ABSTRACT

ZIMMER, CHRISTOPHER J. Bringing Efficiency and Predictability to Massive Multi-core NoC Architectures. (Under the direction of Frank Mueller.)

Massive multi-core network-on-chip (NoC) processors represent the next stage in both embedded and general purpose computing. These novel architecture designs with abundant processing resources and increased scalability address the frequency limits of modern processors, power/leakage constraints, and the scalability limits of system bus interconnects. NoC architectures are particularly interesting in both the real-time embedded and high-performance computing domains. Abundant processing resources have the potential to simplify scheduling and represent a shift away from single core utilization concerns *e.g.,* within the model of the "dark silicon" abstraction that promotes a 1-to-1 task-to-core mapping with frequent core activations/deactivations. Additionally, due to silicon constraints, massive multi-core processors often contain simplified processor pipelines that provide an increase in predictability analysis beneficial for real-time systems. Also, simplified processor pipelines coupled with high-performance interconnects often result in low power utilization that is beneficial in high-performance systems. While suitable in many ways, these architectures are not without their own challenges. Reliance on shared memory and the strain that massive multi-core processors can put on memory controllers represent a significant challenge to predictability and performance. Resilience is another concern when using NoC processors due to decreased fabrication size.

The aim of this work is to overcome the efficiency, predictability, and resiliency challenges present in massive multi-core architectures. We present new approaches to improve on each of these aspects.

1. First, this work contributes a new real-time paradigm called Forte designed to overcome resilience and predictability challenges. In Forte, we eliminate shared memory use and replace it using TDMA-based explicit communication for increasing predictability. We then extend the traditional task model to introduce the notion of multi-mode data streams to support redundant job execution. This is used to explicitly check coherence ranges of concurrent jobs. Coherence is ensured by protecting against single event upsets and enabling the use of data rejuvenation to increase the mean time to failure of the overall system.

2. Second, this work identifies switch and link-contention as a major challenge to real-time predictability. In this work, we improve upon the standard paradigm by introducing a temporal framing abstraction that uses contention-aware TDMA-like frames to bound communication within real-time tasks. This eases the analysis for determining task-

to-core mappings that aid in decreasing communication contention. We provide two solvers, an optimal and a multi-heuristic solver to quickly and efficiently model inter-task communication and to choose the best layouts that improve predictability of the system. We also provide an automatic framework for evaluating the effects of temporally framed communication on an actual NoC processor using the underlying hardware for explicit message passing.

3. Third, we extend the previous research into a new message-passing library based on MPI by fundamentally designing it for hardware-based network-on-chip strategies. This work compares real-world high-performance benchmarks and shows true application-level scalability limits of shared memory on processors with significant core counts. We then put forth a phase-based optimization strategy that optimizes communication of NoC architectures by eliminating the use of flow control and show significant improvements when compared with current state of the art message passing implementations for network-on-chip processors.

This dissertation puts forth the assertion that taking advantage of message passing on NoC processors increases predictability, resilience, and efficiency of systems.

Bringing Efficiency and Predictability to Massive Multi-core NoC Architectures

by
Christopher J. Zimmer

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

BIOGRAPHY

Christopher John Zimmer was born on May 28th, 1982 in Worthington, Minnesota. He attended Naples High School in Naples, Florida and graduated in 2000.

He studied Computer Science at Florida State University receiving a Bachelor of Science in 2004 and a Master of Science in 2006.

He is married to his wonderful wife Antonia Zimmer and the two are parents to a precocious little girl who goes by Emmerson.

ACKNOWLEDGEMENTS

I would like to thank Frank Mueller, to his credit over the course of four years he tried and tried to teach me to write a proper sentence. In the end upon giving up teaching me how to write, we instead decided to focus on technical challenges. Your open-door policy and availability far exceeded what I could have expected as a graduate student. Your guidance was invaluable and our disagreements were constructive. Thank you again.

Antonia, thank you for your patience, I don't know how many times I fretted that I wasn't going to succeed. It didn't matter, you always patted me on the back and placated my irrational anxieties. You're a spectacular partner in this walk through life.

Mom and Dad, your support was unwavering, it was all I could have asked for. You believed in me even when I didn't, sappy I know but astonishingly accurate. Thank you so much.

Emmerson, thank you for holding my hand and insisting I go down the slides with you. You have no idea how many bad days that made better.

Fellow inmates, Abhik, Yongpeng, Xing, David, Arash, and James. Keep up the inane banter, it is paramount in maintaining sanity in this stressful environment. However, take the soccer ball outside for once.

Finally, I would like to thank Robert Getka, he inspired competitiveness in problem solving that moved Computer Science beyond just an interesting elective for this student.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

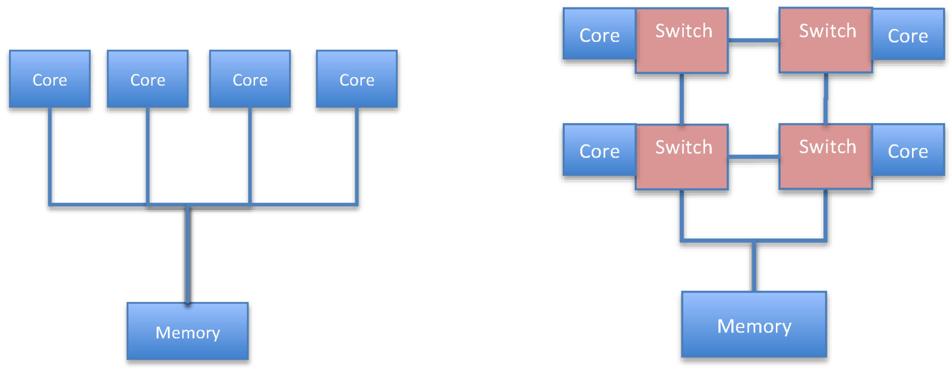
Chapter 1

Introduction

1.1 Network-On-Chip Architectures

Systems architecture is in a constant state of change. The frequency wall [23] has led to the use of multi-core processors and major changes in software architectures. Prior to multi-cores, software threading using time-slicing or cooperative techniques only had to consider memory access from a single processor. With the advent of symmetric multi-cores, interconnect contention increased due to cache coherence traffic. This change affected several areas of computing. In general, greater contention resulted in overheads that were offset by improved performance, particularly in applications that could take advantage of extra hardware pipelines. Nonetheless, interconnect contention led to perturbation that was unacceptable in many real-time systems. However, continuously rising processing demand has required the development of new interconnects and memory layouts.

In this work, we focus on the network-on-chip (NoC) interconnect architecture. NoC borrows its design from traditional networking, which has been meeting demands for increased scalability for years. This interconnect uses packetized transactions and switching hardware to replace the traditional system bus. Figure 1.1 abstractly depicts the rudimentary differences between a 2D mesh NoC and a traditional bus. NoCs provide a new level of flexibility in laying out interconnects. NoC designs include trees, meshes, torus, and rings each, offering their own benefits. The most significant difference in this new design is the addition of a switch coupled with each core as in Figure 1.1b. This switch is responsible for many improvements. In generic bus-based systems, the bus may have to be locked during a given point of time to allow a single node access to the bus. This results in time slicing and round robin arbitration techniques for bus access. In a system containing switches, all connected nodes can interface with the bus immediately through their switch. If a path from source to destination exist, the switches will handle communication agnostically of the pairs of cores. This means that for most cases, there is no waiting for bus access. Instead, NoC accesses result in non-uniform latencies caused by hop-count variations and, more importantly, contention-based delay experienced at the switching level.



A Front Side Bus B Network on Chip

Figure 1.1: Abstract Bus Architectures

A major challenge that NoC designers face today to accommodate the legacy of shared memory software. To be precise, when this work refers to shared memory, it is referring to the abstraction created in software that implements the notion of data sharing between threads/processes. This abstraction requires dedicated hardware resources in order to work properly. Cores are designed around large cache hierarchies and the NoC is optimized to perform cache-to-cache transfers. These features require hardware for tracking coherence states of shared data. A widely used coherence protocol, MOESI, maintains five states for tracking shared data.

- Modified, which means the data is dirty but the local copy is correct and exclusive;
- Owner, which means the cache line is shared but its home is the local cache, containing clean data while shared copies are dirty;
- Exclusive, which means the data is clean in both other caches and memory;
- Shared, which means the data is clean in shared caches but not in memory;
- Invalid, which means the cache line does not contain a valid copy of the data.

Often, multiple networks exist to separate message passing, memory, I/O, and coherence traffic and a memory network to increase bandwidth. A separate coherence network avoids clogging the memory network with data status updates. Consider a large multi-core containing 100 cores running a single application. Cache access hops in the NoC play a large role in the design as architects want to limit expensive long distance updates. In the 100 core case, a worst case access could require up to 36 hops through the network to request and receive the desired value. Due to this cost, a generic design may consider using local cache-line copies to avoid remote cache accesses for each read. This results in a separate, more expensive coherence network.

Additional costs in using shared memory in NoCs come through the necessity of synchronization to avoid race conditions or data hazards such as atomics, barriers, or locks. The problem is exacerbated when we consider that we are no longer synchronizing 2 or 4 cores but rather 64 to 100 cores. System designers put great effort into designing system-wide high-performance techniques for improving synchronization. These techniques may include implementing systems for locking synchronization objects into a single cache and optimizing native atomics to improve access times to this data. Other techniques may involve vendor-provided libraries to best take advantage of the hardware interfaces. Unfortunately, all of these techniques still result in limited scalability for legacy code.

As the core counts increase, current system and programming abstractions, such as pure task-level parallelism, become an obstacle rather than an aid in harnessing multi-core power. Design trends in NoC lend to software reuse through shared memory. Unfortunately, limitations of shared memory are already proving to be the next performance bottleneck in existing systems containing 32 and 64 cores. Next generation NoC architectures are being designed to contain hundreds (if not thousands) of cores to meet processing demands. Fortunately, NoCs also offer a novel paradigm of on-chip message passing. The focus of this dissertation is identifying techniques for taking advantage of on-chip message-passing. In this work, we focus on real-time and high-performance computing.

1.2 Real-Time Systems

Real-time systems are a technology domain in which the design of the system focuses on guaranteeing temporal correctness. This means that real-time systems focus on the ability to guarantee that running software will finish within precise timing thresholds. This differs from traditional systems where the primary metric is performance. Real-time systems are often obfuscated from view but enable important functions that humans rely on. An example can be seen in anti-lock braking systems, which constantly monitor our car brakes and must react within certain amounts of time to guarantee that the brakes do not lock up. Real-time control systems are ubiquitous and increasingly demand new technologies to meet increased requirements.

A real-time system is traditionally a multi-tasking system relying on a schedule of periodic tasks where each instance is referred to as a job. For each task in the system, we maintain strict information, such as the tasks phase, period, execution time, and deadline. There are two categories of real-time systems based on the implications of systems correctness for deadline misses. *Soft* real time indicates that a deadline can be missed without rendering the system incorrect. However, the later a result is provided after a deadline, the more likely it is to reduce the systems quality of service. *Hard* real time indicates that a miss of a deadline results in an incorrect system. Hard real-time systems encompass a class of systems known as safety critical. In such systems, missing a deadline could result in damage to the environment and even loss of life.

1.2.1 Definition of a Real-Time System

A periodic real-time system is defined as a set of tasks where each task is represented by $\tau =< \phi, p, e, d >$.

1. ϕ represents the phase of a task or the release time,
2. p represents the period of a task, i.e., the minimum time between job releases,
3. e represents the worst-case execution time (WCET) and
4. d represents the deadline, i.e., the time instant by when the job must be finished.

Compositions of real-time tasks are analyzed using schedulability analysis to ensure that each job in a system can execute within their respective deadlines.

1.2.2 Timing Analysis

Accurate knowledge of execution time is a strict requirement for hard real-time systems where a missed deadline may render the entire system incorrect. Timing analysis determines an application's best-case execution time (BCET) and worst-case execution time (WCET) bound that allows verification if a task's deadline can always be met. Timing analysis can be performed via dynamic techniques [14, 72], static techniques [76, 45] or hybrids of them [12, 44, 75]. Dynamic timing analysis determines the effect of different inputs on execution time to approximate the WCET, *e.g.*, to determine that an inversely sorted list maximizes bubble sort's computational complexity. Static analysis bounds aggregate costs of instructions in blocks and then compounds the costs of paths throughout the program taking architectural timing effects into account to provide a safe WCET bound at compile time. In contrast to the dynamic approach, static timing analysis has been shown to provide *safe* WCET bounds [72]. The challenge with static analysis is that it must model systems precisely for safety. This task is well studied for single core and single memory controller systems but the addition of multiple cores, memory controllers, and new interconnect techniques adds complexity to static analysis.

1.2.3 NoC Real-time Systems

Massive multi-core architectures contain features that may be beneficial to the domain of real-time systems. Commercial off the shelf (COTS) components can be used to reduce development time. Processor cores are utilizing simpler designs to increase power efficiency that also lend well to simplified modeling for static analysis. Finally, NoCs contain abundant and cheap processing resources that can relax the need for sophisticated schedulers and lead to simplified schedulability analysis, particularly when the number of processors exceeds the number of tasks in the system. This fits well into the paradigm laid out in [21] where the concept of dark silicon is introduced. Dark silicon is the notion that we are quickly reaching the threshold in which we can possibly power and cool all of the cores on a single die. Instead of powering all of them, each core will be bound to a single task. It will be the responsibility of the scheduler to execute tasks through core activations and deactivations.

1.2.4 Challenges of Massive Multi-core Real-time Systems

Today's NoC processors can contain up to 100 processors on a single die/socket with 4 memory controllers. Over-provisioning these memory controllers and the cost of non-uniform memory latencies can have adverse effects in terms of predictability. Decreases in fabrication size increases the need for software based resilience for these control systems to operate safely. Task placement in NoC architectures can lead to increases in worst-case behavior due to inter-processor communication contention. As such, task-to-core layout is an important factor in implementing real-time systems on these processors.

1.3 High Performance Computing

High performance computing (HPC) utilizes a class of applications that can generally be defined as highly parallelizable. A major challenge facing the HPC environment is powering and cooling large machines. NoC systems offer interesting architectural features because these processors contain low power processing elements and high-speed interconnects. Another major benefit of these architectures is that they contain both message passing and shared memory features. This allows for significant reuse of HPC codes without having to fundamentally redesign algorithms.

1.3.1 Multi-Computer vs NoC

There is significant crossover in terms of problems and solutions that pertain to multi-computers and NoC processors. Both are tailored toward solving highly parallel problems and in both message passing enables the scalability necessary for efficient computations. However, differences exist that separate these systems. The most significant is that NoC message passing is architectural in nature. Instead of requiring multiple software hierarchies involving operating systems and network interface cards, NoCs facilitate message passing through ISA instructions and register to register transfers. This subtle difference eliminates software overhead and results in low latencies. However, this level of access may result in deadlocks within the NoC or pollution of the network with malformed packets.

1.3.2 High Performance Challenges

The major challenges that HPC faces with NoC is in defining which inter-processor communication paradigm to use and determining how to take advantage of on-chip message passing in a manner suitable to the software algorithms. Traditional HPC software is written in a manner in which deadlock and flow-control considerations are abstracted from the designer. Unfortunately, current generation NoCs are designed for throughput while maintaining flow control and avoiding deadlocks are burdens placed on the designer. To properly take advantage of message passing abstractions, portable techniques for message passing IPC must be designed without limiting system scalability.

