the message queues but the real-time measure of latency in messages
  will be lost. This change will also remove the ability of nodes to use
  the information gathered about them in the remote nodes, which was
  sent back to them.
- Each Heartbeat is now sent in an all-to-all manner, but optimization
  could set each node only inform several of its neighboring nodes. If
  we assume or measure a low probability for nodes from far sides of
  the chip to communicate, we do not need them to know the precise
  state and workload of each other.

## 5.5 Implementation Details

### 5.5.1 Unique Frame Identifiers

A small dive into the structure of a Unique Frame Identifier (UFI) could help
understand how the run-time mechanisms could be implemented efficiently.
A UFI is a single word of 64 bits so it can be transferred and stored efficiently.
The structure of a UFI is shown in Figure 5.3, it contains the following
fields:

**Source node identifier** The ID of the node that created the task.

**Destination node identifier** The ID of the node that was originally des-
tined to execute the task. This field does not change even if the
 task is later re-assigned to a different node, which may happen for
load-balancing reasons or when it was recovered from a failed node.

**Node-local task identifier** Source-local unique identifier of the task. It is
unique only among tasks that were created by the node that created
the task. It enables system-side unique identifier generation without
any synchronization with the other nodes. Values for this field are
generated in increasing order, it cycles back to zero on overflow. The
ordering from the running counter is used for performing some checks
when receiving TaskLoad, see Section 5.1.3.2.
VFP flag (reserved at 0)  This flag specifies whether the value associated with the write to the target task is an address within a frame of a VFP, i.e., a temporary task identifier, returned to a running thread to defer its side-effects until it completes (see Section 5.1.3.1). When this flag is set, the task finalizing process (Section 5.2.3) knows that the value field is an address within a newly created task, and not a generic value that should stay as-is. If the translation is needed, the finalization process maps the value to write from addressing an offset in a temporary task identifier according to the final address that was assigned to the task, and then clears the flag. Note that the target of a TaskWrite might also need translation regardless of this flag; this state is detected simply by the value of the destination node field that contains a predefined value (see Section 5.2.2.3).

Input offset (reserved at 0)  This field remains zero to be later masked with frame offsets in TaskWrite (Section 5.1.2.2). Using this field enables a TaskWrite request to be only two 64-bit words, making it efficiently buffered for running tasks when deferring their side-effects and later transferred to other nodes if needed. The target address in TaskWrite is a single word that combines the destination UFI and the input offset within the frame of the task.

Input offset (reserved at 0)

<table>
<thead>
<tr>
<th>63</th>
<th>52</th>
<th>51</th>
<th>40</th>
<th>39</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src. node</td>
<td>Dest. node</td>
<td>Node-local task identifier</td>
<td>Write offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reserved

Figure 5.3: Structure of a task identifier (UFI)

5.5.2 Task Descriptor

This section illustrates the structure of a TaskDescriptor object, which contains all information required to execute a given task. These objects are transferred between nodes in TaskLoad messages (Section 5.1.3.2) and are also kept in a shared collection of the node that was assigned to execute the task.

A new dataflow task is created with several attributes, as specified by the DF instructions, those need to be kept with the task information when it is assigned to a node or restored from backup. One of the fields in the descriptor specifies the thread routine, as explained earlier in Section 5.1.2.1; either the thread name is included or the actual thread binary. Other fields in the descriptor are the frame size of the task that contain its input values,
the state of written input values (see Section 5.3.3), declarations of the user-level shared memory regions the thread needs to access and room to hold the input values are all included with the TaskDescriptor. Additional fields are used for validation when recovering the descriptor, as mentioned in Section 5.5.2.1.

The content of the descriptor is illustrated on Figure 5.4.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task identifier (UFI)</td>
<td>Unique identifier for the task descriptor.</td>
</tr>
<tr>
<td>Written inputs bitmap</td>
<td>Bitmap indicating which inputs have been written.</td>
</tr>
<tr>
<td>Frame size (bytes)</td>
<td>Size of the buffer where inputs are written.</td>
</tr>
<tr>
<td>Number of shared regions</td>
<td>Count of shared memory regions.</td>
</tr>
<tr>
<td>Size of binary or name (bytes)</td>
<td>Size of the binary or name segment.</td>
</tr>
<tr>
<td>Number of inputs (words)</td>
<td>Count of input values.</td>
</tr>
<tr>
<td>Validation cookie</td>
<td>Cookie used for validation.</td>
</tr>
<tr>
<td>Checksum (up to this point)</td>
<td>Checksum calculated so far.</td>
</tr>
<tr>
<td>Thread name or compiled binary</td>
<td>Name of the thread or compiled binary version.</td>
</tr>
<tr>
<td>Thread frame</td>
<td>Buffer where task inputs are written.</td>
</tr>
<tr>
<td>Task inputs are written here as they arrive</td>
<td>Available to the user when the thread starts.</td>
</tr>
<tr>
<td>User region 1 offset (words)</td>
<td>Offset of the first user region.</td>
</tr>
<tr>
<td>Size (bytes)</td>
<td>Size of the user region.</td>
</tr>
<tr>
<td>r/w</td>
<td>Read/write permission.</td>
</tr>
<tr>
<td>User region 2 offset (words)</td>
<td>Offset of the second user region.</td>
</tr>
<tr>
<td>User region N offset (words)</td>
<td>Offset of the Nth user region.</td>
</tr>
<tr>
<td>User-defined shared regions declarations</td>
<td>User-defined shared memory region declarations.</td>
</tr>
</tbody>
</table>

**Figure 5.4:** Structure of a TaskDescriptor

### 5.5.2.1 Structure Validation

Additional fields in the descriptor are used to check for the validity of the structure. The check is needed when recovering work from a failed node (see Section 4.3.1.2), in a case that the collection of the shared descriptors was corrupted. When creating a new TaskDescriptor in shared memory,
the validation cookie is the last field that the system issues a release for (see details on memory semantics in Section 2.2.2.2). The field has a constant value which is considered unlikely to appear in data by chance. Since the field is only published after the rest of the structure, its existence in memory means it is likely that a TaskDescriptor surrounds it. When the field is found in memory, the structure is checked against the checksum field. The discovered TaskDescriptor enters the recovered collection only if the check succeeded. If the check failed it either means that the validation cookie appeared in memory by chance, or that the structure was not completely written when the fault occurred. If the structure is not complete, it means that it is still stored in the messaging channels, and can still be recovered (see Section 5.1.3.2).

5.5.3 Thread Binaries

Each user task in the system is associated with a compiled thread binary to execute. The compiled binary can contain any type of user code in any size, from a single dataflow thread to a whole executable that has never heard of dataflow. Although tasks can be stand-alone user applications, once they received all of their inputs and began executing, they do not further participate in the data-flow game that created them. Those applications can initiate new dataflow procedures but are considered outsiders to the process that created them. From the moment they started executing, no further fault-tolerance is automatically provided for these tasks, as they can contain any legacy code and have no insurance on side-effects during execution. Since a primary goal of this work is creating a new execution engine, the discussions throughout the thesis deal mainly with task binaries that remain managed by the engine, those that do not become independent stand-alone applications.

The threads themselves are then usually dataflow tasks, i.e., some serial code and its static data, they can even be just a single function. These tasks are not whole programs but work only as a part of a larger application. To launch a distributed application over the entire system, not all threads need to be copied to all of the nodes. A single node can load the whole executable, e.g., only the front-end node, and the rest of the nodes may only load the code of the tasks that they need to execute. The mechanism saves a lot of bandwidth and space when the system contains hundreds of nodes. For example, the binaries copied to other nodes in the Fibonacci implementation are no bigger than a few hundred bytes. Each thread is compiled as PIC (Position-independent code) so it can be copied and executed from the shared memory which can be mapped differently in each node. There are
several ways threads binary code can be created: They can be compiled separately from the user executable, or compiled as a part of it. If the thread binary is separate from the application executable it can be pre-compiled before the application is launched or at run-time by a JIT compiler. These combinations are supported simultaneously. There are two ways a node can get the thread binary it needs to execute once a TaskLoad message it received. Depending on the binary size of threads and the number of times each binary is used, one of the following is preferable:

- The TaskLoad message contains the actual binary for execution. The executing node does not need to have additional access to disk or shared-memory in order to run the code. This also means a node can execute threads from any application without preparation. This way is probably preferable if the threads are small or if many kinds of them are dynamically compiled at run-time. Using this method of operation resembles message-passing paradigms more than shared-memory ones.