1.3.3 Contributions

This dissertation assess the impact of shared memory in both real-time and high-performance applications.

1. In Chapter 2 [84], we assess the impact that shared memory has on scalability and use this to motivate a new task paradigm for highly reliable real-time systems. In this work we present Forte, a real-time task system based loosely stream data flow applications. Forte breaks task computation into logical parts and promotes message passing for inter-process communication. By harnessing the increased scalability attained through message passing we can directly translate this into a system that enables high software reliability through the use of multiple concurrent task models without affecting temporal correctness when additional tasks are added to the system.

2. In Chapter 3 [85], we consider the impact that link contention has on real-time predictability in systems communicating through message passing. We present a new technique for partitioning communication within a real-time task set to enable analysis. This technique translates into an offline analysis that enables us to exhaustively enumerate the entire search space of task-to-core mappings. This results in a global minimum of link contention. We then perform analysis indicating that global minimum techniques are unable

to scale to current industry size and present a series of heuristics for quickly converging on solutions that reduce contention. To improve upon local minima often associated with greedy algorithms, we apply a multi-heuristic approach to improve solutions. Results show a significant reduction in link contention across our generated task sets when compared with naive layouts.

3. In Chapter 4, we extend our work to the performance domain and consider how the effects considered in the previous work extend to high performance applications. We present a new framework, NoCMsg, for message passing that serves two purposes. First, it borrows semantics from MPI, a well established message passing API, to take advantage of a large amount of code already written in MPI. Second, we abstract out considerations of the hardware from the user such as flow-control management. In this work, we present a technique based off hardware analysis for circumventing deadlocks in NoC switches without having to use mixed protocol messages with interrupts as was done previously. We then identify communication patterns existing on both point-to-point and collective-based communication that enable the relaxation of any type of flow control to gain performance. Our results compare NAS Parallel benchmark codes and TF*IDF using OpenMP and Opera MPI, they show significant performance improvements when using NoCMsg.

1.3.4 Hypothesis

Processor technology has reached the scalability limit of existing on-chip resources. This will affect the advancement of several computational domains. With the increasing demands of both high-performance and real-time computations, techniques must be devised to avert the limits of scalability. These limitations are claimed to be caused by inter-processor communication. Current IPC techniques suffer from

1. **coherence state distribution and delays,**
2. **request/response based IPC for all requests, and**
3. **contention at cache and memory controllers.**

Shared memory is the fundamentally limiting technology for scalability of applications exhibiting significant inter-processor communication.

The hypothesis is as follows:

On-chip message passing further improves resilience, predictability, scalability, and performance of applications running on large-scale multi-cores.

1.3.5 Organization

Chapter 2 describes our work in a new programming model for these architectures that eliminates traditional reliance on shared memory and enables fault tolerance by taking advantage of abundant processing resources. In Chapter 3, we address the effects of network contention on predictability by introducing a new communication abstraction and techniques for generating low contention task-to-core mappings for large NoC-based real-time systems. In Chapter 4 we develop a new message passing framework built specifically for NoCs to ease design and improve performance. In Chapter 5, we conclude and discuss future plans for this work.

Chapter 2

Network-On-Chip Predictability Challenges

2.1 Introduction

ASIC-based cyber-physical systems are costly to design in terms of time and money. Multi-core COTS processors are becoming increasingly used in the high-end handheld market and are also seeing increased use in the lower-end embedded control market. An example is the Freescale 8-core PowerPC P4080 that is being marketed in the power utility domain for control devices. In such processors, traditional software design techniques coupled with increasingly smaller transistor sizes can negatively affect the real-time predictability and the fault reliability. Predictability challenges in multi-cores are due to non-uniform memory latencies [46] as contention on buses and mesh interconnects increases.

Another trend is an increase in transient faults due to decreasing fabrication sizes. These faults surface as single event upsets (SEU) that can render computation incorrect. SEUs are faults that can modify logic or data in systems leading to incorrect computational results or software system corruption, which can result in temporary or even permanent incorrect actuator outputs in control systems if not countered. SEUs have three common causes: (1) Cosmic radiation, particularly during solar flares, (2) electric interference in harsh industrial environments (including high temperatures, such as in automotive control systems) and (3) ever smaller fabrication sizes and threshold voltages leading to increased probabilities of bit flips (for all) or cross-talk (for the latter) within CMOS circuitry [18, 71].

For example, the automotive industry has used temperature-hardened processors for control tasks around the engine block while space missions use radiation-hardened processors to avoid damage from solar radiation. An alternative approach is taken by commercial aviation. The latest planes [53] (Airbus 380 and Boeing 787) deploy off-the-shelf off-the-shelve embedded processors without hardware protection against soft errors, such as the PowerPC 750. Even though these planes are specifically designed to fly over the North Pole where radiation from space is more intense due to a thinner atmosphere, processors deployed on these aircraft lack error detecting/correcting capabilities. Hence, system developers have been asked to consider the effect of single-event upsets (SEUs), *i.e.*, infrequent single bit-flips, in their software design. In practice, future systems may have to sustain transient faults due to any of the above causes. COTS architectures are not specifically designed for real-time fault tolerance and contain few hardware-based fault mitigating mechanisms, such as processor radiation hardening. Previously, researchers have designed techniques to mitigate SEUs in software using task scheduling [29, 24, 17]. This often leads to sophisticated scheduling techniques utilizing alternate algorithms, re-execution, or replication. In contrast, we argue that massive multi-cores with NoC interconnects greatly simplify scheduling and allow high levels of replication.

In a system with replication of entire task sets under the traditional shared-memory model, considerable strain is placed on memory controllers due to compounded memory pressure and coherence traffic resulting in contention. This contention limits scalability and reduces predictability of advanced multi-core architectures. In spite of the potential drawbacks, multi-core COTS processors remain quite attractive for real-time systems. For example, ARM promotes "dark silicon" each real-time task is mapped to a separate core as cores are plentiful [21]. Scheduling then amounts to simple core activation thereby eliminating context switching costs and preemption delays. Such an abstraction also facilitates parallel, replicated execution and voting in n-modular redundant environments to increase reliability.

Contributions: This work introduces a Fault Observant and Correcting Real-Time Embedded (Forte) design for large multi-core architectures with NoC interconnects. The detailed contributions of Forte are as follows: (1) Forte provides a task abstraction framework that takes advantage of *message passing* capabilities implicit in NoC systems to *eliminate the use of shared memory*. Forte thus *increases the overall predictability* of the system as contention on the memory controllers is reduced. Furthermore, Forte improves *scalability* for contemporary mesh-based NoC architectures. (2) Forte *improves reliability* by provisioning simultaneous task models of varying complexity and measuring the strength of association (coherency) to *facilitate voting* in a modular redundancy scheme. (3) Forte further ensures sustained reliability by enabling fine-grained task rejuvenation. This includes the ability to replace faulted data models and to refresh rejuvenated tasks to align redundant models. Such rejuvenation is critical particularly for long-running or 24/7 control systems. Experimental results of Forte with a cyber-physical flight control software show improvements up to an order of magnitude in overhead reduction over standard shared memory implementations, reduced jitter and scalability. Reliability is improved in line with results reported for modular redundancy. Yet, Forte sustains these reliability levels through rejuvenation, which significantly increases reliability as opposed to a scheme without rejuvenation, as shown in experiments.

The remainder of this paper is structured as follows. Section 2 presents the design of our proposed framework. A case study developing an unmanned air vehicle control system is detailed in Sections 3 and 4. Section 5 provides the experimental framework. Section 6 presents experimental results. Related work is discussed in Section 7. The paper is summarized in Section 8.

2.2 Forte Design

This section provides an overview of the Forte framework to exploit massive multi-core processors to facilitate highly redundant cyber-physical systems. Careful use of this technique can improve system integrity in the form of protection from soft errors by providing a framework for running multiple concurrent versions of a task, called **shadow tasks**, and verifying their output coherence. The framework assumes that each task is permanently assigned to a unique set of cores and that the number of tasks in the system is less than the number of cores. The scheduling system is periodic with dynamic priorities based on relative deadlines. (Notice that scheduling amounts to activation/deactivation of tasks as only one task may be assigned to a core in our abstraction. Hence, we honor the periodicity of real-time tasks, yet scheduling becomes trivial as preemption never needs to occur.) Figure 2.1 depicts our model of a massive multi-core NoC processor. Our sample processor model contains 64 processing elements connected in a mesh grid. Each processing element contains a switch so that network communication and routing can be handled without additional overhead to the processing pipeline. NoC processors often support both static and dynamic message routing. Due to this, our framework operates agnostic of the underlying message passing API.

Forte capitalizes on the additional processing elements available in advanced COTS processors to run multiple simultaneous system models. These models can vary in feature set and complexity, extending the model from the basic requirement, to a model with more precision/features to increase the system efficiency. We use the standard notation of ϕ, p, e, and d to denote phase, period, execution time, and deadline of a real-time task [41]. Using terminology from [59], we group the functional models and order them via complexity ranging from the most complex features to the "simple" baseline model. For example 1, consider two tasks $c =< \phi_c, p_c, e_c, d_c >$ and $s =< \phi_s, p_s, e_s, d_s >$, where c and s perform the same system function but c is a complex version of s, the baseline version. In Forte, we assert that $p_s = p_c$, since they provide the same system function, though e_c may be larger than e_s. We further assert that s and c generate output data where a coherency range can be determined. For additional redundancy, we consider two more tasks, c_{shadow} and s_{shadow}. These tasks are added to the system as mirror images of c and s, operating on the same data to validate the correctness of

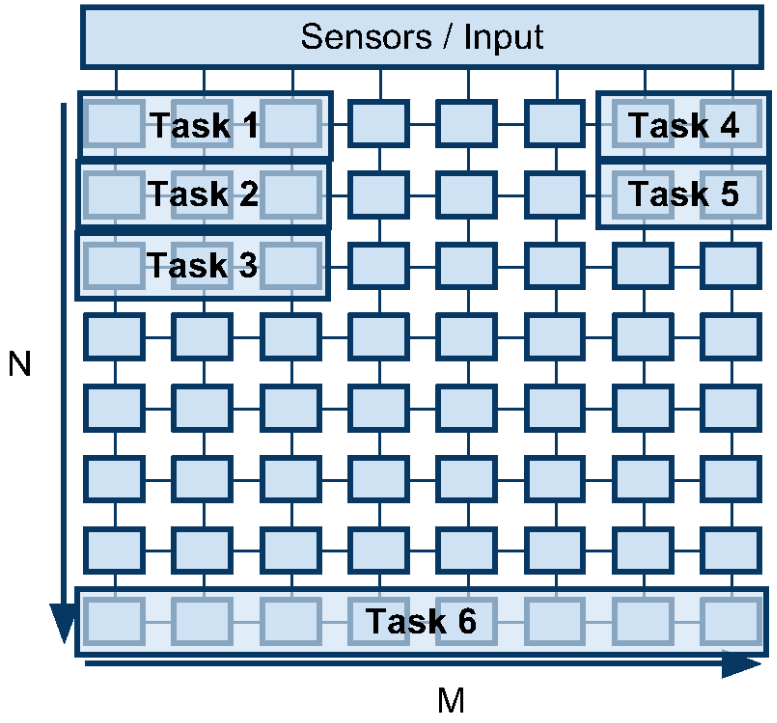


Figure 2.1: Forte Task Layout Over Cores

each model's output data. To formalize the framework, we extend the classic task model such that:

$$\tau =< I, O, T, C, R > \tag{2.1}$$

- I is the set of inputs i for the m shadow tasks in T;
- O is the set of outputs o of τ that must be validated for coherence prior to allowing the output change on the system to take effect;
- T is the sequence of shadow tasks $< t_1, t_2, ... , t_m >$ where each element t_i in T is ordered by a descending complexity coefficient k_i such that $k_1 \geq k_2 \geq .. \geq k_m$;
- C is the set of coordinates $< x_1, y_1 >, < x_2, y_2 >$ that enable the system to bound τ to a specific core within the architecture;
- R is the set of data within a task that must be transferred to a rejuvenated task to ensure convergence. If natural convergence is used $R = \emptyset$.

The Forte framework characterizes each task within the system with a set of inputs and outputs. Figure 2.2 depicts a Forte task where the input phase splits the data so that three redundant tasks of f can operate on separate models of the input data in parallel. We use the term f to describe the defining function(job) of the real-time task. When a task finishes execution it then sends the outputs to a coherence check that determines the correct output for the system. These sets allow tasks to execute independently or to be chained together to facilitate data flow within the system. Figure 2.3 depicts how the various tasks communicate data without using shared memory. The model forms an abstract chain: Once a task generates output data, its output data becomes the input data for a subsequent task. This model can be implemented on NoC architectures through explicit message passing.

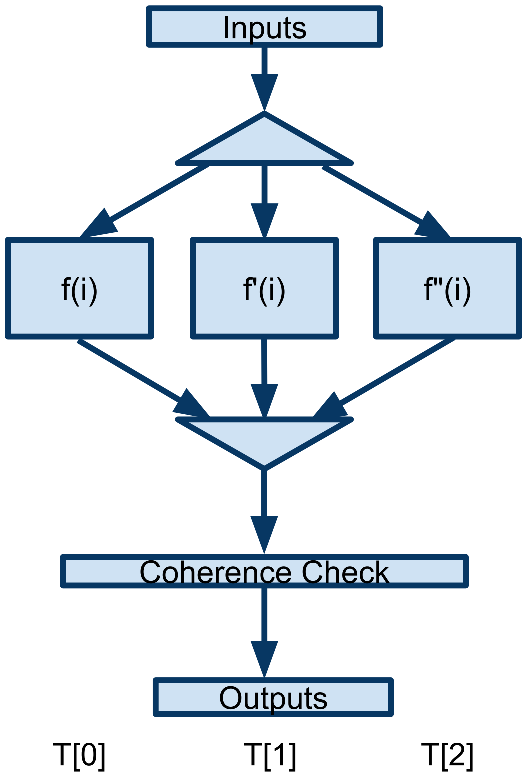


Figure 2.2: Abstract Task Layout After Splitting

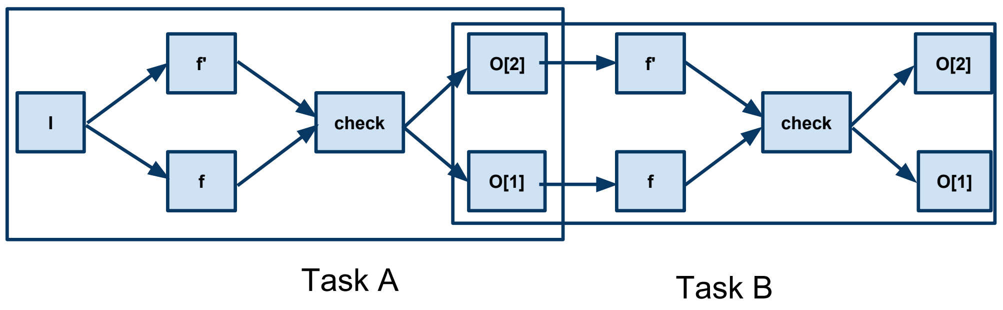


Figure 2.3: Task Chaining

2.2.1 Shadow Tasks

Forte is designed to exploit the high-level of concurrency that NoC architectures provide. Cyber-physical systems deployed in harsh environments are subject to *Single Event Upsets (SEUs)*. These are compelling reasons to utilize the multi-core paradigm and generate several models of a single task called shadow tasks, which improves the level of data integrity of the system. In the previous example from this section, c,c_{shadow},s, and s_{shadow} are considered shadow tasks of a single system level task τ. In Forte, shadow tasks are represented in a complexity ordered list. To state this more formally, for each shadow task t_i in T, there is a complexity coefficient k_i, such that

$$< t_i, t_{i+1}, ..., t_m > \implies k_i \geq k_{i+1} \geq ... \geq k_m \tag{2.2}$$

The complexity coefficient k_i is best generalized as a scoring value generated by deriving less precise real-time task models from the most sophisticated design. A degraded complexity model for real-time systems was put forth in [59]. In this work, a complex and a simple feature set for a given control task helped to increase the model safety. Deriving a score for k_i considers effects of a reduction in features and reductions in data precision or utilization for faster converging algorithms with a larger tolerance range ϵ.

2.2.2 Input

Real-time tasks have a variety of data models that can be supported in the Forte framework. Referring back to Figure 2.1, task 1 acquires input from sensors or other I/O devices that are not part of the task set. Task 2 derives input from task 1 and task 6 operates independently or receives input from a device that is pinned to the lower portion of the core layout. Supporting an abstract input set allows the framework to be flexibly used to deploy a variety of real-time tasks. Forte considers multiple data streams separated by complexity, shown in Figure 2.4. Streams enable shadow tasks of varied complexity to ensure that data is not unnecessarily losing precision by forcing a single stream of data.

In practice, input acquisition is a precondition for each task in Forte. If the input is derived from a sensor or other external hardware, it requires one of the shadow tasks to acquire the data and then distribute the data over the message passing network. If the input is derived from the output of another task, each of the shadow tasks must receive their input from a proceeding output of equivalent data complexity.

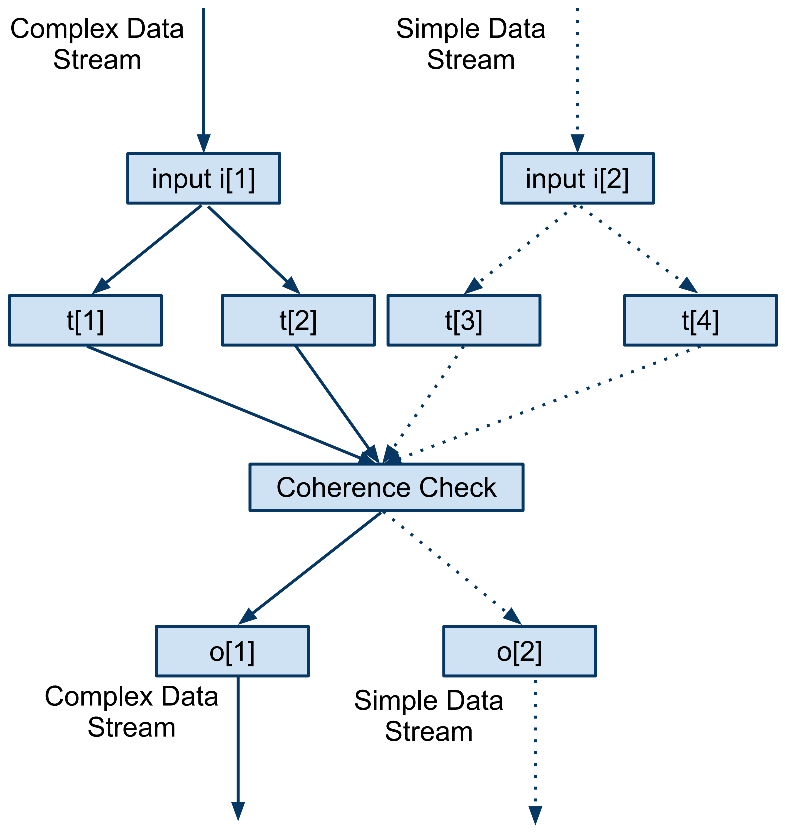


Figure 2.4: Data Stream Abstraction

2.2.3 Output

Forte improves integrity by validating the coherency of each shadow task's data. A potential but undesirable result of this coherency validation is that the designer may have to reorder the code in control tasks to defer a decision until the shadow task decisions can be verified. Coherence formulations are determined by the system designer. Automatically identifying how to determine these is algorithm specific.

For a given task set, each shadow task operates on local data sets. Upon completing the necessary computation, the data is checked by the coherence-checking phase of the task. This may be performed by every shadow task or in a subset of them to reduce the data transfer cost. In the coherence check in Figure 2.4, shadow tasks $t[1]$ and $t[2]$ are of equal complexity and the data must match exactly. The same holds for $t[3]$ and $t[4]$. When this verification is complete, a range check is performed to validate that the data in the complex and simple streams are within a preset range. Certain features of the complex stream may not exist in a lower precision model. This makes it important to maintain multiple checks for each level of complexity. Successful coherency checks result in the mapping of output data to locations designated by complexity. This allows subsequent tasks dependent on this output to be mapped to the data of matching complexity. If the coherency checks fail, the failing task can be isolated to remove any impact it may have on the control system. If the failure is within the highest complexity model, subsequent shadow tasks that operate on that model can be canceled, allowing the system to rely on the less complex data models. If it is a lower complexity model that sustains the failure, data of the higher complexity models can often be filtered to allow a lower complexity model to continue operation. This output data flow is shown in Figure 2.4. The result of the complex data stream is filtered into the simple data stream in this case. When using fine grained coherence checks in a n-modular redundancy configuration rejuvenation can be used to repair the faulting task.

The formalization of input/output sets also supports feedback control loops. Forte allows data within the output set to be specified in the input set of subsequent tasks. This formalization supports task chaining. Feedback loops are supported as a chained loop of multiple tasks or the redirection of a single task's output back into its own input.

2.2.4 TDMA

Contention can hinder performance on message-passing networks, *e.g.,* when multiple fault models transmit their data to coherence checks in the Forte system. Tasks could potentially overwhelm routers or their buffers with adverse affects on performance. Forte addresses this problem by arbitrating the underlying NoC network through Time Division Multiple Access(TDMA). TDMA makes Forte more predictable by reducing contention on the message-passing network and facilitating the bounding of worst case behavior for all message-passing phases. TDMA isolates core communication into global window frames. Any particular core is only able to transmit data during its predetermined frame. Using TDMA across all cores effectively allocates all links within the NoC to sender during their frame, guaranteeing that no two cores contend for a link during any period.

2.2.5 Task Rejuvenation

Real-time control systems are developed to run for extended periods of time if not even 24/7. They may thus be exposed to multiple event failures over the course of their lifetime. Single event upsets are handled through coherence voting and elimination of the faulty data. A subsequent second or third event upset to one of the remaining redundant task may leave the system without decision capability as to which the correct results is. The objective of rejuvenation is to correct the faulting model to ensure that resilience of the model is sustained. According to a study from the high performance domain [19], as devices advance and die sizes decrease, the projected failures per hour for a single node in an HPC system is $4.1x10^{-7}$. Another study [15] from the satellite domain using a hardened COTS multi-core device evaluates the failure rate as $2.2x10^{-4}$ failures per hour. Both studies indicate that the probability of multiple-event upsets in a short time period is low. But if the runtime of the system is long, a second SEU is likely.

Forte addresses this challenge by supporting fine-grained rejuvenation as a part of the framework. Fine-grained coherence checks allow failing tasks to be identified. In Forte, an SEU is confined to a single task that is considered to have failed since tasks are associated with disjoint cores and do not share memory, i.e., only the failing task needs to be terminated. Subsequently, one of the remaining correct tasks supplies its output as input data to subsequent tasks of the terminated one during its rejuvenation. This is implemented as follows. The scheduler terminates the faulting task and creates a rejuvenated version of the task on the same core starting with newly initialized data values. The rejuvenated task is not caught up in its data output after such a restart and would fail the coherence check as thresholds would be exceeded. The coherence check is therefore temporarily relaxed to only validate the outputs of the remaining tasks (ignoring the rejuvenated one).

Coherence validation via voting is deferred until the rejuvenated task converges with the correct models in terms of its output. Many control algorithms exploit convergence algorithms in feedback loops to guarantee stability, i.e., they will naturally converge over a period of time if the output is dependent on the input. In other cases, running state is maintained between each job invocation of a task so that models do not converge by itself. Here, the state of one of the remaining (correct) tasks is utilized to allow the rejuvenated task to catch up. Forte supports data refreshing of rejuvenated tasks as follows. A correct task is designated by the coherence module to refresh a rejuvenated task with local memory values specified during system design. These memory regions are transferred to the rejuvenated task in between job invocations to assure consistency. Data refresh is a requirement for non-converging algorithms. But it can (and often should) also be utilized to more quickly catch up with the correct tasks for converging algorithms. This reduces the vulnerability window to receive another SEU while operating under degraded redundancy (e.g., dual redundancy) during rejuvenation. After data refreshing (or convergence without refresh), the coherence validation can reactivate voting again upon reception of outputs from the reborn task within thresholds.

2.3 UAV Application

The next two sections describe our experimental implementation of the Forte design using a cyber-physical control system. This section describes the control system and its tasks. The next section describes the changes necessary to move the control system into the Forte framework. To evaluate the design, we selected Paparazzi [49], a traditional shared memory real-time control system. Paparazzi is an unmanned air vehicle (UAV) control software. We ported it using the Forte design framework and evaluated it on a hardware NoC architecture. Our port of the Paparazzi control system is based on a java implementation [33] that we rewrote in C++. Paparazzi is structured as two separate sets of real-time tasks that enable a switch between manual control of the aircraft and autopilot mode. These modes are detailed as Fly-By-Wire (FBW) and Autopilot (AP). The basic structure of Paparazzi allows only the FBW mode to control the servos. However, when there is no pulse position modulation (PPM) control, the autopilot mode sets the actuation by controlling the values that the FBW mode uses to control the servos. This relationship is detailed in Figure 2.5.

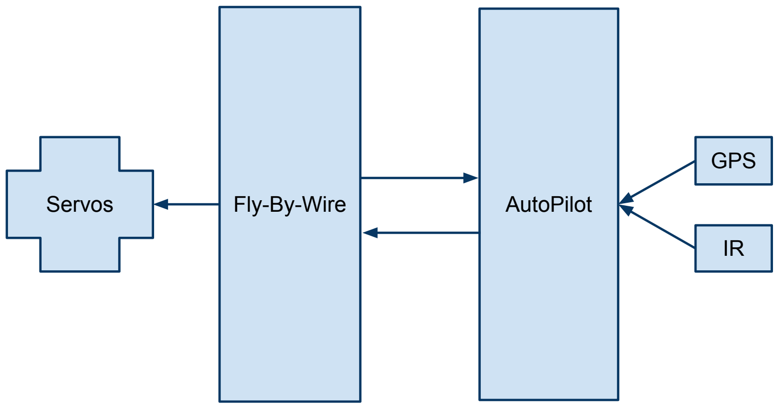


Figure 2.5: Paparazzi Design

2.3.1 Paparazzi Autopilot-Base Design

The basic design of the shared memory version of Paparazzi uses several shared objects accessed various tasks to calculate vectors to control the UAV. This information consists of a navigator, estimator, and a flight plan. The following paragraphs will briefly cover each task and how it operates on these shared data structures in order to illustrate later how to redesign for a message-passing framework. The basic task layout for the auto pilot module with task dependencies and data flow are shown in Figure 2.6. **Navigation Task:** The navigation task is responsible for taking information from the GPS device, determining the current location of the UAV and then storing the values into the estimator data structure for later tasks that cannot read the GPS data. It then compares this information against the flight plan and determines target metrics for the UAV to meet the flight plan. **Altitude Control Task:** The altitude control task is responsible for determining the control values to reach/maintain the desired UAV altitude. It first ensures that the system mode is set to allow autopilot control. It then obtains data from the estimator's z coordinates and determines the error from the desired altitude. It then uses this error factor to determine any corrections and commits them to one of the shared memory objects. **Climb Control Task:** The climb control task is responsible for determining the system's output in terms of thrust and pitch in order to maintain the necessary altitude. It takes as input the altitude determined in the altitude control task and the z directional speed vector determined in the navigation task. It uses these inputs to calculate the necessary pitch and thrust to control the altitude of the UAV's vertical changes. **Stabilization Control Task:** The stabilization control task uses data from the infrared (IR) device, the climb control task, and the navigation task. This task is responsible for determining the roll and any changes to the pitch. The stabilization control task in this implementation is also responsible for transferring the data to the FBW task that updates the actuation on the servos. The data sent is the pitch, roll, throttle, and gain to control the servos. **Radio Control Task:** This task takes the last radio control command from the FBW module and stores the data in the autopilot in case it needs to take over control.

2.3.2 Fly-By-Wire Base Design

The Fly-By-Wire(FBW) task set is used to control the servos and to take control from the ground control unit, the latter of which is not exercised in this implementation. The task layout of the FBW module is shown in Figure 2.7. **Pulse Position Modulation (PPM):** The PPM task receives the radio commands from the PPM device and uses them to control the servos of the UAV if the autopilot mode is not enabled. **Transfer to Autopilot:** This task takes the message retrieved from the PPM device and transfers it over the systems designated bus to the Radio Control Task. **Check Fail Safe Mode:** This task controls whether the auto

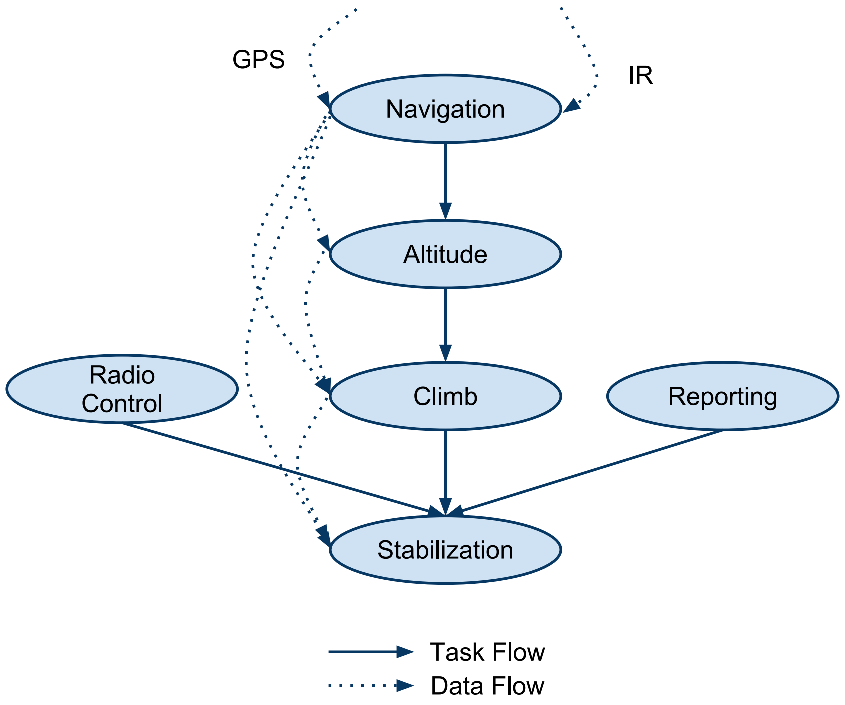


Figure 2.6: Auto Pilot Task and Data Flow

pilot or the PPM device is controlling the UAV. It validates several device-based metrics to determine if the device is still receiving signals from the PPM device or if a fail-safe mode has been activated. **Check Auto Pilot:** This task controls the servos based on data received from the AP. The task receives data from the stabilization control task over the systems specified bus and then transfers these control values to the servos for actuation. **Flight Model and Simulated Devices:** In order to function appropriately Paparazzi requires a GPS device, IR device, and a functional flight model. The Flight model specifies flight dynamics based on the rudimentary version found in the Paparazzi open source code. The GPS device infers several metrics based on its current position, its last position and the change in time. The IR simulates a dual axis differential IR device, that uses IR temperature readings between space and the earth to stabilize the roll and pitch of the aircraft. The output data from the IR device is critical in the stabilization task.