- Only the binary name is included in the TaskLoad message, a node receiving this information will look in its buffer of previously launched threads for the specified thread and try to load it. If the thread is not in the buffer the node assumes it was pre-compiled and tries to load it from disk. This option is probably better if threads are only pre-compiled or are relatively large in size.

The mode to use is determined statically for each thread but it could be automatically determined at run-time based on the thread size or history of its usage.

![Binary layout of a dataflow thread](image)

*Figure 5.5: Binary layout of a dataflow thread*
5.5.4 **Service Threads**

Besides user tasks, some of the OS threads also need access to the shared memory, since the message queues, thread binaries and TaskDescriptor objects are stored there. These supporting tasks use a special kind of Data-flow (DF) threads internally for some of its service threads, created with a similar interface as the one to create user DF threads but having extra privileges.

These threads are not the normal dataflow threads that have no side-effects, have a short life-span and just need a single *acquire* operation, some of the service threads are persistent to the lifetime of the node, they are allowed to have side-effects before their completion and they require a more powerful access to the shared regions. They are responsible to maintain the communication channels through the shared memory, poll on message queues from other nodes and run message pumps.

5.5.5 **Dataflow Threads**

The mechanism that executes the user dataflow tasks can have several implementations, there were several flavors of future hardware support in mind during development, some options are left that allow passing some of the control on threads to hardware using added instruction set extensions. Regardless of the chosen method, if one wishes to add the possibility of transient faults, as assumed by the model described in Section 2.3.4.2, then those can currently be to detected and recovered from at the OS level. Basic Double Execution can be enabled on a per-task basis, i.e., starting each task on two different cores in parallel, and only committing the side-effects if the results match. More details on DE are described in Section 5.3.5.

5.5.5.0A **Hardware Supported Dataflow**

The current implementation assumes that special hardware support is only required to access the shared memory, since its used with explicit synchronization operations through a new hardware interface (see Section 2.2.2.2), but not to perform any of the data-flow scheduling logic. The software-only solution suggested in this work provides greater portability and flexibility compared to an equivalent hardware solution, but it has higher overhead compared to an application-specific hardware that may have intimate access to the memory buses. Other works suggest the use of a specialized hardware to perform coarse-grained dataflow scheduling on a many-core architecture [59].
Correspondingly, one may assume that each node has a local data-flow-aware hardware scheduler, capable of monitoring and executing tasks according to their inputs availability. The presented operating system, therefore, has added support to transfer some control of a task to a hardware scheduler. The OS can notify the hardware scheduler about task creation and its required inputs, then relinquish control over the task until being notified of its completion. Once the control returns to the OS when execution completes, the OS performs the regular finalization of the task and its side-effects (see Section 5.2.3) just as it would without the additional hardware support. The functionality for leveraging hardware-supported dataflow threads was implemented and tested in experiments using the dataflow x86-64 instruction set extensions described in [70].

5.5.5.0B Software Implemented Dataflow Another supported variation that is implemented for dataflow threads creates the DF threads as regular kernel-level threads (pthreads). Although the possibility that a future many-core device will support a form of a dataflow in hardware, to remain compatible with existing x86-64 cores as was desired by the model described in Section 2.2.1, the implemented framework does not rely on special dataflow-aware hardware. All dataflow thread management and inter and intra-node scheduling is currently implemented at the software layer, so there is no requirement for hardware-supported dataflow threads. This mode enables either the local OS kernel or a user-level thread pool to manage the dataflow computation in the node and share them among many running applications, making it more flexible as opposed to relying on the hardware scheduler module.
The ambition of this work is not only to propose a new idea of a software solution for a fault-tolerant many-core operating system, but also to create a functional prototype for it. The top-level goal of the experiments performed is to verify the feasibility and to illustrate the behavior and performance of the proposed operating system.

The operating system was implemented to a working prototype that can run general applications, its functionality and performance were then evaluated in a simulated environment for the many-core system described in Chapter 2. A simulation is necessary since the assumed hardware architecture has not yet become a reality. This work leveraged an existing multi/many-core full-system simulator to verify the functionality of the system and evaluate its performance under different architecture configurations and workloads. The chosen simulation tool, that will be described in Section 6.1, also allows integrating custom behavioral and instrumentation code at the simulator level. This powerful feature was used to implement the assumed shared-memory semantics, including its timing behavior, and gain complete visibility on internal states of the proposed operating system without affecting the performance results.

This chapter is structured as follows: Section 6.1 describes the platform used to simulate the assumed hardware model that was described in Chapter 2. An overview of how experiments tracing data was generated, analyzed and aggregated to meaningful results is described in Section 6.2. Section 6.3 mentions the high-level objectives of the simulated experiments. Section 6.4 describes the user application used in the experiments and Section 6.4 presents and discusses some of the experimental results.

6.1 Full-system Simulation

This section briefly describes the simulation tools that were used throughout the development of this research and while carrying out the evaluation. A timing-based simulator (a.k.a. performance simulator) is required for an accurate evaluation, as opposed to a functional emulator/virtualizer alone (e.g., VMware, VirtualBox), since it allows achieving a high degree of
accuracy not only in functional tests, but also in the timing behavior of the full architecture. Depending on the accuracy of timing, variations in the way parallel instructions are interleaved in the simulation may cause a ripple effect and lead to significant changes in the overall behavior.

Assuming the hardware is based on off-the-shelf processing cores, as mentioned in Section 2.1, has the practical benefit that it minimizes the required modifications to existing tools in order to accurately evaluate the system. The main simulation environment used during this research was created by leveraging the open-source COTSon simulation framework [60], [71] to model the assumed hardware architecture and its interface to software. HP's Exascale Computing Lab and AMD originally created the tool, it targets simulations of clustered architectures with hundreds or more multi-core nodes. COTSon controls the timing aspects of the simulation and uses AMD's SimNow [72] as the functional simulator for nodes. SimNow supports simulation of generic x86-64 processors, which makes it suitable for the requirements of the assumed node model in this work.

With this powerful tool, the architecture layers and timing policies of the simulation could be controlled to trade accuracy for speed. Various configurations were used while developing and evaluating the operating system, ranging from fast simulations that test functionality while neglecting timing information, to using more accurate timing models of architectural components to assess the system's performance. However, reducing the simulation accuracy cannot be avoided even when performance results are needed for real-world applications because an accurate timing simulation can be over five orders of magnitude slower than a real system. Therefore, some accuracy had to be sacrificed in order to gather results that cover a wide design space for many large experiments ($10^{10} - 10^{12}$ cycles), such as those presented in this chapter.

6.1.1 Simulator Operation

To run experiments on the system, the many-cores machine is simulated using the COTSon full-system simulator. The simulation host (1 TB RAM, 64 x86-64 Opteron cores) runs a VM instance for each node in the global system, each simulates several cores, e.g. 4/8/16/32 cores per node were used in the presented experiments. Each simulated node boots into its own kernel/microkernel and starts the run-time environment created in this work that takes over its operation and joins the global system through the new shared memory interface.
Although the simulator creates a full x86-64 virtual environment for each node and the compute nodes now run a Linux-like kernel, application tasks may not assume that such environment exists. Dataflow tasks assume a bare-bone processor, the only external services available to them are those supplied by the run-time environment created in this work. This restriction is enforced by the compilation process of task binaries (Section 5.5.3), which prevents the tasks from having additional dependencies.

6.1.2 Simulator Modifications

Some simulator capabilities were modified and extended in this work in order to facilitate the assumed shared memory architecture and new software-hardware interfaces. The simulator allows adding custom x86 ISA extensions to the simulated cores, to create a functional implementation for them at the simulator level, and provide custom on-line timing specifications.

To implement the consistency model of the shared memory in simulation, an implementation of a similar model [73] was adapted to meet the assumptions of this work in terms of the semantics of the distributed shared memory between nodes (Section 2.2.2.2). The functional principle of the shared memory implementation is that the host allocates a memory region outside the simulated environment and allows the VMs to access it indirectly using special machine instructions that are captured by custom simulator code. The behavior of some existing instructions was modified and a few were added to expose a new interface to the shared memory assumed in this work,
which implements the memory semantics described in Section 2.2.2. As the simulation runs, the simulator layer also supplies timing specifications of the new instructions back to the simulator.

Some additional changes and extensions were made in this work to the existing simulation environment, mainly to support the evaluation stages of the research. The changes were crucial to streamlining the experiments execution process and to monitor the behavior and collect tracing data from simulations, as mentioned in Section 6.2.