2.4 Forte Implementation

The Forte implementation follows the design in respecting the relation between input and output tasks, supporting the fault model of n-modular redundancy with coherence checks on outputs and optional task rejuvenation.

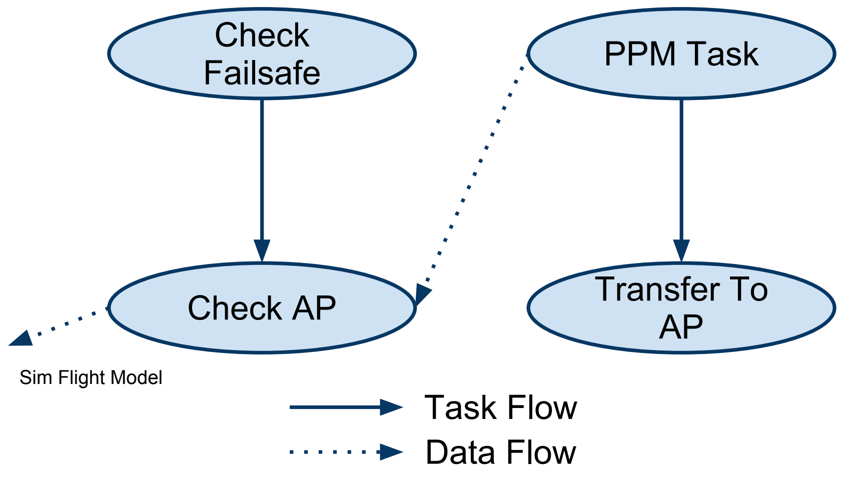


Figure 2.7: Fly-By-Wire Task and Data Flow

2.4.1 Input and Output Tasks

Implementing Paparazzi using the Forte design required analyzing the shared memory accesses that occurred within the task set and expressing them as data-flow relationships between tasks. The original implementation of Paparazzi uses logical objects to store data in containers. This eased programming requirements in that it made the data logically organized. However, it also made all data in these objects globally accessible. While this is suitable for single-core implementations, using shared data in multi-core scenarios adds overhead. We remedied this by transforming data flow relationships to remove shared object containers altogether. They were replaced by data designated in two ways.

First, we utilize local data when data is only operated on within a task. The majority of data in our implementation could be categorized as local data. This contains all temporary variables and most of the state variables that update the primary flight metrics during operation.

Second, we utilize remote data. This data is stored locally but the actual data values originated else-where and are communicated between cores via sends and receives. Remote data values are written to local memory of the task before the task is released. In Figure 2.6, the dotted lines represent the flow of remote data in the auto pilot module.

We then converted each task into Forte tasks. Each Forte task consists of an input phase, a computation phase, and an output phase. The input phase of each task is generic. The task simply receives data and stores it in local memory for subsequent execution. Task computation differs from the shared memory version only in that instead of operating on global containers all data is local to the tasks core. The output phase sends any data to subsequent tasks according to the data flow specifications.

2.4.2 Scheduler

In the introduction of this paper, we made the claim that massive multi-core architectures could ease the problem of task scheduling. Trends in the market indicate that in the near future architectures with tens if not hundreds of cores will be arriving. In the past, processing resources were in heavy contention and sophisticated scheduling techniques were needed to arbitrate access to limited resources. The term limited can no longer be used to describe processing resources for massive multi-core architectures. For the Forte implementation of Paparazzi, the scheduler is a simple periodic scheduler. The scheduler statically deploys each task to its own core where it remains stationary. Taking advantage of the massive multi-core architecture, no tasks shared a core. Scheduling thus reduces to core activation/deactivation to release or terminate a task. Each task would then be set to sleep until it received a NoC-based message from the scheduler core waking it up to perform its task. The impact of the sleep state is significant in terms of power consumption. As the number of cores on these architectures scales up, that ability to power them simultaneously will become a serious challenge. In order to limit the scope of the power consumption of such chips, many chip designers are implementing low power sleep modes with instant-on functionality. This enables software to constantly turn off and on the resources needed while conserving power.

2.4.3 Fault Models

To simplify our experimental implementation, we integrated an n-modular redundancy configuration using the Forte model instead of a Simplex implementation. In our evaluation, we use a triple modular redundancy. This shows the flexibility of the architecture in that can use Forte's design for three completely simultaneous instances of Paparazzi. This enables coherence checks to identify the faulty model in times of failure so that voting can occur to determine which model controls the simulated servos.

2.4.4 Coherence Checks

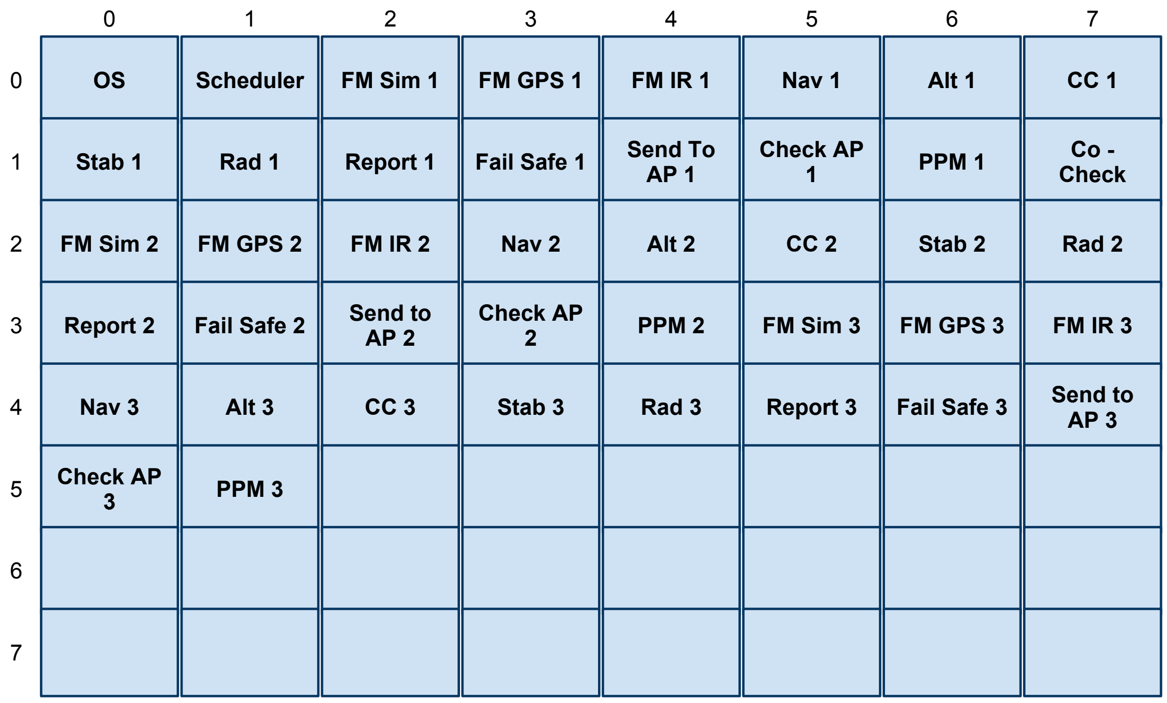
We designed several coherence checks to enable robust fault checking for our Paparazzi implementation. Since our fault model in Forte was designed with redundancy tasks, our coherence checks simply verify data consistency. Each coherence check is designed as a sporadic task that immediately follows the execution of a system task in the Paparazzi suite using precedence constraints. Each coherence task is assigned to a specific system core. When the coherence task receives data from the first model, it sets a timeout in order to not wait indefinitely for the remaining models to transmit their data. When all of the models have transmitted the data, the coherence check validates the data. When there is a validation error, the coherence check uses a 2/3 majority. It determines the failing model and notifies the voting routines to prevent the faulting model from controlling the system servos. When a timeout occurs coherence is checked between the models that did submit data, any models that did not submit data are considered to have failed.

2.4.5 Rejuvenation

Rejuvenation is implemented in Forte in two ways. The feedback control algorithms support natural convergence and, as such, just require a restart mechanism and a warm up phase to re-enable coherence validation. Paparazzi utilizes such natural convergence, i.e., our implementation exploits this restart capability. In addition, rejuvenation with refreshed data was realized as an optional extension. This allow us to compare the time (overhead) for convergence with and without refresh. To facilitate rejuvenation under data refresh, the coherence module uses the message passing network to indicate the source data refresh, i.e., one of the remaining correct tasks (cores). Refresh data is transmitted during the next idle phase to ensure non-interference with real-time deadlines of the correct tasks. The refresh data is also received during the idle phase of the restarted task as redundant tasks are harmonic (not only in period but also in idle phase). Received data subsequently refreshes uninitialized state in tasks, either to ensure that outputs are within coherence thresholds or, as in the Paparazzi example, to speed up convergence amongst the redundant tasks.

2.5 Experimental Framework

Our experiments were conducted on a Tilera TilePro64 development board. This platform features a 64 tile (core) chip multiprocessor (CMP) suitable for the embedded space with lower power requirements [6]. The Tilera platform has been selected for satellite deployment. Tilera processors support both message-passing and coherent shared memory models, and the choice is up to the user. Tiles are connected by multiple meshed NoCs that support memory, user, I/O, and coherence traffic on separate interconnects. Each tile processor is equipped with level 1 caches and split TLB making each core a fully independent processor. For evaluating our framework, we implemented the PapaBench real-time task set from the Paparazzi UAV cyber-physical system. Two implementations were created for evaluating not only the framework's fault resilience but to also compare computational jitter in systems relying on shared memory vs. message-passing. The shared memory task sets follow the proposed model in the paper (but with input and output phases integrated with computation phases of tasks). Figure 2.8 depicts the system layout of the control tasks identified by their abbreviated name combined with their execution identifier *e.g.*, CC2 is Climb Control Task 2 (see Section 2.3 for task identifiers). The figure illustrates the linear task layout across the tiles. This layout is agnostic to the execution models (shared memory vs. message-passing). All experiments using more than two tasks arbitrate access to the NoC using TDMA as described in previous sections. This reduces the impact of NoC effects on the system.



	0	1	2	3	4	5	6	7
0	OS	Scheduler	FM Sim 1	FM GPS 1	FM IR 1	Nav 1	Alt 1	CC 1
1	Stab 1	Rad 1	Report 1	Fail Safe 1	Send To AP 1	Check AP 1	PPM 1	Co - Check
2	FM Sim 2	FM GPS 2	FM IR 2	Nav 2	Alt 2	CC 2	Stab 2	Rad 2
3	Report 2	Fail Safe 2	Send to AP 2	Check AP 2	PPM 2	FM Sim 3	FM GPS 3	FM IR 3
4	Nav 3	Alt 3	CC 3	Stab 3	Rad 3	Report 3	Fail Safe 3	Send to AP 3
5	Check AP 3	PPM 3						
6								
7								

Figure 2.8: Paparazzi Task Layout

We conducted experiments with both the message-passing and shared-memory approaches using triple concurrent redundancy to evaluate the effectiveness of the Forte framework. We employed targeted fault injection in each of the models by generating data errors to evaluate the effectiveness of the coherency checks. We injected faults into both code segments and actively used data segments. Faults were dynamically inserted into the code and configured to be trigger randomly during the execution of the control system through the flight path. To model full redundancy, we duplicated the simulated UAV hardware so that each model operated on unique device inputs.

2.6 Experimental Results

Table 2.1 depicts the number of injected faults that are detectable (resulting in output faults) and the number of actually recognized faults. The results indicate that all detectable faults were recognized and subsequently averted using voting in the coherence checks. We implemented a single coherence check to validate system data prior to servo actuation. The coherence check assessed the output data that was passed over the peripheral bus to the servo controller. We only included outcomes from SEUs that created an actual effect on the output state of the running systems. Faults were categorized as follows: (1) Downstream data errors: prior to servo actuation, outputs of the models were compared for consistency. By using three duplicated models, the faulting model is defeated (voted out). (2) Read-only (RO) memory upsets caused one of the models to fault. When this occurred, one model failed the coherence check through a timeout mechanism set by the coherence check's data deadline.

Table 2.1: Fault Injection Evaluation

SEU Type	Detectable SEU Count	Recognized
Heap Flip	15	15
Device Failure	3	3
Stack Flip	10	10
Read Only Flip	4	4

The next experiment exemplifies one of the major benefits of the message passing design over shared memory. Figure 2.9 depicts the computational cost (in cycles) for accesses to data subject to coherency checks for both models. These results measure the coherence within the climb control model that maintains computational control over five of the system control variables. This coherency check validates the consistency of the three simultaneous climb control data sets. As Figure 2.9 indicates, shared memory results in an order of magnitude performance penalty compared to message-passing. The overhead of the latter is due to maintaining coherency for remote writes for the validation checks. The message-passing model eliminates the need for coherence and reduces conflicts on the interconnects resulting in more predictable and lower execution time.

Figure 2.10 depicts the overheads for computing integer data in the climb control task. These results show stable timings for task computation with message passing, much in contrast to shared memory. We evaluated integer computations because of a lack of hardware floating point units (FPU) on the Tilepro64. This data demonstrates how easily contention on the NoC results in jitter. In this result, three simultaneous models are executing while the previous results utilized only one active tile during the actual check. Note that when multiple tiles are active simultaneous jitter is easily introduced into shared memory accesses. In contrast, TDMA arbitrates NoC access for messages.