6.2 Investigation Methodology

This section provides an overview of the process that was iteratively used to expose information on the behavior and performance of the proposed operating system. Acquiring meaningful experimental results and isolating some of the different behavioral aspects of the proposed system was challenging. A lot can be learned from each experiment launched but originally, the information was far from being readily available.

Each experiment can take a very long time to run in simulation and should be repeated to decrease variance. Some of the graphs presented in this chapter contain results from experiments totaling weeks of simulation time. Although the results from each experiment appear as no more than a single dot in most of the presented graphs, there is a lot of data gathered on various types of metrics on each experiment.

Automated tools were developed in this work to help carry out many simulated experiments on various hardware architectures and software configurations. Additional support tools were also made to process the huge amounts of data generated during the simulations. The tools analyze and aggregate the results from all experiments to be easily accessible for investigation, they also provide an interface that allows to easily explore the results visually.

The high-level operations involved in an evaluation cycle of the system include:

**Tracing** Instrumentation code is added at the simulator level to generate the trace data required to collect the additional metric of interest. The tracing is performed on the simulator layer and exported during execution from each node or core. Data generation is done by modifications to the simulator and not at the OS level, because it allows extensive amounts of accurate data to be generated without affecting the timing of the simulated system. Some key points of the execution can be traced if needed at the guest (simulated) level, but only a few should
be used since it affects the timing results. The only ones used in the presented experiments are those marking hard-fault detection and the beginning of the recovery process.

**Configuration** Specifying which system configurations are of interest for the desired experiment. The instructions can be simple lists of ranges for parameter combinations to run, but since the number of available parameters and their options became large, shortcuts are also available. The list of possible configuration options was ever growing during development and evaluation, but it was also regularly trimmed when results revealed insignificant effects of some changes. A few of the major parameters in most configurations are usually the number of nodes in the system and cores count in each of them, the user application to run and its attributes, latencies specifications for shared-memory operations and possible injected faults.

**Simulation** The configuration directives are supplied to a script that was created in this work in order to automate the simulation process for series of experiments. It builds and runs the simulation for each specified configuration, monitors the simulation health during execution and collects the generated data. Extensive amounts of data are generated for each node in the simulation, from both using preexisting simulator features and by the added tracing features specific to this work. Control was added to choose which modules to trace and level of specification, in order to speed up the simulation when focusing on limited aspects of the behavior.

**Processing** A program that post-processes the generated simulation data was created in this work and evolved as additional aspects of the execution were examined. Its purpose is to expose various high-level behavior aspects from the low-level scale of all the scattered tracing data. This tool was vital to reaching meaningful conclusions on many aspects of the system that would have remained hidden otherwise, if only the raw outputs were used.

**Exploration** The results from the previous step are gathered into a single file that contains aggregated results of all experiments. This file allows investigating the results without the need to repeatedly access and process the terabyte of experiment outputs. The generated file is structured as a tabular spreadsheet, which makes it easy to explore in many data analysis tools (Python, MATLAB and Microsoft Excel were mainly used during the research).

The aggregated data contains both temporal information on each experiment and summarized end results. All the evaluation graphs
presented in this thesis were generated from performing slices on the data in this results file. Although only a handful number of graphs are presented in this document, there is a lot more depth in the results that is readily available to be explored and examined by the tools created in this work.

6.3 Goals of the Experiments

The main goal of the experiments was to verify the feasibility and test the functionality of the new kind of operating system proposed in this work. The implementation of the proposed operating system covers a broad range of different types of functionality, creating realistic experiments for the entire system is challenging.

This section describes the objectives that the performed experiments aim to achieve.

Functionality Evaluation

The main focus of the experiments is to test functional aspects of the created operating system and verify overall resilience under faulty conditions. The experiments coverage ranges from scenarios targeting a single module of the runtime, to testing system-wide behavior and performance. The three main aspects of the functional tests are described in the following paragraphs.

**Functionality of all OS modules:** Performed by examining basic operations of the new operating system mechanisms on various workloads with faulty and non-faulty cases. Special focus was given to the dataflow engine and inter-node communication services such as the transport layer, message queues and message processing. The overall operating system was also tested as a whole, by running complete user applications and distributing their work to all nodes with pre-compiled dataflow threads. Tracing of shared memory accesses and operations of OS modules shows the scheduling of tasks according to their dependencies, and the communication of scheduling requests and tasks results over the shared memory. Tests performed by analyzing tracing data from the simulation and verifying correct results from applications.

**Hard-fault detection and recovery:** Reliable detection of faulty nodes on various workloads, continuous micro-checkpoints in a weak-consistency
memory, recovery of pending work from failed nodes, recovery of all unprocessed and partially processed inter-node messages after a fault, applications recovery and completion after a surprise node failure, tolerance to nested/cascaded node failures, i.e., node failure during the recovery process. Hard failures were simulated by violently halting nodes at random times during the experiments.

Transit fault detection and recovery: Detection by Double Execution, i.e., executing a single dataflow task on two threads in sibling cores, deferring thread side-effects until verified, returning temporary system identifiers to threads and later replacing with final ones, transient fault detection on user code. Faulty cases were simulated by randomly injecting various amounts of errors into results of user tasks.

Performance Evaluation

There are many types of actors in the implemented operating system and interactions between them, they span various levels of the solution; from directly interfacing hardware and reaching up to the application level. Since each of the actors in the system dramatically affects the overall performance, comprehensive performance results could not be reached during the research with the available resources for simulation and development. Therefore, the experimental results presented in this chapter have a limited view of the wide design space that can be explored on the implemented operating system prototype. The following paragraphs describe the high-level performance aspects of the system that are examined.

Parallelization: Measure the amount of parallelization that is supplied to applications on various configurations, from a few cores to many cores. The main resulting metric is the parallel speedup factor compared to serial execution time.

Scalability: Measure the overhead of the operating system when increasing the number of cores and nodes, and varying other key parameters in the architecture and the application.
6.4 Experiments and Results

This section presents some of the behavioral and performance results gathered from the experiments on the simulation platform described in the previous sections. The following sub-sections illustrate the effects of some of the significant control parameters on behavior, and performance.

Each experiment was defined by some set of system parameters that affect key aspects of the architecture structure and characteristics, the operating system, the application behavior and fault conditions. The effects of around 20 parameters were experimentally investigated on the full-system simulator. Tools were made to perform groups of tests using ranges of system parameters and gather the results, and to process the information on an ever-increasing number of metrics from the traces of the experiments executions. The investigation methodology is described in Section 6.2. Test results from thousands of parameter combinations were collected and made easy to explore and analyze. Many aspects of the results can be investigated from a collective level, down to slicing individual runs. Some parameters had subtle and sometimes unexpected effects that were not immediately explainable, investigative work was sometimes required in order to track down reasons for patterns that appeared on the results.

The simulation setup (Section 6.1) used in the experiments was limited to running up to 1024 cores, the experiments presented in this section show results from various possible partitions of those 1024 cores to nodes. We all love powers of 2 and know that to reach 64 total cores, for example, one can have 2 nodes with 32 cores each, 4 nodes with 16 cores each, 8 nodes with 8 cores, 16 nodes of 4 cores etc. For this work, however, not all combinations were investigated. For each total cores count there are 4 extreme combinations, e.g., $1 \times 64$, $2 \times 32$, $32 \times 2$, $64 \times 1$ (nodes \times cores), those resemble non-clustered systems and were therefore considered not interesting in this work since they do not illustrate the effects of the clustered hierarchy.

**Application Under Test**

The purpose of the user application used in the presented experiments is to demonstrate several aspects of the functioning system developed in this work. A single application was used throughout all experiments and was used with no modifications across all of them unless where explicitly mentioned. Please note that although a single test application was used, the developed prototype can run general applications. Of course, the behavior
and performance when running other types of workloads from different applications should also be evaluated in future work. Using a single application in most experiments keeps most experiments comparable to each other, gradually revealing many patterns that would not have been noticed or searched for otherwise.

The chosen application should be highly parallelized so it will not be a limiting factor on the performance scaling when increasing core count. All experiments were performed on a recursive calculator for the Fibonacci series. The heart of the application code was presented earlier, it is listed as-is on Section 4.1.3. Although the chosen application is only a rudimentary example of dataflow, and terribly inefficient when computing the Fibonacci series, it is very effective in stressing the operating system modules and its dataflow engine. The double recursion in the implementation quickly spawns a very large number of threads that are not completed until then process folds back. Because of the simplicity of the dependency graph in the example, a minified snapshot of the threads in dataflow engine can be easily visualized, see Figure 6.1.