We implemented a naturally converging model and a data refresh model to assess the benefits of rejuvenation. To compare the models, it is necessary to measure the time from failure until

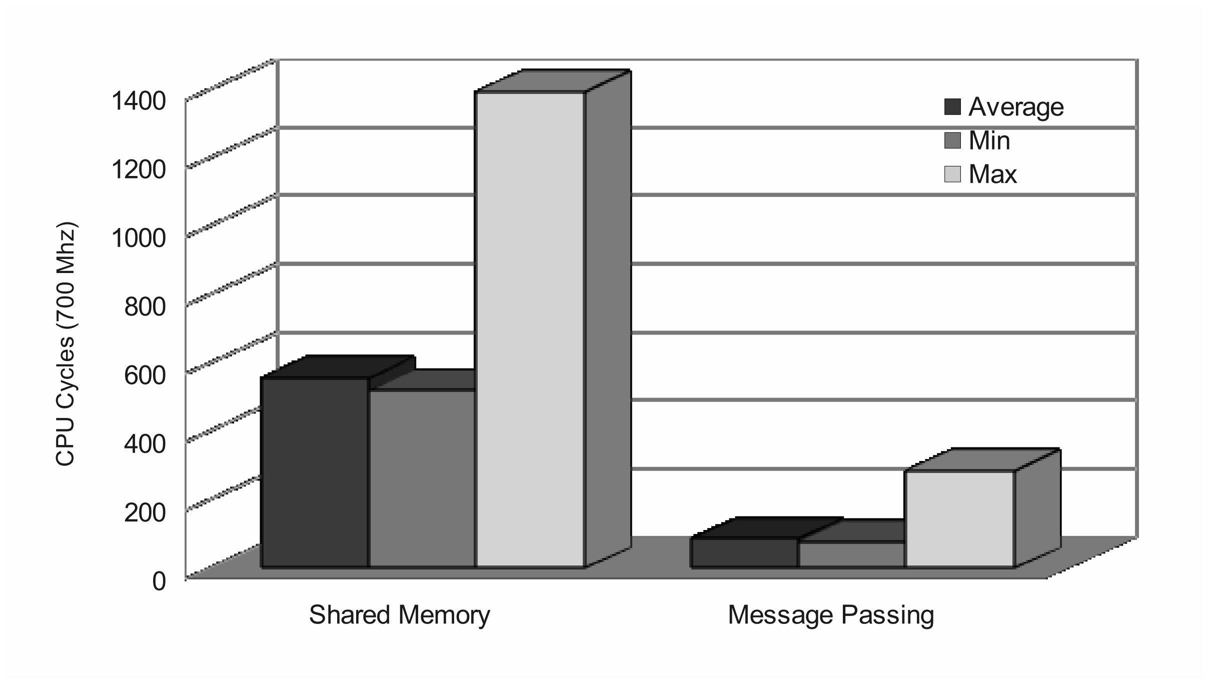


Figure 2.9: Overhead of Coherence: Shared Memory vs. Message Passing

Table 2.2: Rejuvenation: Time to Full Restart

SEU Scheme	Time To Repair	Mean Time to Failure
No Rejuvenation	∞	157 Days
Natural Convergence	8 (2s)	$2.27x10^8$ Days
Data Driven	1 (250ms)	$2.05x10^9$ Days

triple redundancy is restored, i.e., voting within the system can restart. Table 2.2 depicts this as the time to repair for each scheme. Column two indicates that natural convergence took eight job cycles (periods) before voting could restart while rejuvenation with data refresh was able to accurately measure coherence one job (period) after the original failure. Column three assesses the mean time to failure (MTTF) for each scheme. Without rejuvenation, the model to derive data for the second row follows the standard MTTF calculation $MTTF_{TMR} = 5/(6\lambda)$. The model with repair via rejuvenation used to derive results for the third and fourth rows is based on a modified Markov formulation that calculates MTTF as $MTTF_{TMR-Repair} = 5/(6\lambda)+\mu/(6\lambda^2)$ [63]. μ is the maximum number of repairs that can be performed within an hour. We evaluated our model based on the $\lambda = 2.2083x10^{-4}$ derived from a radiation-hardened Tilera processor for these results [15]. This provides a worst-case λ as the processor is hardened and the error rates are evaluated in space making λ higher than values derived for single-node failures for terrestrial applications, e.g., λ values reported for HPC environments. As can be seen from the

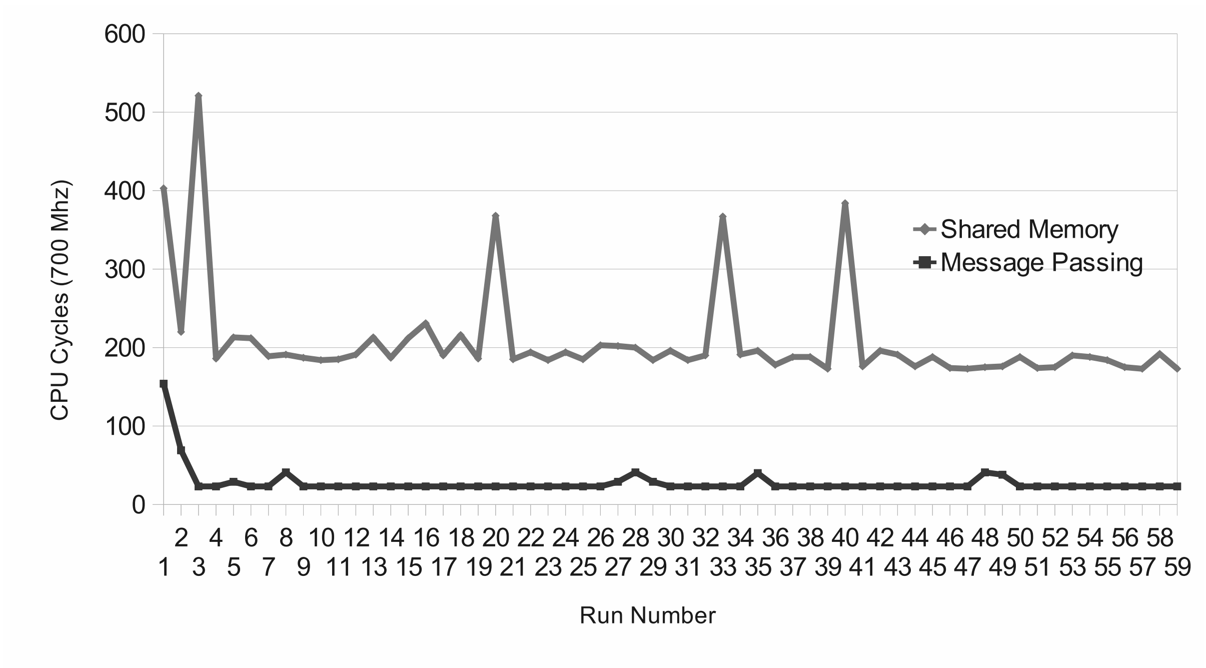


Figure 2.10: Climb Control Task Jitter: Shared Memory vs. Message Passing

results, rejuvenation increases reliable operation by six to seven orders of magnitude.

The experiments thus far assess the cost of communication in a cyber-physical system that only exercises some aspects of Forte's design. To evaluate the limits of Forte, we implemented a micro-benchmark that transfers a data payload of varied size between two cores. The benchmark utilizes both shared memory and message passing to evaluate the cost of aggregate data transfers. Memory addresses are uniformly distributed across the L3 cache by the hardware. Notice that this is a virtual L3 cache implemented through a hypervisor by distributing memory references over the L2 caches of all cores. The distribution uses a home-based protocol where the hash of a shared memory address redirects a look-up to a home core over a specific coherence interconnect on the NoC. Hashing can thus significantly increase the performance of shared memory by reducing the average distance to cached data and by increasing cache capacity of L3 to the aggregate of all L2 caches. This effect is demonstrated in Figure 2.11. Hashed shared memory significantly outperforms the non-hashed counterpart. However, even with a significant reduction in the cost of shared memory access, the message passing transfer outperforms shared memory in both configurations. Figure 2.12 depicts the cost (in cycles) of a zero-contention data transfer scenario over multiple runs illustrating the jitter for the respective models. These results prompted us to not use hashing in the previous experiments. The results indicate that hashing improves performance of shared memory but at the cost of 1.5% additional jitter since accesses to distributed L3 have variable hop counts over the NoC. A jitter of 1.8% is even observed in the message passing results when L3/hashing is active as a result of the forced address resolution and non-local data placement associate with hashing. Message passing under deactivated virtual L3/hashing results in lower jitter(only .5%).

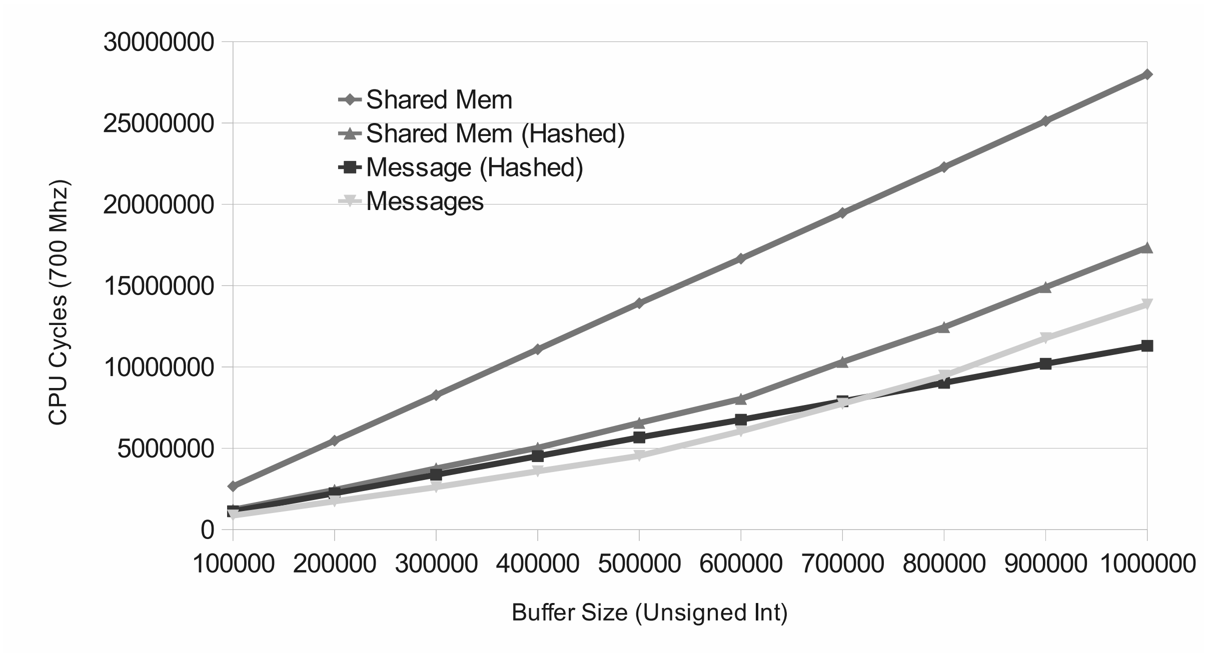


Figure 2.11: Bulk Transfer Overhead: Shared Memory vs. Message Passing

Next, we evaluated the scalability of the Forte design. We ran a single Paparazzi model of the full system in this experiment. The number of replicas of the altitude control task was scaled up gradually from 10 over 20 to 30 redundant instances. All replicas were executed in parallel on separate cores. This raised the overall utilization to 45 cores for the Paparazzi task set including the scheduler and coherence check. Figure 2.13 depicts the cost of data transfer/computation (in cycles) over multiple benchmark run for 10, 20 and 30 replica. A relatively inconsistent access cost is incurred with 30 replica cores for shared memory. Interestingly, a consistent additional overhead of approximately 50 cycles is observed for shared memory using 20 and 30 replica cores relative to just 10 cores, which can be accounted to scalability limits of the coherence protocol due to contention on the coherence interconnect. In contrast, additional replicas have virtually no measurable effect on the overheads for message passing (without L3/hashing) as TDMA arbitrates NoC access when messages are transferred. The occasional spikes in these results are caused by the virtualization layer in our experimental platform, which periodically activates a required monitoring daemon resulting in system noise. Such daemons would need to be eliminated or modeled as a separate task to meet real-time requirements.

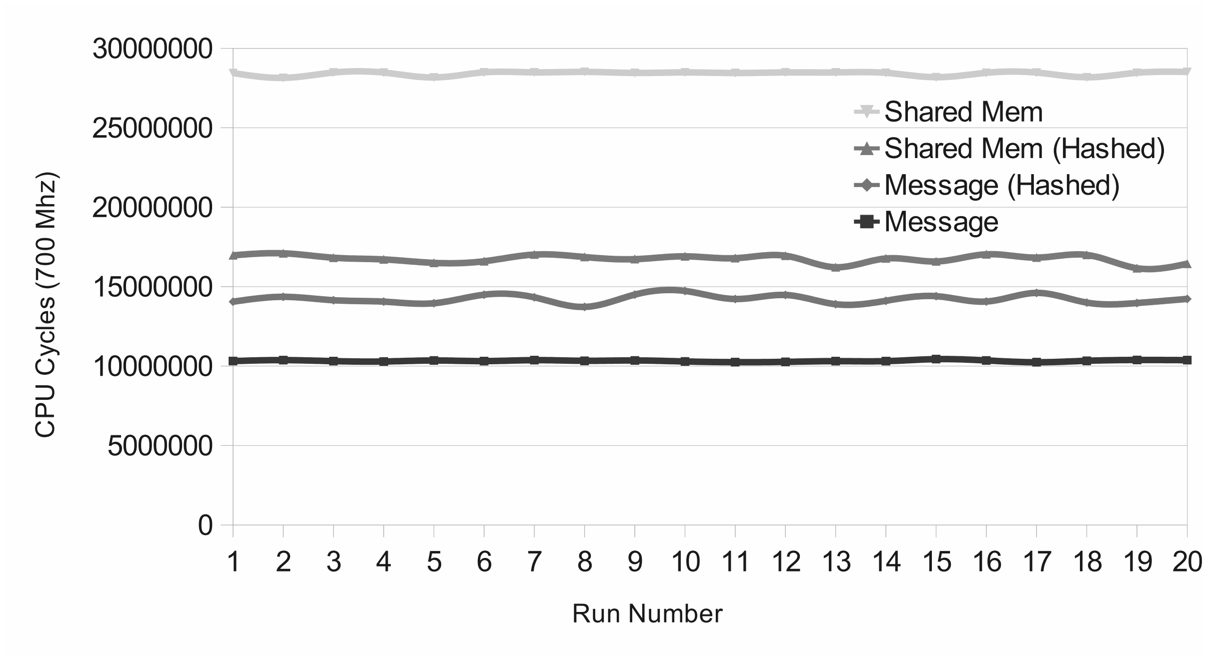


Figure 2.12: Bulk Transfer Jitter: Shared Memory vs. Message Passing

Overall, the results indicate superior performance, increased predictability and reduced jitter of pure message passing (without any background coherence protocol) in this massive multi-core platform with a mesh-based NoC. Performance and predictability benefits of message passing over shared memory improve as the number of utilized cores increases, *i.e,* message passing scales in contrast to shared memory programming. The cause of these benefits lie in the potential of one-sided communication and TDMA arbitration of message passing in a push-based (explicit) access model. These advantages cannot be matched shared memory protocols with its pull-based (implicit) on-demand access requests and its required hand-shake semantics of the coherence protocol.

2.7 Related Work

Our study appears to be the first one combining a *real* implementation on a *massive* multi-core with 64 cores with real-time constraints and fault tolerance. Prior work was strictly simulation based, let it be for studying topologies for *simulated* shared memory architectures [46, 74] or fault tolerance with hardware support in a simulated NoC environment [50]. One notable exception is the Multikernel (aka. Barrelfish) system that indicated that message passing can be superior to shared memory in an SMPs system using Hypertransport [10]. In contrast, our work is not on SMPs and focuses on much larger core counts that introduce scalability problems

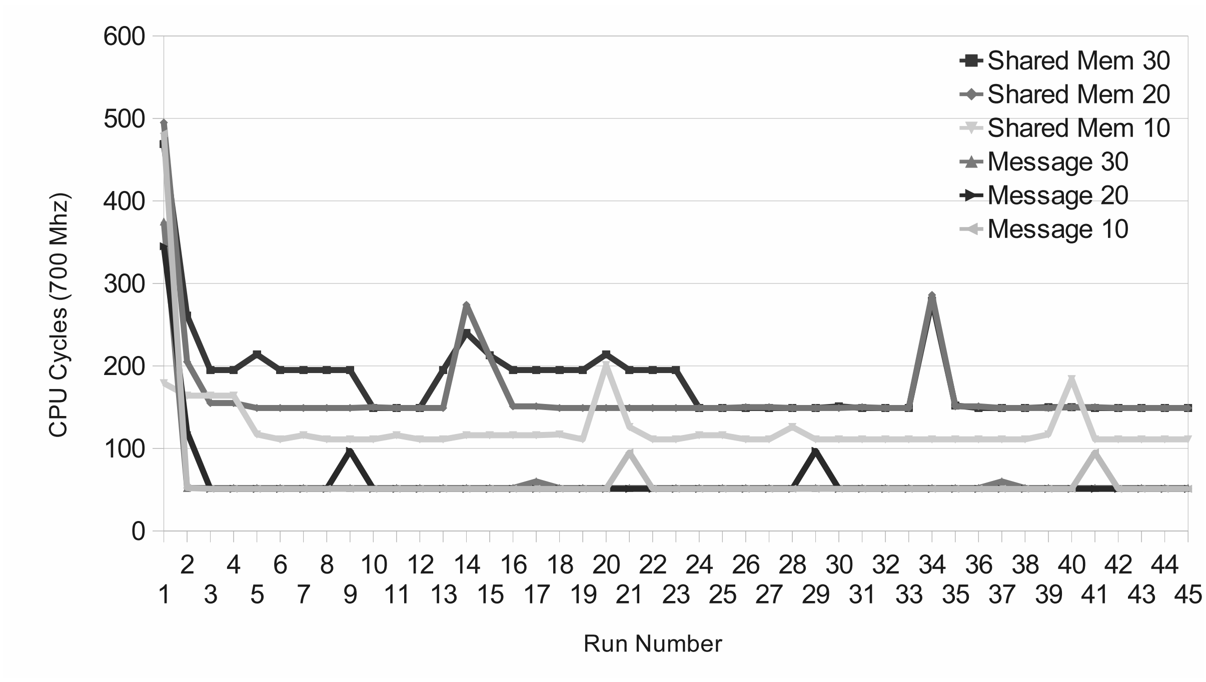


Figure 2.13: Scaling Contention: Shared Memory vs. Message Passing

due to NoC resource contention on a single processor die. Another exception is Tilera's iMesh paper [74], which investigates higher-level software overheads of the iLib abstraction for buffered channel and high-level dynamic messaging vs. row channels. Our work provides more insight on jitter and clarifies the overhead of shared memory accesses vs. the benefits of a much lower level messaging layer, which exposes the true overheads at the lowest software layers.

There is significant related work in the area of fault tolerance. Past approaches utilize scheduling, replication, or radiation hardening to achieve fault tolerance. Scheduling techniques, such as in [29, 24, 17], often introduce sophisticated scheduling policies to track faults. In particular, [29] introduces a last chance scheduling technique with the notion of task alternates to correct data in times of faults. A complicated scheduling algorithm then delays the execution of these alternates until the last possible moment to provide a fault tolerant schedule. We use advanced multi-core architectures to remove the need for such sophisticated scheduling by enabling the software to run alternates simultaneously at virtually no additional resource cost.

There exists a significant amount of work on detection of and protection against transient faults. Hardware can protect and even correct transient faults at the cost of redundant circuits [8, 80, 79, 65] Software approaches can also protect/correct these faults, *e.g.*, by instruction duplication or algorithmic design [62, 55, 52, 69, 51] Recent work focuses on a hybrid solution of both hardware and software support to counter transient faults [56, 57, 78]. Such hybrid solutions aim at a reduced cost of protection, *i.e.*, cost in terms of extra die size, performance penalty and increased code size. Hybrid approaches have been proposed for selectively protecting hardware regions, for control-flow checking and for reduced instruction and data duplication in software [56]. Data representations, however, have been widely ignored. Radiation hardening is another common technique in fault protection for real-time systems [32, 68] with overheads in costs and speed. In contrast to our work, these solutions either promote hardware approaches or do not consider massive multi-cores (or even real-time systems).

Modular redundancy is a replication technique[42]. This work provides an easy to implement and validate approach to ensuring fault tolerance. The technique has been used widely in research. [50] describes a heterogeneous NoC architecture to implement triple modular redundancy. This work focuses on a specialized architecture that supports multiple levels of hardware integrated fault detection. This work uses TDMA on a NoC to interconnect the various IP elements in the architecture. Our work also utilizes a replicated task mapping but differs in that it is a pure software approach that enables comparisons of varying task complexity models with COTS applicability. More significantly, their study is based on hardware simulation, ours is an actual implementation on a hardware platform. Theirs covers a small number of cores, ours is for *massive* multi-cores with high core count, which creates novel challenges for harnessing NoC contention.

Rejuvenation [81, 31] is a technique originally introduced as a software restart technique to protect long-running software. Rejuvenation is often associated with rebooting. A major hurdle in software rejuvenation is data loss due to the rejuvenation. Forte uses software rejuvenation to maintain reliability in the control system. Data loss is circumvented through selective rejuvenation and data refreshing from validated data models.

2.8 Conclusion

In this chapter we have presented the design of Forte, a framework that utilizes massive multi-core NoC architectures in order to create a reduced jitter and fault tolerant cyber-physical environment. The primary tenets of this approach encompassed systematic restructuring of traditional real-time tasks to eliminate the use of shared memory by instead relying on message passing to move data between tasks. By reducing contention on memory controllers, it becomes more feasible to scale up the number of cores while sustaining performance and predictability. This enables support for fault tolerance through replicated real-time tasks combined with consistency verification and task rejuvenation using modular redundancy. Our results feature experiments with triple modular on-chip redundancy for a UAV control system and illustrate capabilities of Forte to detect errors and correct tainted results due to data errors, such as SEUs. We also show that by putting greater emphasis on message passing and eliminating shared memory accesses, we are able to increase predictability and decrease overheads by up to an order of magnitude. System reliability can be further increased by six to seven orders of magnitude when triple modular redundancy is combined with naturally converging and refresh-assisted rejuvenation, respectively.

Chapter 3

Low Contention Mapping of Real-Time Tasks onto a TilePro 64 Core Processor

3.1 Introduction

Distributed software models on network-on-chip (NoC) processor architectures provide significant advancements but also challenges for real-time systems. These advancements come from simplifications in processor cores that result in increased accuracy of static timing analysis, simplified scheduling algorithms due to an abundance of cores, and synchronization free data resource models implemented through explicit inter-process communication (IPC) in the form of messages. Due to these advancements, this processor architecture is seeing increased use in hard real-time systems such as in [70] where the authors explore real-time hazard detection in satellites using the Opera Maestro processor [15], a radiation hardened TilePro with 49 cores developed by Boeing. A drawback of these processors is posed by NoC contention of multiple tasks. Such contention exists for shared-memory accesses, for off-chip memory references and for message passing when utilizing distributed software models instead of shared memory. Our work focuses on message passing over the NoC assuming separate NoC interconnects for memory, coherence, I/O and messaging [6]. Other work on increasing predictability and coping with non-uniform memory latencies is orthogonal [46].

Message-based communication over the NoC has been shown to increase scalability compared to shared-memory programming [10]. We conjecture that it can also assist in increasing predictability by decreasing contention as it is easier to analyze messages statically than shared memory references [54]. Even under message passing, poor task-to-core mappings can result in a loss of predictability due to latencies incurred through NoC contention. Consider a mesh

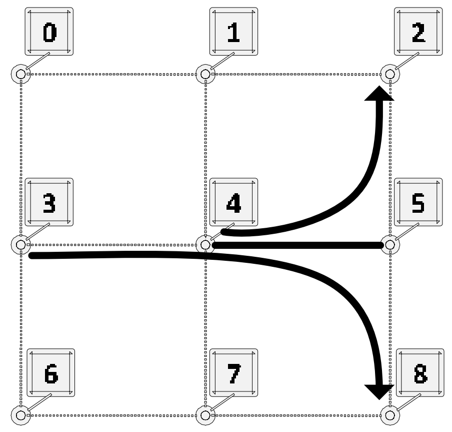


Figure 3.1: NoC Contention (Config 1)

NoC with full-duplex links, i.e., two messages traveling in opposite directions over a link do not result in contention, that utilizes static dimension-ordered wormhole routing favoring horizontal routing before vertical [6]. Consider the example "Config 1" in Figure 3.1 of nine cores with a mesh NoC. Two messages are sent, one from core $4 \rightarrow 2$ and the other from $3 \rightarrow 8$, as depicted by the lines with arrows. When sent at the same time, contention on the link $4 \rightarrow 5$ (depicted as a thick link in the NoC mesh) results in a delay for one of these messages due to arbitration within the NoC hardware routers. (Packets are not interleaved as an open virtual channel monopolize links between endpoints.) As a result, sending tasks experience highly variable latencies. Such variability can be reduced or even eliminated when tasks are layed out intelligently to lower or even completely avoid contention, respectively. The effect shown in this example is amplified as the size of NoC meshes increases resulting in larger paths through networks and communication that is more frequent.

We propose an abstraction model for message-based NoC contention that, when applied to statically scheduled hard real-time tasks, allocates messages into temporal windows, so called "frames". These frames provide the foundation for static analysis on communication paths to evaluate task-to-core mappings. We formulate a constraint problem and implement an "exhaustive solver" that provides optimal mappings. Unfortunately, exhaustive approaches do not scale beyond small NoC mesh sizes as they can take days to solve mapping layouts. Hence, we further develop a multi-heuristic solver, called "HSolver". We identify a set of effective mapping patterns that yield near-optimal results while providing sustained scalability in finding such solutions even for large NoC meshes. Finally, we contribute a micro-benchmark that empirically tests our designed model and evaluate our approach on a Tilera TilePro processor with 64 cores [6]. Using our solvers, we are able to reduce contention by over 70% compared to a naïve and the constrained exhaustive solutions for NoC sizes too large to be solved optimally within hours.

To the best of our knowledge, this is the first work to address predictability of NoC communication via framing messages into temporal windows for real-time tasks. Previous work [16] viewed communication as temporally stateless. This limited the amount of communication that could feasibly be solved. It also resulted in solutions that were overly conservative in that *any potential* for common message routes were considered contention. By using temporal windows, we are able to solve the problem by *separating temporally disjoint messages* when analyzing link contention scenarios and thus increasing communication predictability.

3.2 Software-Based Temporal Framing

Temporal framing is a technique similar to time-division multiple-access (TDMA) that imposes frames to bound communication into time windows. It differs from TDMA in that it does not limit the amount of IPC in a single frame; instead, it is used solely for analysis purposes to try and statically map tasks to processors and to reduce communication interference. Programmatically, this is facilitated using self-referential frame checking within a task for identifying when a specific message can be sent/received. To guarantee predictability within a real-time environment, we assume that senders and receivers are at least predetermined within the hyper-period of a periodic task set and that communicating pairs are guaranteed to be active during any frame in which they send or receive data. This assumption is easily supported under the dark silicon model because high utilization and delays due to resource sharing do not exist. Because of this assumption, we can then use temporal frames as a means of synchronizing senders and receivers to reduce latency incurred through non-synchronized IPC.

As an example consider the nine real-time tasks shown in Figure 3.2 where tasks are represented by shaded blocks. In this figure, the execution of the system is broken into twelve temporal frames. Communication in a frame is represented by indicating within a sender's frame the receiver address. Communication too large to fit within a single frame is considered across multiple frames. For example, Task 3 send a message to Task 8 during frames nine and ten (see Figure 3.2). Using this model, we formulate a mathematical model to map tasks to cores while maintaining temporarily disjoint communication.

In this model, we abstract out directional links from a core and its switch (input queues and an output port, see Figure 3.1) without affecting correctness: Output contention cannot occur between core and switch as only a single task, i.e., only a single sender exists per core. Input contention could result when multiple tasks send to a single receiver along disjoint paths to a destination switch but only one message can be delivered at a time via the switch-to-core link. Such contention is permissible under our model, and we show how blocking under input contention can be bounded by the number of senders. As an example, if task 6 sends a message

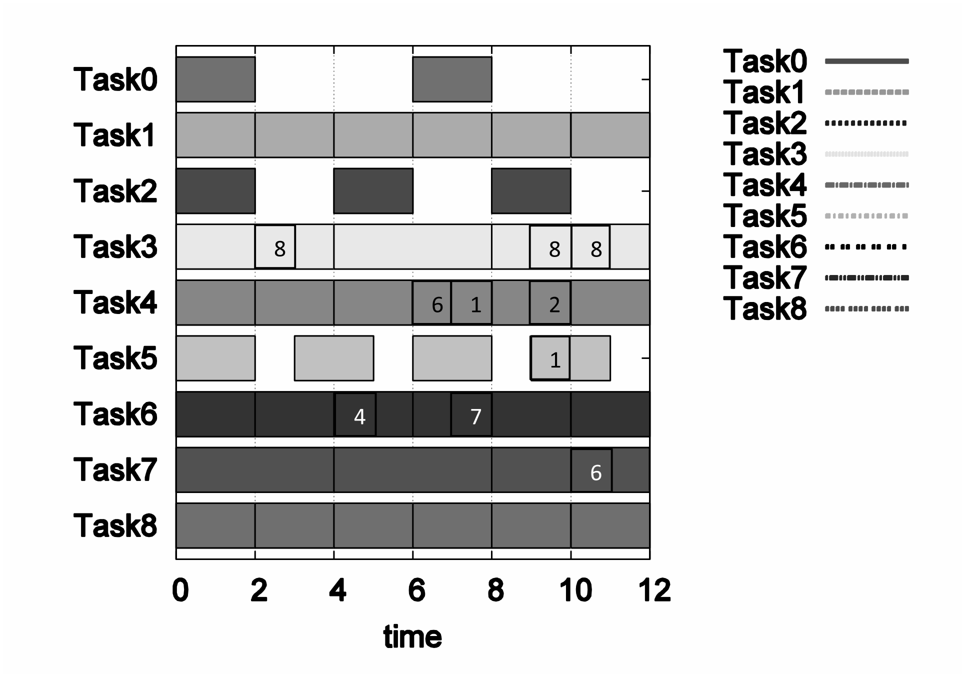


Figure 3.2: Temporal Framing Example

to task 8 in frame 2 of Figure 3.2, sends from tasks 3 and 6 would result in input contention at task 8, the receiver of both. Whichever send, that of task 3 or 6, comes later, it would block until the earlier message has been received due to input contention. In the following, we will explicitly refer to "input contention" and otherwise use the term contention to refer to "link contention" within the mesh.

3.3 Motivation

Let us provide a motivational example to assess the impact of contention-based latency on real-time tasks. We use a 3x3 NoC with nine tasks broken into 12 temporal frames as described in the previous section, our "running example" used throughout the paper. The randomly generated task set has high utilization that takes advantage of the NoC architecture using message-based IPC. There are 10 messages that are sent within the hyper-period. These messages are shown in the temporal framing example in Figure 3.2. We evaluated three layouts of the tasks on the NoC, each with different amounts of network contention, to show the impact of contention on jitter.