### 6.4.1 Application Timeline

Functional tracing information is exported from the operating system during simulation, it contains descriptions of selected operations and their timestamps. When the simulation completes, the data generated from all nodes is collected and processed to extract both global metrics on the execution and high-frequency function trace. Figure 6.2 shows two examples for real-time data that can be examined during the execution. It provides deeper insight into the online behavior compared to more aggregated metrics, such as total executed cycles in the system, which are presented in following sections. Figure 6.2a shows that throughput reaches a stable maximum level in a relatively short time, both for short and long thread durations. When nearing the completion of the computation, throughput drops equally fast as not enough parallelized work remains to stay at full utilization. Figure 6.2b shows the total number of tasks that have all their inputs available and await execution. Jitter is higher when tasks are very small, there are over 100000 dataflow threads that complete every second on just 32 cores.

**Deeper look**  The previous experiments are expanded to show the behavior on several cores partitioning and the workload of each node is colored in different shade within the experiments. The results presented in Figure 6.3 show a more detailed view of the results shown in Figure 6.2.
Figure 6.2: Throughput and parallel workload during run-time

32 cores in 8 nodes, fib(25), 225k tasks executed

Figure 6.4, Figure 6.7, Figure 6.5 and Figure 6.6 aim to show a detailed view of the results from additional architecture configurations. The figures show the wall-clock time on the horizontal axis in each of the graphs, it indicates the time to completion when the graph reaches zero and there are no tasks left to execute. Please note that these are stacked graphs, the top outline of each one is equal to the total number of ready tasks in the system, as similarly depicted in Figure 6.2b.

The scalability of the system when changing nodes count and cores partitioning can be seen in the graphs by examining the completion time of each experiment. These experiments show that when doubling the cores count, the completion time is roughly reduced to half. More detailed scalability tests are demonstrated later in this chapter so this section will continue with
the focus on behavior. Some differences between the configurations are clear when comparing several experiments that have the same total cores count but use a different partitioning of the cores. For example, although the experiments with $4 \times 16$, $8 \times 8$ and $16 \times 4$ (nodes $\times$ cores) all have the same total cores count, the configurations with fewer nodes and more cores in each node have completed a little sooner and the workload was more balanced throughout the execution.

All experiments show some imbalance of load on the first seconds of the experiment, as indicated by the ragged behavior. This effect increases generally becomes more noticeable when the tests are short, because the initial stabilization time becomes more significant in the overall execution time. The erratic behavior is also increased when there are more nodes in the system, mainly because the workload estimation they have on each other is not always accurate, as mentioned in Section 5.4.1. The duration of the initial imbalance of load can be reduced by issuing more frequent Heartbeats (see Section 5.4.3) or by improving the estimation of load on neighboring nodes, which is a significant consideration on task placement, as mentioned in Section 5.4.1.1. Longer application tasks also help to smooth the performance behavior because they lower the stress on the task-placement mechanisms, allowing the operating system more time to
Figure 6.5: Workload versus cores per node - 4 nodes
16..128 cores, fib(25), 225k tasks executed, average 2.2 ms/task.

Figure 6.6: Workload versus cores per node - 8 nodes
32..256 cores, fib(25), 225k tasks executed, average 2.2 ms/task.

communicate load-balancing information between nodes (Section 5.4.3), as demonstrated in Figure 6.7 that shows results from an experiment with an average of 11.5ms/task.

Figure 6.7: Workload versus nodes - long tasks duration
32..256 cores, 8 cores per node, 225k tasks executed, average 11.5 ms/task.

6.4.2 Recovery from Faults

This sections describes how the behavior of the system was tested in response to faults.

The faults were injected on many test scenarios such as various nodes/cores configurations, workloads, time of the failure and more. The fault recovery and the correct completion of user applications were verified automatically by comparing the user-level results to a non-faulty execution. Although the fault-tolerance experiments were carried out on many architecture configurations, the top-level performance characteristics of the presented operating
system were similar on many of them. Therefore, results from several experiments are presented to demonstrate the main features of the system behavior and performance following a hard-fault. The results shown in Figure 6.8 and Figure 6.9 complement the hard-fault recovery scenarios demonstrated in Section 4.3.1.3 and Figure 1.1.

Tolerance to node hard-faults was tested by forcing a surprise failure of a node during the distributed execution of the user application. Section 7.3.0.0A shows an example of the output from these experiments in the simulator. Statistics about various aggregated metrics from the architecture components are collected on each experiment, as mentioned in Section 6.2. In addition, real-time traces from the local kernels and underlying simulation layers are collected as the program runs, which are then used to visualize the performance of the system. Real-time traces include the number of threads started and completed, number of tasks that is ready to run or waiting for inputs, shared memory accesses and more.

![Figure 6.8: Increased hard-fault detection delay - 128 cores in 4 nodes](image)

Fault detection is artificially delayed and the recovery starts just before all available work is complete. Note that the throughput of the remaining nodes is only (slightly) affected when the recovery process begins.

### 6.4.3 Speedup Through Parallelization

These experiments demonstrate the speedup achieved by the presented operating system when the total number of cores increases, and the impact of the tasks granularity on efficiency. Only the number of cores and threads
The application is launched with varying amounts of computation that each task must perform, which varies its duration. To artificially increase the work of each task, instructions are inserted in each dataflow thread in addition to the work that the user code does. Figure 6.10 shows that while having threads duration long enough, the system scales well as the number of cores increases. Perfect scaling would make horizontally flat graphs on Figure 6.10a and a straight line from \((16_{\text{cores}}, 100\%)\) to \((512_{\text{cores}}, 3.125\%)\) on Figure 6.10b. Perfect scaling means that:

$$\text{perfect parallel run time} = \frac{\text{serial run time}}{N_{\text{cores}}}$$

Where \textit{serial run-time} is the total duration it would take a single core to run the entire application, and \textit{parallel run-time} is the time duration until all cores completed the parallel execution.

The values in Figure 6.10a represent the total work that was performed by the machine in order to run the application, including OS overhead, which...
is further examined on Section 6.4.4. In the presented results, serial run time is measured in terms of CPU core cycles, it is the sum of clock cycles performed by all cores of the processor. Consider that the execution was parallelized to on more than one processing unit; therefore, the elapsed real time is much shorter than the combined CPU time. Thus, although the values in Figure 6.10a are increasing w.r.t. the number of cores, the actual wall-clock duration of the calculation is (mostly) decreasing, as evident from the parallel run time presented in Figure 6.10b.

(a) Serial run time - sum of the cycles performed on all cores

(b) Parallel (wall-clock) run time, relative to a baseline of 16 cores

![Graph showing speedup versus the number of cores](image)

Figure 6.10: Speedup versus the number of cores

It is clear from Figure 6.10b that relatively long threads, with duration around 2.2 ms each, improve the ability of the system to scale well as core counts increases. The longer threads on the example are still rather small, the speedup the system offered when going from 16 cores to 128 is $\times 7.58$ and $\times 14.66$ when going to 256 cores. When the tasks are the smallest and each lasts around 0.14 ms in the example, there is a point where the OS overhead overcomes the speedup from the added parallelization of user code.
on additional cores. The tipping point occurred on the presented example on the transition from 256 cores to 512 cores, where the parallel run time began to increase.

The efficient scaling is expected to continue beyond 256 cores on more practical applications than the application in the tests (see Section 6.4), with real-world tasks lasting more than a few milliseconds each. The behavior with longer threads than are presented here was not yet examined because even at this scale, an experiment to produce a single point on the graph can take days to run in a full-system simulation.

These results present a significant limiting factor on scaling that is highly dependent on the application behavior - user tasks duration. This is not a surprising discovery because such is expected in any parallel system. As in any distributed architecture, the overhead always increases with core count, due to the additional maintenance and distribution of work to more processing units. The main challenge in the industry is to maintain acceptable efficiency when scaling-out to increasing number of processing units. This challenge becomes easier when the duration of isolated tasks increases because it mitigates the overhead associated with communication, synchronization and reliability.

6.4.4 Overhead of Parallelization and Reliability

The experiments in this section emphasize the amount of work performed by the operating-system itself when running user applications on non-faulty cases, i.e., the overhead from the distribution of tasks and the reliability mechanisms.