We first evaluate a contention scenario based on a naïve layout, "Config 1", as shown in Figure 3.1. In this layout, the tasks are mapped to the core corresponding to the task id.

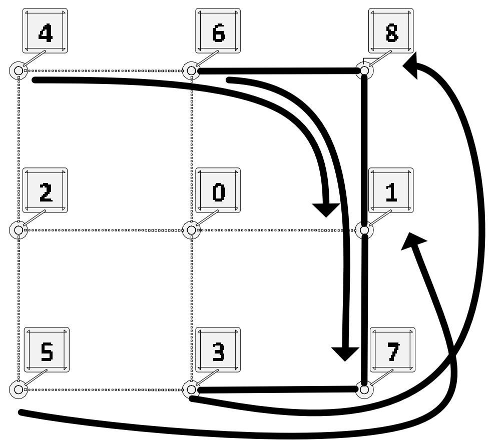


Figure 3.3: Contended Network Resource (Config 2)

The naïve layout results in contention along edge 4 → 5. This contention is a result of two simultaneous messages, as previously described. There is actually an additional message during this frame 9 from Tasks 5 → 1 (see Figure 3.2) that does not result in contention due to link duplexing.

The second contention scenario, "Config 2" in Figure 3.3, contains a shows the effect of contention across multiple temporal frames and its effect on jitter. Temporal frames 7 and 9 result in contended links between two sending/receiving pairs in each frame.

We also evaluate a third layout without contention, "Optimal" in Figure 3.4. No routes are shown in this figure due to the absence of contention since links are full duplex, i.e., edge traversals initiated on opposite ends of a link do not result in contention.

The graph in Figure 3.5 shows a comparison of the jitter across the worst case transfer times (WCTT), i.e., the maximum measured transfer latency for each transfer in the presence of contention, measured over the course of 100 task executions for each of the layouts. The x axis indicates message size, and the y axis shows the amount of jitter in CPU cycles on a 700 MHz TilePro64. As message sizes increase, jitter and WCTT along edges with link contention also increase for both Config 1 and Config 2. Additional contention across temporal frames leads to greater jitter within the system as seen by high jitter for Config 2 than Config 1. Results for Optimal, a zero-cost layout, show constant and low jitter as data transfer sizes increase. It is important to note that small jitter is incurred even in optimal layouts as NoC grid sizes increase due to added latency from additional hops to traverse the network. These results emphasize the necessity to consider and minimize task layout and network contention on NoC architectures for real-time systems.

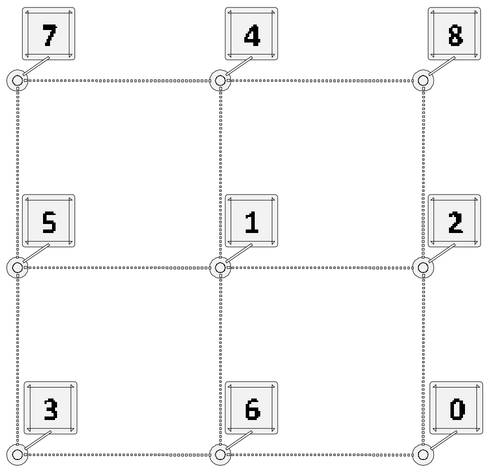


Figure 3.4: Zero Cost Network Layout (Optimal)

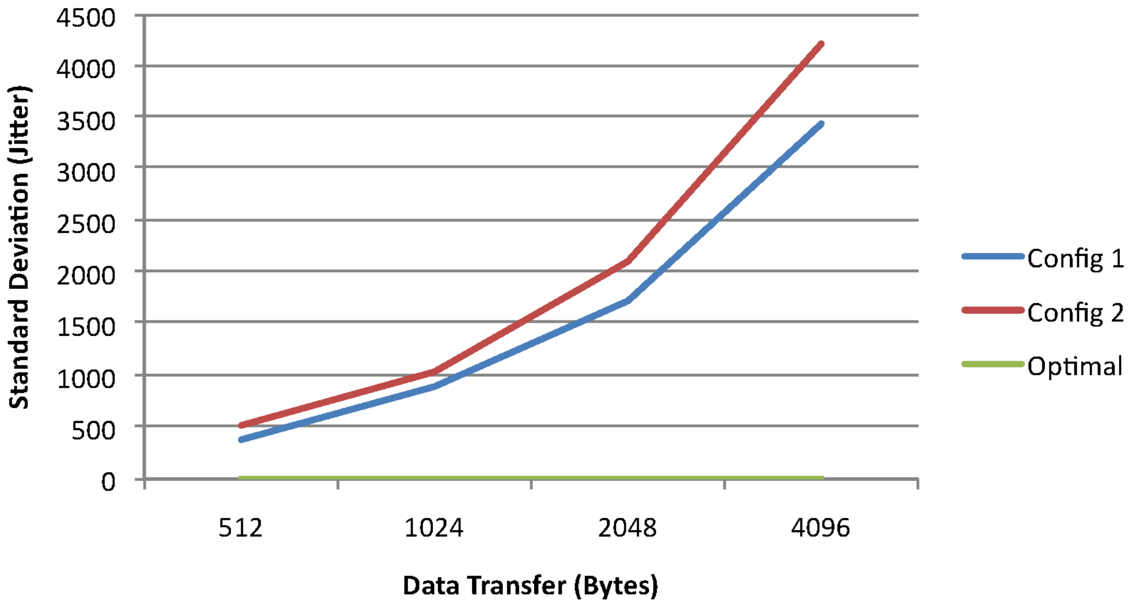


Figure 3.5: Contention Related Jitter

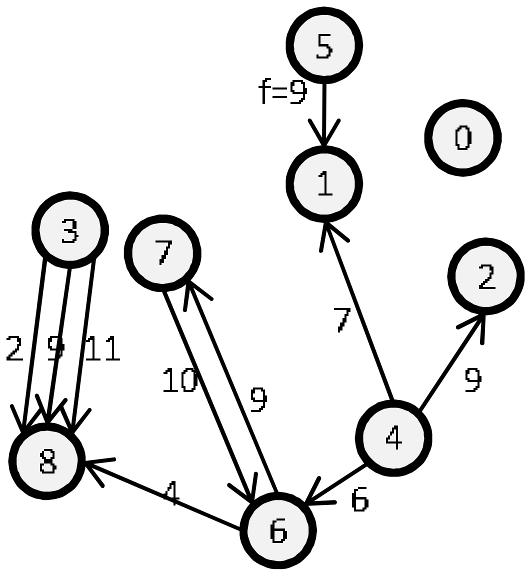


Figure 3.6: Temporal Framing Graph

3.4 Exhaustive Solver Model

We utilize the temporal framing model described previously as a basis for a constraint programming formulation. This reduces contention, ease analysis, and maintain communication flexibility. The constraint framework allows us to systematically determine optimal task-to-core mappings for NoC architectures. In the formalization, the set of tasks is considered as a temporal frame graph shown in Figure 3.6. The figure depicts the graph representation of our running example representing any inter-task communication, and edge weights indicate the time frame during which the communication occurs. We construct temporal frame graphs based on the specification of communication within the real-time task sets (as in Figure 3.2) and map them onto a core graph, a representation of the NoC topology. We then formulate a constraint-programming model that is solved in a branch-and-bound traversal to enumerate and evaluate all possible mappings so that the amount of IPC based contention is minimized. The following definitions specify the constraint model for determining an optimal solution (there may be more than one) that minimizes contention.

Definition 1: A Temporal Frame Graph $TFG = (T, C)$ is a weighted and directed graph, where $t_i \in T$ represents tasks in a real-time system and a directed edge $c_{i,j,f} \in C$ represents communication between two vertices (t_i, t_j) where t_i is the sender and t_j is the receiver within the temporal frame f indicated by c_f.

Definition 2: A Directed Mesh Graph $G = (V, E)$ is a representation of cores over a NoC where $v_i \in V$ represents cores on a NoC and $e_i \in E$ is an edge between two cores v_i, v_j identified in directional order by (v_i, v_j).

Definition 3: A function $Map(t)$ maps vertex $t \in T$ onto a vertex $v \in V$.

Definition 4: An ordered set $Path(v_i, v_j)$ denotes the XY dimension ordered edges on the Manhattan path [40] (edge traversal) between $v_i, v_j \in G$.

Definition 5: A function $Cross(v_i, v_j, v_m, v_n)$ is defined as

$$Cross(v_i, v_j, v_m, v_n) = \begin{cases} |Path(v_i, v_j) & if\ v_i \neq v_j \wedge \\ \cap Path(v_m, v_n)| & v_m \neq v_n \wedge \\ & |Path(v_i, v_j) \cap \\ & Path(v_m, v_n)| > 0 \\ 0 & Otherwise \end{cases}$$

.

Definition 6: Function $Cost(c^1, c^2)$ with $c^1, c^2 \in C$ is defined as

$$Cost(c^1, c^2) = \begin{cases} Cross(Map(t_i^1), Map(t_j^1), & \\ Map(t_i^2), Map(t_j^2)) & if\ c_f^1 \neq c_f^2 \\ 0 & Otherwise \end{cases}$$

.

Definition 7: The objective function (to be minimized) is $Min(TFG) = \sum_{c^1 \in C, c^2 \in C} Cost(c^1, c^2)$.

Definition 8: A set of constraints on the minimization function are defined as $\forall t_1 \in T, \forall t_2 \in T : t_1 \neq t_2 \implies Map(t_1) \neq Map(t_2)$.

The constraint framework defined above specifies an optimization problem whose cost is to be minimized, i.e., the cost associated with mapping the TFG onto a Mesh Graph G. To understand the cost function, let us revisit the definitions. Between any two vertices in the core graph there is an ordered set $Path(v_i, v_j)$ that represents the set of edges traversed over an XY dimension-ordered route between v_i and v_j (see Def. 4). The set contains edge tuples in which $< v_x, v_y >$ defines a single edge between v_x and v_y. This tuple is strictly (directionally) ordered such that $< v_x, v_y >$ and $< v_y, v_x >$ refer to separate edges to support the notion of full duplex edges that exist in many NoC architectures. To determine the conflicts that occur between two paths, we define the function $Cross(v_i, v_j, v_m, v_n)$ that specifies the cardinality of the intersection of the two paths defined by $(v_i v_j)$ and $(v_m,\, v_n)$.

The *Cross* function is used to define a scalar function $Cost(c^1, c^2)$ parameterized by two edges obtained the TFG graph (see Def. 6). It then applies the *map* function on source and destinations in c^1, c^2 and determines the number of contended links that exist between the two paths. This determines the total number of contended links that exist on the paths defined by c^1 and c^2 assuming that c^1 and c^2 occur during the same temporal frame. Otherwise, the result is zero since communication does not exist during the same temporal frame and thus no messages can interfere with one another. The optimization function of this constraint framework is minimizing the sum of the cost functions across all edges in the TFG. The constraints to bound this function enforce a unique mapping, *i*.e. each task is mapped to one core and no two tasks share a core (see Def. 8).

3.5 Heuristic Model

Branch and bound, exhaustive optimization solvers scale exponentially as the number of variables grow. This holds true in our exhaustive contention solver detailed in the previous section. Solutions for NoCs 5x5 and larger can take hours to obtain an optimal solution. This is particularly true when the full depth of the search tree has to be traversed, even when optimized in C and parallelized over multiple nodes (using MPI) as in our implementation. We developed HSolver, a heuristic solver to create a low contention layout for NoCs too large to be solved exhaustively. HSolver is a multi-heuristic solver designed based on patterns identified from optimal solutions generated from the exhaustive solver.

HSolver composes multiple heuristics and generates fast and low contention mappings of tasks to cores based on communication traces. It determines the lowest cost solution over a set the heuristics during the mapping process. HSolver applies two classes of heuristics at each stage of the mapping process: (1) task selection heuristics and (2) core selection heuristics. The base algorithm operates by choosing a task selection heuristic and then mapping it to each unmapped core identified by each of the core selection heuristics, ultimately mapping it to the core that results in the lowest contention cost (local minimization). The mapping process for each task terminates after evaluating every core or when a single mapping results in an unchanged system cost. In this section we use the term degree to denote the edge degree of the corresponding temporal frame graph.

3.5.1 Task Mapping Heuristics

HSolver applies three different heuristic selection techniques with each of the core heuristic strategies to determine the lowest cost solution.

(1) **Maximum Degree First**: This selection strategy is based on the premise that high degree solutions will result in high contention on a given NoC with few scheduling choices to avoid contention. Here, we seek to schedule the highest degree tasks first so that we have a higher degree of flexibility when scheduling subsequent tasks adjacent in the TFG. This heuristic performs best in situations where only a few tasks communicate frequently. In such a situation, the few communicating tasks are scheduled to cores early offering the most scheduling alternatives to reduce contention cost. We expect that this heuristic will be used frequently.

(2) **Minimum Degree First**: The lowest degree selection strategy chooses tasks starting with the least frequently communicating one first. Using either sending or receiving activity increases the communication degree. This strategy is the inverse of the previous strategy. We do not expect frequent use but include it for symmetry.

(3) **Maximum Cross Chat First**: This strategy operates on the principle of scheduling tasks together that frequently communicate. The selection heuristic starts by scheduling the task with the highest degree onto the empty core map. We then select subsequent tasks that most frequently communicate with the currently scheduled group if the number of message exchanges with this group is greater than a pre-determined minimum threshold. When no tasks to be scheduled remain within that group, determined through graph connectivity within the TFG, we schedule the remaining highest degree task and begin group scheduling again. This heuristic is based on a common pattern seen within the optimal solutions of frequently communicating partitions. We expect this to be a frequently used heuristic for high-frequency communicating task sets.

(4) **Minimum Cross Chat First**: This strategy operates on a trivial change to the Maximum Cross Chat First strategy: It schedules the lowest degree nodes first. The objective here is to schedule the smallest partitions first. We do not expect to see this pattern but included it for symmetry.

3.5.2 Core Mapping Heuristics:

The following are strategies used to select the order of the cores to evaluate an also selected single task at a time.

(1) **Maximum Degree First**: NoC tiles on 2D meshes have a varying number of communication edges dependent on their location. Innermost tiles contain four communication edges, non-edge corner tiles contain three communication edges, and corner tiles contain two communication edges. This technique attempts to schedule tasks to high degree cores first to

try to reduce opportunities for contention later. This example is based on optimal solutions that map frequently communicating tasks to highest degree cores. This gives high degree tasks more flexibility in establishing communication channels over the NoC without contention.

(2) **Maximum Cross Chat First**: Similar to the task mapping strategy, this core mapping strategy attempts to place tasks with high degrees of common communication physically close. When a task is selected, the core mapping heuristic analyzes the current mapping layout to determine the task on the map with the largest common communication degree. If the highest degree of cross chat is greater than a predetermined threshold, it will then determine an empty core within the fewest number of hops and attempt to schedule the task in that location. If the task selection heuristic selects a task below the threshold, the core selection heuristic places the task as far away as possible from the previously mapped groups.