6.4.4.1 Effects of Parallel Workload

The results shown in Figure 6.11 are for experiments with very small tasks, each one with a duration of around 140 microseconds. Having tasks so small makes the overhead of the operating-system the main contributor to the total cycles performed by the device. These encouraging results show that the amount of work performed by the operating system has a linear dependency on both the total number of cores in the system and on the total threads executed. Furthermore, the average overhead on each task was not found to be significantly dependent on the parallelized workload.

When varying the number of parallelized user tasks to perform from 86k to 953k, Figure 6.11 shows that the change on average task overhead stayed
within ±10% across all cores configurations, and increased by 25%-75% for each doubling of the cores count across the range of user workloads.

The tests with 512 and 1024 cores begin to align with the trend only toward larger tests since the small tests (up to 225k tasks) ended before reaching a stable throughput (100-300ms). Note that in order to keep the graph simple, each point on the graph is an average of from all cores/nodes partitioning configuration that produces the same total cores count, Section 6.4.4.2 examines the differences in performance between different cores partitioning, and Section 6.4.5 shows the equivalent results when varying the duration of tasks.

![Figure 6.11: OS dominated cycles on different workloads](image)

fib(23..28), average 0.14 ms/task.

### 6.4.4.2 Impact of Cores Partitioning

The previous section examined the performance depending on the total number of cores in the system, but as mentioned, that was an aggregation of results from several cores configurations. Since the assumed many-core processor uses a clustered architecture, the impact on behavior and performance from changes in hardware partitioning was of critical interest during the research. Varying cores partitioning greatly aided the operating system development and calibration process by exposing hidden patterns in the complex behavior.

Several patterns appear when the same application is launched on different cores partitioning, Figure 6.12 shows the OS overhead when varying the number of nodes and number of cores in each node.

The test application used in the experiments shown in Figure 6.12 has very short user threads, 140 microseconds average duration, the results therefore
Figure 6.12: OS dominated cycles on different configurations

fib(27), 589k tasks executed, average 0.14 ms/task.

mainly show the OS work needed to distribute and synchronize the work over all of the cores and maintain resiliency. The following paragraphs examine several patterns that are immediately noticeable in the experiments when examining increases in nodes count and increases in the number of cores per node.

Note that the total cycles count which is presented in the graph also contains the relatively inefficient prefix and suffix of the application, when only a few tasks are ready to be executed in the system. While only a few cores perform actual work, the rest are idle at full speed, wasting cycles. Since the tasks are so short, the inefficient prefix and suffix of the application greatly contribute to the differences between results on different configurations, since the ramp up and down in throughput take similar times but there are a different number of cores cycling idly. This appeared to be the main cause for the negative effect on increasing the number of cores per node for the same node count.

To further examine the cost of distribution, notice the difference in the total work performed on dual configurations in Figure 6.12, for instance: $4 \times 8$, $8 \times 4$ (nodes $\times$ cores). Both configurations have 32 cores total, but Figure 6.12 shows that having 4 nodes of 8 cores is more efficient than having 8 nodes of 4 cores. The OS does not present this property by chance, it is to be expected since the design considers NUMA, where the overhead of sending work items to remote nodes is more expensive than to neighboring cores in the local node, as discussed in Section 2.2.2.3. Following the goal mentioned in Section 3.2, the scheduling algorithm described in Section 5.4 and other
run-time mechanisms aim to minimize dependency on expensive shared-memory communication. Although node-local communication is preferred, a top priority of the OS is to balance load and to maximize parallelism by utilizing all nodes in the processor. The difference in overhead for dual cores configurations is therefore expected since more shared-memory communication is required to distribute work on an $8 \times 4$ (nodes $\times$ cores) system than on $4 \times 8$.

The previous paragraphs mentioned some repeated relations between experiment results but more can be learned by adding another dimension to the design space. Notice the differences in each group of 4 experiments having the same cores per nodes in Figure 6.12. A repeated pattern is seen for other groups with a different number of cores in each node, as the architecture changes from 4 to 8 to 16 and 32 nodes. Figure 6.13 clarifies the pattern by varying the duration of tasks on each configuration. As task durations become more dominant, the relative differences between cores configurations decrease because the operating system overhead is not significantly affected. The change in overall cycles count is larger on the configurations that were more initially more efficient, as discussed earlier in this section.

![Figure 6.13: Run-time with varied tasks durations and cores partitioning](image)

**Figure 6.13:** Run-time with varied tasks durations and cores partitioning

fib(25), 225k tasks executed

### 6.4.5 Impact of Parallel Workload

In the following experiments the duration of tasks remained fixed while varying the total amount of computation application required, which creates different workloads that the system must handle at the same time. These
scenarios stress the parallelization capabilities of the system by quickly generating large number of small tasks that can run in parallel.

**Figure 6.14** shows the total amount of cycles performed while the tasks duration remained fixes at 1.75 ms average. The results show that varying the total amount of tasks the application required by some factor effected the total cycles performed by a similar factor, ±10% as seen the the previous section. This is demonstrated across several cores configurations.

![Figure 6.14: Varied application workloads on different cores configuration](image)

These experiments also show that although the total work performed by the machine to complete the computation remains similar on most configurations for partitioning cores into nodes, there is an advantage to an increased number of cores per node over increased nodes count. This was also observed in the results of Section 6.4.4.

The differences in the instantaneous workload of nodes between cores configurations are evident in **Figure 6.15**. When clustering the cores into fewer nodes there are more tasks that each node must keep track of at the same time. As before, the local kernels have no trouble maintaining efficiency up to 32 cores per node so the overall efficiency is still improving when there are more cores in each node compared to having more nodes. The performance improves when using more cores per node because it allows spending more time processing tasks from the node-local memory and requiring expensive shared-memory operations less often.
The results show that the operating system can maintain consistent application performance regardless of utilization, throughput does not deteriorate when increasing workload.

Figure 6.15: Impact of cores partitioning on workload

*fib*(25), 225k total tasks, average 1.75 ms/task
7.1 Goal of the Experiment

This experiment launches a distributed Fibonacci number calculator over the presented operating system. During execution, the simulation displays the operations performed by the runtime and the user code in the virtual monitor of each virtual machine instance, additionally, the output is logged and can be examined after execution. Soft-errors can be injected randomly to some of the threads to demonstrate Double Execution in action. Hard node failures can be triggered during execution by the user with a keystroke to watch the recovery mechanism. Various compile-time flags and launch arguments can be modified to change parameters of the runtime and processor architecture, e.g., the number of nodes and cores, cache arrangement, simulation timing accuracy, shared memory related latencies, scheduling algorithms, dataflow threads model, logging level and more.

The visualization and logging level of each OS module can be controlled individually by the user and changed during runtime at any point. A list of possible keystrokes is displayed on the simulated monitor.

7.2 Instructions to Start

The first step is getting the simulation environment. Check-out and build the COTSon simulator with:
$ export COTSONHOME=<desired installation dir>/cotson
$ svn co https://svn.code.sf.net/p/cotson/code/ $COTSONHOME
$ cd $COTSONHOME/trunk
$ ./configure && make release

Where COTSONHOME is an environmental variable identifying the path where COTSon will be copied to. If anything above goes wrong, please read the detailed instructions in cotson/trunk/README.

The run-time environment and sample applications are contained in the following folder:

$COTSONHOME/branches/tflux-test/tfos/

To run the experiment navigate to the tfos folder mentioned above:

$ cd $COTSONHOME/branches/tflux-test/tfos/

Now start the simulation by executing:

$ make run_multi

After startup, the simulator window will display general information about the node and list the accessible commands, e.g., show logs, test node failure, and others that can be interactively triggered by the user with a keyboard command on the SimNow window. Additional parameters can be configured in the launch scripts and in the code. For example the number of nodes in the system is specified in tfos/os-tests/tsu_multi.lua:

cluster_nodes=4

The number of cores in each node is specified by the bsd file used:

use_bsd('32p.bsd') -- Can also be '4p.bsd', '8p.bsd' or '16p.bsd'.

To test node crashes it is recommended to have more than 4 cores in each node. Notice that the bsd’s with large number of cores are not created using the default build configuration, they can be downloaded from:

https://upload.teraflux.eu/uploads/BSDS/
  bsd_images_initialized_for_karmc64_1Ghz.tar.gz

Some other parameters are specified in tfos/os-tests/Makefile:

OWMSZ=67108864 # Size of the shared memory.
SZ=44 # Parameter for the application (e.g. Fibonacci number).
#NT=32 # Worker threads per node. Leave undefined to use the number of cores.
Several additional parameters are specified as compile time flags, they enable some variations of the dataflow tasks implementation and can be used to evaluate the system under fault conditions.

**Fault-tolerance and fault injection**  For example, to specify whether to enable Double Execution (Section 5.3.5), and whether to randomly corrupt some of the threads results in order to see the fault-tolerance in action, the following macros can be used:

```c
#define DOUBLE_EXECUTION 1
#define INJECT_CORRUPTIONS 0
```

During execution, the user can manually inject a hard-fault to the system by keyboard input (see Figure 7.1). In addition, a hard-fault can be automatically injected if the following flags are set:

```c
// Setting INJECT_HARD_FAULT to 1 will cause a hard-fault after // roughly MIN_TASKS_BEFORE_CRASH tasks were executed.
#define INJECT_HARD_FAULT 0
#define MIN_TASKS_BEFORE_CRASH 150000
```

**Scheduling control**  To specify whether to include the actual thread binary in the TaskLoad scheduling requests (see Section 5.1.2), or only the name of the pre-compiled binary, the following flag can be set:

```c
#define SEND_TASK_NAMES 1
```

When the flag is cleared to zero, each TaskLoad message is self-contained and allows immediate execution on any node without access to shared storage for loading the pre-compiled threads, at the cost of possibly sending the same binary code many times, see Section 5.5.3 for more details. Although threads are usually small, e.g., 100-200 bytes for the Fibonacci example on Section 4.1.5, the overhead can be avoided by sending a small thread identifier instead of the binary code, later used to load the thread from the common file system. When thread identifiers are used, the loaded binaries are cached so subsequent requests are quickly loaded from the node-local memory, without accessing the file system.

Scheduling algorithm variations can be selected with simple macros. As mentioned in Section 5.4, the implemented algorithms are rudimentary examples but illustrate how the information gathered from heartbeats (see Section 5.4.3) can be used to help load balancing among nodes.
// Prefer to place tasks on the local node unless utilization is high, then
// a secondary method is used. If this is not enabled, the method chosen
// below is immediately used.
#define PREFER_PLACING_LOCALLY 1

// Enable only one of the following:
#define RANDOM_SCHED_POLICY 0
#define UNIFORM_DISTRIBUTION_POLICY 1

7.3 Expected Output

When launched, node instances will open in SimNow windows and display
the simulation progress:

![Simulation screenshot](image_url)

Figure 7.1: Simulation screenshot

When the simulation completes, the behavior of each node can be examined
in the `stdout` log files, provided that logging support was enabled at compile
time, and activated for the requested modules at run-time by default or on
keyboard command. Example output of a node is presented in Listing 7.1.

Listing 7.1: Example output trace of the dataflow engine

```c
[Manager 1] System parameters:
[Manager 1] 16 cores in 4 nodes with 4 cores each.
[Manager 1] 64MB public shared memory, 16MB per node.
[Manager 1] 4*1MB message queues, leaves 12MB for dynamic allocation.
[Manager 1] Starting service thread, ip 0x4202e0.
[Scheduler 1] Dynamic allocation area rounded from 0x7fffa46f4140 to 0x7fffa46f5000,
size 12MB.
[Manager 1] Starting service thread, ip 0x40c360.
[Test] Computing fibonacci(41).
[Scheduler 1] Starting message pump.
[Scheduler 1] Submitting task fib_reporter_task with UFI 10010000000200.
[Node 1 Writer] Sending message type 1, 73 bytes in 2 frames.
```
7.3.0.0A Injecting a permanent fault  If a certain node (node 3 in the example) was killed by user input, the recovery node (node 1 was chosen in this experiment) will begin to take over and process the work of the failed node. The display should appear similar to the following:
The log of the recovery node would show additional details on the recovery process:

Listing 7.2: Hard-fault detection and recovery

```
[Watchdog] Node 3 probably died, no heartbeat received in the last 189 milliseconds.
[Manager 1] Starting recovery procedure for node 3.
[Manager 1] Starting service thread, ip 0x406d60.
[Recovery Scheduler for 3] Checking shared segment sanity...
[Recovery Scheduler for 3] Task descriptors map in shared memory has 37 items.
[Recovery Scheduler for 3] Adding new task from desc 0x7ffff65793c0.
[Task 1003000000bc00] Creating task from desc 0x7ffff65793c0, UFI 1003000000bc00, original sc 2, current sc 0.
[Task 1003000000bc00] Ready.
[Task 1003000000bc00] [tid 7fffe3ff700] fib main for n=29 - calculating.
[Task 2003000000e600] Creating task from desc 0x7ffff65791e0, UFI 2003000000e600, original sc 2, current sc 2.
[Task 2003000000e600] Ready.
[Task 2003000000e600] [tid 7fffe3ff700] fib main for n=29 - calculating.
[Recovery Scheduler for 3] Got thread write message, tloc 2003000000e601, value 0x1e.
... Processing more recovered messages ...
```

Figure 7.2: Recovery status screenshot
7.3.0.0B Injecting a transient fault  If Double Execution and random error injections are enabled, an injected soft-error will produce output similar to the following:

![Output screenshot]

**Figure 7.3:** Double Execution in action

This is a simple implementation of Double Execution as described in Section 5.3.5; each task is executed twice (notice the different tid on each execution), and the results are not committed to the shared memory until the results of both threads are ready and found equal. When an error is injected, the mechanism detects it and launches the task again on two threads.
This thesis presented a new fault-tolerant distributed operating system for future many-core shared-memory systems. The operating system is able to dynamically distribute applications over a clustered architecture with thousands of cores and a shared-memory without hardware-based consistency. The system is resilient to transient and permanent hardware faults that are assumed on future exascale processors, without requiring special hardware assistance. Distributed software mechanisms detect various faults and provide on-the-fly recovery, allowing the operating system and all running applications to continue their correct operation with minimal delays. The cost of fault isolation and recovery is minimized due to the extremely small size of checkpoints that are implicitly maintained at the level of single words. All permanent faults and harsh transient faults in nodes are detected by their neighbors through a distributed watchdog mechanism, which then initiates a global recovery procedure that restarts the affected tasks. User-level transient faults are detected and recovered at a node-local level by a Double Execution mechanism. During recovery, any task that was active during the fault can be safely restarted because the run-time environment ensures that all side-effects of dataflow tasks are idempotent.

The distributed run-time environment implicitly allows a single application that is parallelized using dataflow principles to efficiently utilize all cores in the dynamic system, even when simple task-placement algorithms are employed. Static or dynamic (i.e., at run-time) variations of the architecture parameters, such as available cores and nodes count, shared memory capacity, and expected fault types, do not require user-level modifications at compile time, nor involvement at run-time.

The shared memory as the only means of communication between nodes, the operating system explicitly manages all consistency over it. A message-passing layer was added using the shared memory to assist task-placement and communication. Thus, a fully wait-free, shared-memory/message-passing hybrid parallelization engine was created. It transfers only small amounts of data on the messaging channels and relies on the weakly-consistent shared-memory to minimize duplication and data migration between nodes.
Experiments on a many-core full-system simulator for up to 1024 cores showed encouraging results. The distributed operating system efficiently scaled up and out with cores and nodes count until reaching application limits for additional performance benefits from parallelization. The results show that with a fixed total cores count, the efficiency benefits from increasing the grouping of cores into nodes at the expense of having fewer nodes. This is to be expected since the proposed operating system promotes locality and prioritizes placing new tasks on the originating node, to avoid the extra overhead of inter-node operations. The expected tipping point is when some nodes become over utilized and it is worthwhile to pay the inter-node communication fee to transfer work to underutilized nodes. This tipping point, however, was not reached even up to 1024 cores in 32 cores per node, which was the largest machine configuration supported by the simulator.

When trying to assess the scalability of the system under different conditions, the most significant factor that affects the efficiency of scaling was the coarseness of application tasks. When keeping the total number of tasks to execute fixed, and varying the duration of each task from 0.14 ms to 2.2 ms the speedup greatly improved with additional cores. Efficient scaling is expected to continue to larger number of cores when running more practical applications than those tested at the time of writing.

Furthermore, the proposed operating system is ready to be extended to be used in different architectures and parallelization paradigms for future many-core systems. To name a few example, the system is ready to accept different task placement algorithms, fault detection units, inter-node communication channels and even task execution models other than dataflow.
9.1 Experiment Output of 512 Cores

The following Listing 9.1 shows the dataflow engine output from a single node on a 512-core experiment. A small test application was used in order to keep the presented output short, only 4 threads were executed on the monitored node. Note that the experiments that were launched for evaluation purposes had logging disabled to prevent it from shadowing the performance results.

Listing 9.1: Node trace output on a 512-core experiment

[Manager 1] System parameters:
[Manager 1] 512 cores in 16 nodes with 32 cores each.
[Manager 1] Shared memory mapped to 0x7fff97ffd000.
[Manager 1] 1536MB public shared memory, 96MB per node.
[Manager 1] 16*2MB message queues, leaves 64MB for dynamic allocation.
[Subscriber 1] Node 1 shared region at offset 0, size 100663296:
[Subscriber 1] Loopback input queue info at frame offset 0, output queue info at frame offset 24, OWM offset at 0 for both.
[Subscriber 1] Dynamic allocation at frame offset 48, OWM offset 33555712.
[Subscriber 1] Node 2 shared region at offset 100663296, size 100663296:
[Subscriber 1] Input queue info at frame offset 56, OWM offset 2097232.
[Subscriber 1] Output queue info at frame offset 80, OWM offset 100663296.
[Subscriber 1] Dynamic allocation at frame offset 104, OWM offset 134219008.
[Subscriber 1] Node 3 shared region at offset 201326592, size 100663296:
[Subscriber 1] Input queue info at frame offset 112, OWM offset 4194464.
[Subscriber 1] Output queue info at frame offset 136, OWM offset 201326592.
[Subscriber 1] Dynamic allocation at frame offset 160, OWM offset 234882304.
...subscribing to more message queues and public arenas of the other nodes...
[Subscriber 1] Node 15 shared region at offset 1409286144, size 100663296:
[Subscriber 1] Input queue info at frame offset 784, OWM offset 293621248.
[Subscriber 1] Output queue info at frame offset 808, OWM offset 1409286144.
[Subscriber 1] Dynamic allocation at frame offset 832, OWM offset 1442841856.
[Subscriber 1] Node 16 shared region at offset 1509949440, size 100663296:
[Subscriber 1] Input queue info at frame offset 840, OWM offset 31458480.
[Subscriber 1] Output queue info at frame offset 864, OWM offset 1509949440.
[Subscriber 1] Dynamic allocation at frame offset 888, OWM offset 1543505152.
[Manager 1] Starting service thread, ip 0x426960.
[Manager 1] Starting service thread, ip 0x402c80.
[Manager 1] Starting service thread, ip 0x40c380.
[Node 1->1 Writer] Sending message type 3, 40 bytes in 1 frames.
[Node 1->2 Writer] Sending message type 3, 40 bytes in 1 frames.
...more sent heartbeats...
[Node 1->16 Writer] Sending message type 3, 40 bytes in 1 frames.
[Fib Test] Computing fibonacci(4).
[Scheduler 1] Starting message pump.
[Node 1->1] Received message type 3, 40 bytes in 1 frames.
(Node 1->1) Got heartbeat - has 0 pending tasks.
(NODE 2->1) Received message type 3, 40 bytes in 1 frames.
(NODE 16->1) Got heartbeat - has 0 pending tasks.
(NODE 16->1) Got heartbeat - has 0 pending tasks.

(Scheduler 1) Submitting task fib_reporter_task with UFI 10010000000200.
(Node 1->1 Writer) Sending message type 1, 73 bytes in 2 frames.
(Scheduler 1) Submitting task fib_main_task with UFI 10010000000400.
(Node 1->1 Writer) Sending message type 1, 77 bytes in 2 frames.

(Scheduler 1) Finalizing 0: Write value mapped VFP 0 to UFI 10010000000200.
(Scheduler 1) Finalizing 0: Write target mapped VFP 200 to UFI 10010000000400.
(Scheduler 1) Submitting write to node 1, tloc 10010000000400.
(Node 1->1 Writer) Sending message type 4, 24 bytes in 1 frames.
(Scheduler 1) Finalizing 0: Write target mapped VFP 200 to UFI 10010000000400.
(Scheduler 1) Submitting write to node 1, tloc 10010000000400.
(Node 1->1 Writer) Sending message type 4, 24 bytes in 1 frames.

(Node 1->1) Received message type 1, 73 bytes in 2 frames.

(Scheduler 1) Got task load message for UFI 10010000000200, binary size 17, frame size 8, 0 OWM regions, sc 1.

(Node 1) Got thread write message, tloc 10010000000400, value 0x10010000000200.

(Scheduler 1) Creating task desc for UFI 10010000000200 @ 0x7fff99ffe150.
(Scheduler 1) Adding new task from desc 0x7fff99ffe150.
(Task 10010000000200) Creating task from desc 0x7fff99ffe150, UFI 10010000000200, original sc 1, current sc 1.


(Node 1) Received message type 1, 77 bytes in 2 frames.

(Scheduler 1) Got task load message for UFI 10010000000400, binary size 13, frame size 16, 0 OWM regions, sc 2.

(Scheduler 1) Creating task desc for UFI 10010000000400 @ 0x7fff99ffe1d0.
(Scheduler 1) Adding new task from desc 0x7fff99ffe1d0.
(Task 10010000000400) Creating task from desc 0x7fff99ffe1d0, UFI 10010000000400, original sc 2, current sc 2.

[BinariesStore] New binary requested: fib_main_task.
[BinariesStore] Adding task binary: fib_main.task, 1617 bytes.

(Node 1) Received message type 4, 24 bytes in 1 frames.

(Scheduler 1) Got thread write message, tloc 10010000000400, value 0x10010000000200.

(Node 1) Received message type 4, 24 bytes in 1 frames.

(Scheduler 1) Got thread write message, tloc 10010000000401, value 0x4.

(Task 10010000000400) Ready.
(Task 10010000000400) Started.
(Task 10010000000400) [tid 2200000000] fib main for n=4 - spawning.
(Task 10010000000400) Ended.

(Scheduler 1) Submitting task fib_main.task with UFI 10020000000600.
(Node 1->2 Writer) Sending message type 1, 77 bytes in 2 frames.
(Scheduler 1) Submitting task fib_main.task with UFI 10030000000800.
(Node 1->3 Writer) Sending message type 1, 77 bytes in 2 frames.

(Scheduler 1) Submitting task fib.adder.task with UFI 10040000000a00.
(Node 1->4 Writer) Sending message type 1, 86 bytes in 2 frames.

(Scheduler 1) Finalizing 10010000000400: Write target mapped VFP 0 to UFI 10020000000600.
(Scheduler 1) Submitting write to node 2, tloc 10020000000600.

(Node 1->2 Writer) Sending message type 4, 24 bytes in 1 frames.

(Scheduler 1) Finalizing 10010000000400: Write target mapped VFP 200 to UFI 10030000000800.
(Scheduler 1) Submitting write to node 3, tloc 10030000000800.

(Node 1->3 Writer) Sending message type 4, 24 bytes in 1 frames.

(Scheduler 1) Finalizing 10010000000400: Write target mapped VFP 400 to UFI 10040000000a00.
(Scheduler 1) Submitting write to node 4, tloc 10040000000a00.

(Node 1->4 Writer) Sending message type 4, 24 bytes in 1 frames.

(Scheduler 1) Finalizing 10010000000400: Write value mapped VFP 400 to UFI 10040000000a00.
Scheduler 1] Finalizing 10010000000400: Write target mapped VFP 0 to UFI 10020000000600.
Scheduler 1] Submitting write to node 2, tloc 10020000000600.
[Node 1->2 Writer] Sending message type 4, 24 bytes in 1 frames.
Scheduler 1] Finalizing 10010000000400: Write value mapped VFP 400 to UFI 10040000000600.
Scheduler 1] Submitting write to node 3, tloc 10030000000800.
[Node 1->3 Writer] Sending message type 4, 24 bytes in 1 frames.
[Node 2->1] Received message type 1, 77 bytes in 2 frames.
Scheduler 1] Got task load message for UFI 20010000000200, binary size 13, frame size 16, 0 OWM regions, sc 2.
Scheduler 1] Creating task desc for UFI 20010000000200 @ 0x7fff99ffe1d0.
Scheduler 1] Adding new task from desc 0x7fff99ffe1d0.
[Task 20010000000200] Creating task from desc 0x7fff99ffe1d0, UFI 20010000000200, original sc 2, current sc 2.
[Node 2->1] Received message type 4, 24 bytes in 1 frames.
Scheduler 1] Got thread write message, tloc 20010000000201, value 0x2.
[Node 2->1] Received message type 4, 24 bytes in 1 frames.
Scheduler 1] Got thread write message, tloc 20010000000200, value 0x20040000000601.
[Task 20010000000200] Ready.
[Task 20010000000200] Started.
[Task 20010000000200] [tid 2500000000] fib main for n=2 - calculating.
Scheduler 1] Submitting write to node 4, tloc 20040000000601.
[Node 1->4 Writer] Sending message type 4, 24 bytes in 1 frames.
[Node 4->1] Received message type 4, 24 bytes in 1 frames.
Scheduler 1] Got thread write message, tloc 10010000000200, value 0x3.
[Task 10010000000200] Ready.
[Task 10010000000200] Started.
[Task 10010000000200] [tid 2900000000] Reporter task: fib result is 3.
[Task 10010000000200] Exit requested.
[Task 10010000000200] Ended.
Scheduler 1] Sending termination requests...
[Node 1->2 Writer] Sending message type 5, 8 bytes in 1 frames.
[Node 1->3 Writer] Sending message type 5, 8 bytes in 1 frames.
[Node 1->16 Writer] Sending message type 5, 8 bytes in 1 frames.
...more termination requests...
Scheduler 1] Exiting.
Scheduler 1] Task descriptors map in shared memory has 0 items.
Scheduler 1] Task descriptors map in local memory has 0 items:
  0 initializing
  0 waiting for inputs
  0 ready
  0 running
  0 finished
  4 total completed
References


References


[16] The Teraflux project, teraflux.eu.


[52] R. Kim, U. E. Avci, and I. A. Young, “CMOS performance benchmarking of Si, InAs, GaAs, and Ge nanowire n- and pMOSFETs with Lg=13nm based on atomistic quantum transport simulation including strain effects”, in 2015 IEEE International Electron Devices Meeting (IEDM), IEEE, Dec. 2015, pp. 34.1.1–34.1.4. DOI: 10.1109/iedm.2015.7409824.


אימומת התתחנות לש התמונונים: אתחולה המבנה ביכורים המושקף, עצרת
לעבירה מיתא ביכורים המושקף, וחזור תחתיות בין העבירה לכל עמותה לדורש לכל סכום ב
צמתים. מתן העברה מבנה של אראים המ頓ונים התמונונים עם עמוד פמידה, הל蚬ת מבני שמות התמונונים התמונונים בין הצמתים, וחזור דיני של
Data-flow, אוביקטים שנועדו לשיבוץ המושקף, התאצות בצורת המחברת התמונונים.

אימומת התخصوص לשלים וemetery לכלוב: הורדת לש שגיאות שנועדו לשכת התוחלת
המשתמש שאמות עם ערכה ביהוח, התאוששות, והספיקת והזרעה הנכונה לש.

המשתמשות

אימומת התخصوص שלגיונות קושי לכלוב: הקפתびת של שיבובים של שגרה את המוחות
שאמרות צות, התאוששות, והספיקת והזרעה וזורעה דע לתפימה
המורצל של התוכנה המושקפת, מהל שוי היחידה לכל מודעות לзнанияה.

ethereum Dwight שיתוף שיתוף בבלוב: האוכלוסיון התתחנה ביבול של التطوير
היזםוד של, האוכלוסיון התתחנה ביבול של התמונונים, עליותнятие התתחנה בבלוב ביצוע התומן, חיון התמונונים, אורכון מדריך
ולシリーズdong, תוקלח מינימום התוחלת התוחלת הת吻ושショップים שוניין עם
פיפוס מתחי, השפעת שבעה בבלוב במערכות בשנית הת泽连, עם א
הורדה בבלוב.

132
The experiments and measurements taken include:

- Ability to simulate a large number of cores with configurations in the simulator.
- Changes in the architecture, such as the distribution of tasks, the execution of the shared memory, the validation of the new instructions, and the synchronization of their execution.
- Collection of real-time data for the instructions that were executed and the state of the execution system.

The experiments were conducted on a simulator that met the following requirements:

- The ability to simulate various architectures and platforms.
- The ability to simulate various configurations and settings.
- The ability to simulate various execution scenarios.

The conclusions of the experiments were published in the following journal articles:

- "Simulation of Instruction Sets in a Real-World Environment" (Journal of Computer Science, 2018).
- "Simulation of Architectures and Platforms in a Real-World Environment" (Computer Science Journal, 2019).
- "Simulation of Execution Scenarios in a Real-World Environment" (Computer Science Journal, 2020).

The work was supervised by Professor (Name), a member of the Computer Science Department at Technion - Israel Institute of Technology.
The work assumes that future processors will consist of multiple cores, each with a dedicated memory. In order to provide support for these processors, the cores are divided into sets with a limited number of cores each. It is assumed that each set is similar in structure to the memory in modern processors. Each set contains private memories such as registers, memories for instructions, and others.

The sets are not required to be homogeneous; they can be of different types within the processor, suited for different purposes such as graphical interfaces, data storage, and others. Each set in the processor is independent, in contrast to modern processors, where there is a shared memory cache for all sets.

The work suggests that there is no need for symmetric access to all parts of the memory, and that even if there are no semantic or temporal boundaries, the cores can access different parts of the shared memory in a direct manner.

The hypothesis is based on the assumption that the needs of today’s memory have not changed, despite the increasing requirements on the memory in current computers.

The work examines how the memory hierarchy is used in future processors and suggests that the assumptions of modern memory hierarchies are not valid for future processors.

The memory hierarchy is examined, and it is suggested that the assumptions of modern memory hierarchies are not valid for future processors.

The work proposes a new memory hierarchy that is more compact and efficient, with a memory cache for each core, and a shared memory for all sets.

The work concludes that the assumptions of modern memory hierarchies are not valid for future processors, and suggests a new memory hierarchy that is more compact and efficient, with a memory cache for each core, and a shared memory for all sets.
The development of a new system for real-time execution environment in heterogeneous multi-core processors, which is designed for reliable and efficient execution of applications on a single processor core.

The system is designed to exploit the full parallelism of the multi-core architecture without sacrificing speed or efficiency. The design of the system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.

The system is designed to handle the demanding requirements of modern multi-core processors, which have a large number of independent processing elements and a shared memory. The system allows applications to take advantage of the full parallelism of the system in an efficient way, and in addition, it ensures the security and robustness of the applications under various conditions.
הצ📛יר

Multiplicity of Processing Elements (cores, cores) is increasing in a significant manner; currently it is possible to find dozens and even hundreds of cores in a single processor, and in the future, the number of cores is expected to increase significantly to thousands and even tens of thousands of cores within a single processor. Research in this area focuses on cores-rich processors (multi-core), which are in contrast to processors with many cores (many-core).

Modern systems, system models, and software are demanding more and more of this parallelism, even if it is present in current systems, it is important to deal with this challenge, especially in future systems with a large parallelism.

This dramatic increase in computing power in a single processor, however, requires a continuing shrink in the size of transistors, which leads to a significant decrease in the performance of processors, as the number of transistors decreases, and the problems of external interference, defects in production, which vary between cores, and failures that accumulate in the contiguous connections caused as a result of mechanical stresses, electromigration, and additional failures that spread and cause thermal, electrical, and mechanical damage, which lead to the aging of the material. Therefore, there is a significant increase in accidents that occur during the period of use of the processor. Therefore, a basic assumption in this research is that the cores are capable of performing calculations in a timely manner, and are susceptible to permanent failures, while the processor continues to operate.

The likelihood of random failures is reduced, in the processors of today's systems. In the modern running systems, the systems that run and applications that use the system run in a relatively smooth and stable manner, in a way that a crash that occurs in a critical calculation is unacceptable. Although there are many reliable systems in the world, the reliability of the system is not a reliable system, and this is changing.

This work presents a planning for a running system that enables the software to use the parallelism that is available in the future processors, and also to deal with the likelihood of failures that occur in the future processors, so that the software that is used, or other developers of the software that is used, can leave the remaining stages of the system.
המחקר בוצע בהנחיית של פרופסור אביגיל מנדסמן, באוניברסיטת תל אביב.
מערכת פעלה חסינה
למעבדים עתיירי ליובח

תיבר או מתכון

לשם مليח חלקי של הדרישות לקבלת התואר
곤יסות למודיעים במודעי המהשים

עמיית פוקס

הוגשלסן טכניון—מכון טכנולוגי לישראל
טבריה ח"' חפלה
דצמבר 2017
מערכת הפעלה חסינה
למעבדיםעוירי ליבת

עמיית פוקס