(3) **Spiral Out-To-In**: This mapping strategy orders core selection onto a spiral traversal of the NoC starting at the first core in the Cartesian space and the assigning cores as a spiral around a matrix. This solution is most effective when scheduling lowest degree tasks first. This places low degree tasks along the low edge count tiles on the outside edges of the NoC, with the highest degree tasks toward the center of the NoC. We expect Maximum Degree First selection to work equally well in most scenarios and thus do not expect this core selection heuristic to be used frequently.

(4) **Spiral In-To-Out**: This mapping strategy reverses the allocation order of the Out-To-In spiral and allocates starting with the internal high degree nodes. This allocation strategy while originally included for symmetry is expected to work quite well when paired with task selection strategies that schedule high degree tasks early, giving more flexibility in location to high degree tasks. Later contention then would occur mainly from placement of lower edge tasks and may result in more frequent use of this selection technique.

3.6 Micro-Benchmark

To evaluate the effect of task mappings onto NoCs of a a real processor, we have designed and implemented a micro-benchmark that emulates the layouts and message traces on actual hardware. We found it necessary to implement our own benchmark due to a lack of NoC-level message passing benchmarks for massive multi-cores (e.g., Parsec [13] and Splash-2 [77] support shared memory only while NAS [9] relies on heavy MPI semantics with collectives). Our micro-benchmark provides cycle-accurate measurements on this hardware platform and allows the distinction between hardware or the software overheads. Our benchmark implements the temporal frames abstraction described previously.

The micro-benchmark requires as input the real-time schedule, message traces, and a mapping layout. The complete framework and inputs are shown in Figure 3.7. These inputs are

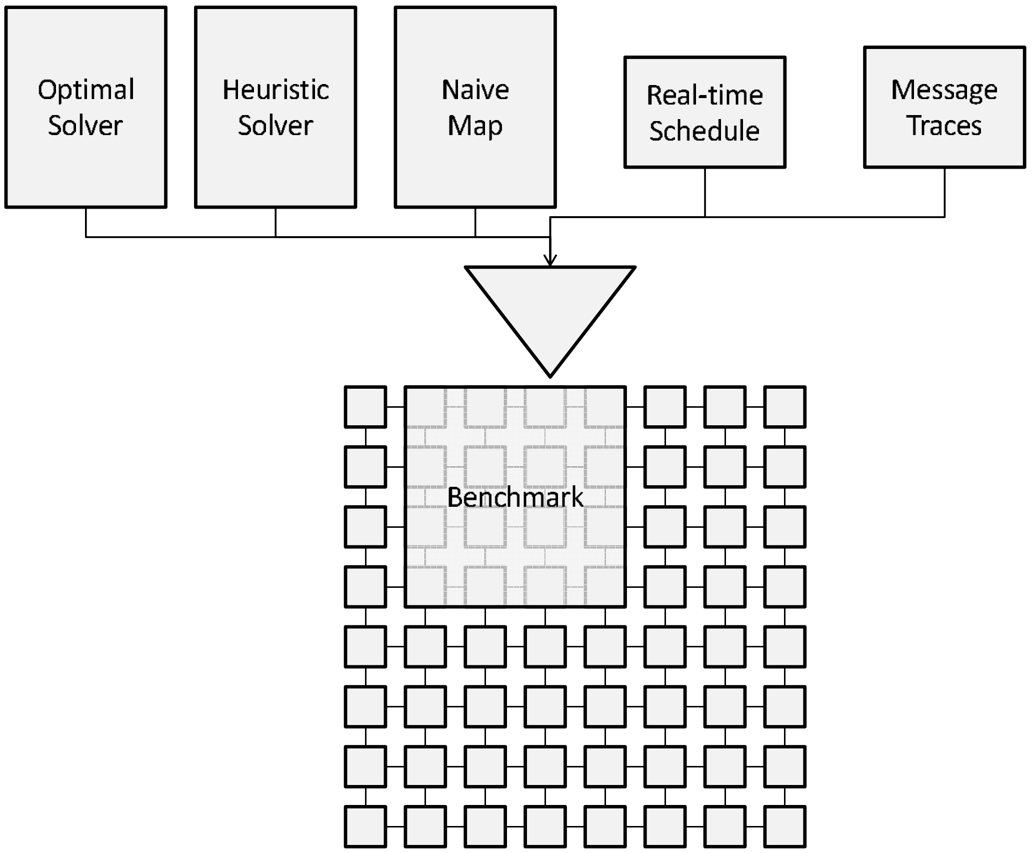


Figure 3.7: Tilera Evaluation Framework

divided into various phases based on the message traces and real-time schedule. These phases are defined as computation, sending, receiving, and idle phases. The benchmark uses the real-time task configuration to determine the emulated NoC size and lays out the task set on a contiguous grid of the configured NoC according to the layout designated.

The benchmark is deployed on a Tilera TilePro64 processor running at 700 MHz that contains 64 cores in an 8x8 NoC mesh and six independent mesh networks for memory, coherence, I/O, etc. Among these meshes is a low latency message-passing network called the user-dynamic network (UDN) that can be used to transfer messages of sizes up to 1KB at a time. The network is 32 bits wide and supports bi-directional communication. Messages are transferred via wormhole routing that locks the XY ordered Manhattan path between two cores during the transfer of a message. Further exploration of the Tilera message passing network can be found elsewhere [74]. The benchmark is designed to take advantage of the dataplane options supported by the Tilera architecture that include a low overhead operating system variant of Linux called Zero-Overhead-Linux. Using this platform, our benchmark is able to allocate a grid of cores on the TilePro64 to emulate smaller grids that do not service operating system interrupts. This results in very predictable execution. Our micro-benchmark allows us to execute randomly generated task-sets derived from their message traces. We measure the impact of layout, network contention, and temporal abstraction on predictability.

3.7 Results

We implemented two solvers, an exhaustive solver and HSolver, to find effective mappings that minimize/reduce contention costs. We then compared their outcomes across 100 randomly generated task sets of multiple sizes. Each randomly generated benchmark consisted of a number of tasks equal to the number of cores in the experiment. The period for each task was randomly generated but limited by varying the maximum resultant hyper-period. Varying the maximum hyper-period and the number of messages within each task set allowed us to control the communication density to improve the likelihood of link contention. The exhaustive solver was implemented using C++ with MPI and evaluated using 64 cores over 4 nodes with two sockets of AMD Opteron 6128 processors (8 cores per socket). HSolver was implemented in C++ using only a single processing thread within the same hardware configuration. To evaluate this work systematically, we randomly generated multiple real-time task sets and message traces for all evaluated NoC dimensions. Thresholds used in HSolver were dynamically evaluated from two to 1/2 the number of cores in the NoC and compared to determine which threshold value resulted in the lowest cost solution.

In our first experiment, we compare the minimum solutions for each of the solvers as the complexity of the systems increase. We refer to two different complexity metrics: (a) the size of the grid and (b) the number of messages sent during a task set's hyper-period. Increasing the number of messages increases the probability that during any single time frame multiple transfer pairs contend for the same link, thus decreasing the chances for a zero cost solution. Also, increasing the size of the grid exponentially increases the time to convergence. This requires time limits on the exhaustive solver to avoid indefinitely waiting for solutions. We refer to this in the remainder of the paper as the *constrained exhaustive model*, which only provides an optimal solution if it terminates within the given time bound — otherwise, the minimum within the traversed subspace is returned. We constrain the exhaustive approach to runtimes ranging from three minutes at 4x4 up to one hour for 8x8 NoC sizes.

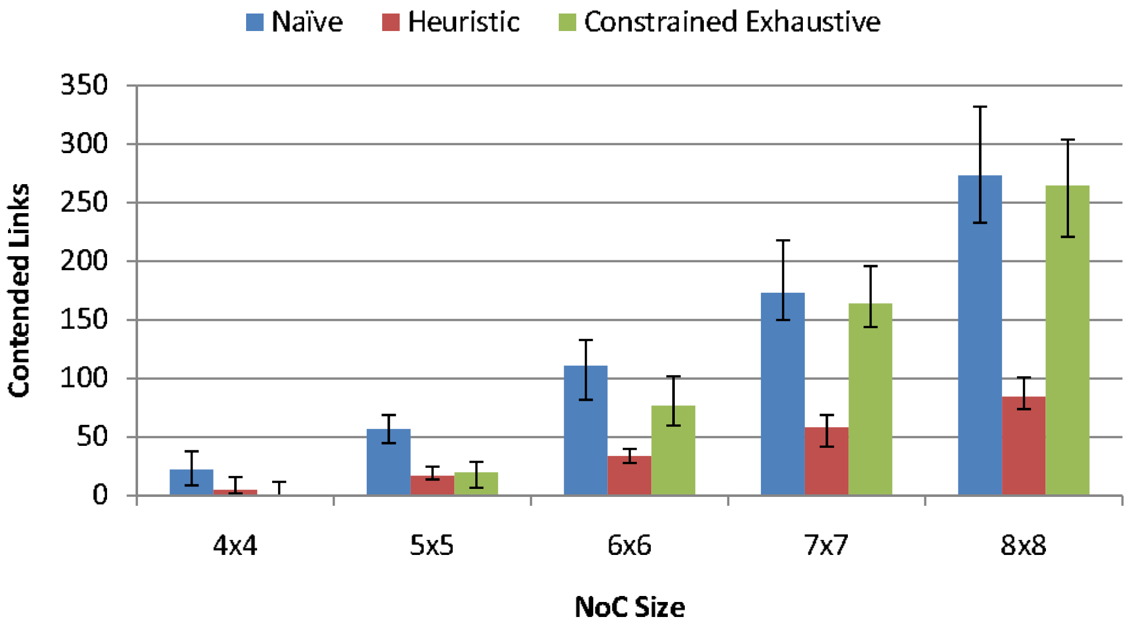


Figure 3.8: Average Contention for Mapping Strategies

We evaluated the minimum aggregate cost across 100 randomly generated task sets in naïve , heuristic, and constrained exhaustive mappings as the NoC size increases along with a linear increase in the number of messages. The results in Figure 3.8 show the aggregate contention for each NoC size. For each NoC size plotted over the x axis, three bars report the CPU cycle delays due to links contention (y axis) for the naïve , the HSolver and constrained exhaustive solver, respectively. The x-axis scales quadratically with the size of the NoC. Table 3.1 shows the averaged amount of processor time taken by the two solver types for each of the NoC sizes. In the constrained exhaustive approach, after the time cut-off, we still allowed execution

Table 3.1: Average Solving Times [hh:mm:ss.ms]

NoC	Heuristic	Constrained Exhaustive
4x4	00:00:00.60	00:08:00.00
5x5	00:00:03.00	02:57:00.00
6x6	00:00:14.00	06:46:00.00
7x7	00:00:50.00	06:58:00.00
8x8	00:03:00.00	11:06:00.00

to occur but no new branches to be formed. This is why the times vary for the constrained exhaustive solutions as the remaining subspace will still be considered after disabling further branch-outs.

The results show that as the grid and message complexities increase, the solving time for the constrained exhaustive case increases exponentially while HSolver's time increases linearly with mesh sizes. Furthermore, the aggregate contention cost for constrained exhaustive solutions exceeds that of HSolver as mesh sizes increase. The constrained exhaustive solver generates optimal results consistently for NoC sizes of 4x4. At 5x5, it was still able to generate optimal solutions for a large subset of the test cases — but not all of them. At 6x6 and beyond, the constrained exhaustive solver no longer provided optimal solutions or even mappings anywhere close to those of HSolver when its traversal was cut short by timing constraints due to the exponential explosion of the search space. Error bars show the disparity between the minimum and maximum results within each local benchmark grouping. The error bars indicates that HSolver results in solutions that are more consistent across each data set. We also observe a near constant growth rate in overhead for the heuristic approach but a faster (at last second degree polynomial if not higher) growth rate for both the naïve and the constrained exhaustive solvers as solution sizes increase. This is important for assessing scalability for large NoC processors and with high IPC utilization. The growth rate shows that as these metrics increase, naïve and constrained exhaustive approaches will result in real-time systems with quadratically increasing levels of contention. Hence, heuristic schemes seem very promising as the first 100-core processors are said to be released in 2011 with the TileGX [5].

We also evaluated the HSolver approach to determine the rate at which heuristics were used to generate the low-cost solution. These results allow us to identify which are the important heuristics and to provide a direction for analysis in determining why certain heuristics are more effective. Results in Figures 3.9 and 3.10 are taken across all benchmarks generated. Figure 3.9 shows the core selection strategies and the percent of use of each during heuristic solving. These results show a significant variation in the effectiveness of core strategies. Overall, minimizing the distances between frequently communicating cores is the most beneficial heuristic. This correlates well with the results from Figure 3.10, where two selection strategies account for 98%

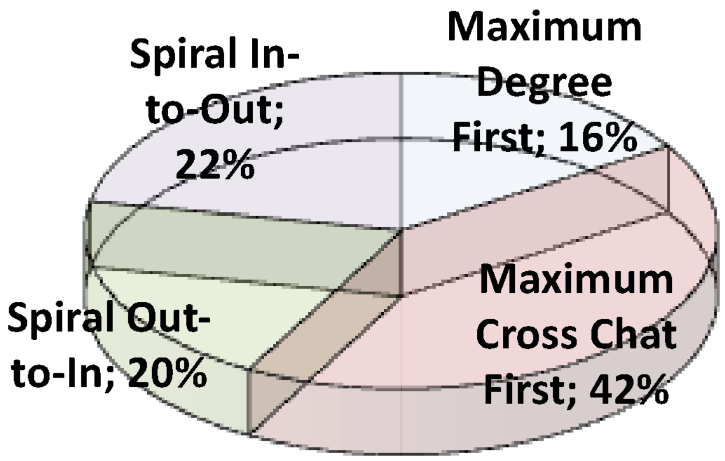


Figure 3.9: Percent Use of Core Selection Strategies

of the low-cost solutions. The most effective solution is generally obtained by selecting tasks by maximum cross-chat relative to the currently mapped tasks.

We further conducted experiments to assess the impact of link contention on communication jitter. We evaluated a 4x4 mesh with 10 randomly generated task sets, each containing 200 messages within their hyper-period. Using the exhaustive solver without time constrains to yield optimal results, we determined that only tests 8 and 10 contained schedules where mappings with zero contended links were found. For all cases, time-frame windows of 10μs were imposed. We then calculated the standard deviations over all 2KB transfers that were issued. Figure 3.11 depicts the standard deviation in clock cycles for different tasks sets for the three mapping approaches. The figure shows that any single contended link can have a significant impact on the standard deviation of transfer latencies. The exhaustive results for all test cases (except for 8 and 10) show that the standard deviation of a system is only affected by cache latencies. Cache warm up is included in these costs and acts as an additive constant on the worst-case transfer times due to additional latencies for data and instruction references. In test cases 8 and 10 the heuristic algorithm shows better performance than the optimal contention solver. The heuristic solver and the optimal solver found solutions with the same amount on contention but the mappings were different. Further analysis into these discrepancies indicated varying performance depending on the path lengths of communication resulting in contention.

HSolver was unable to generate any zero-cost solutions for the benchmarks in our 4x4 configuration but was able to show a reduction in jitter of almost 40% when compared with the naïve mapping. To understand this impact, it is necessary to discuss the amount of time required to converge on a solution within the exhaustive and heuristic approaches. Table 3.2 shows the

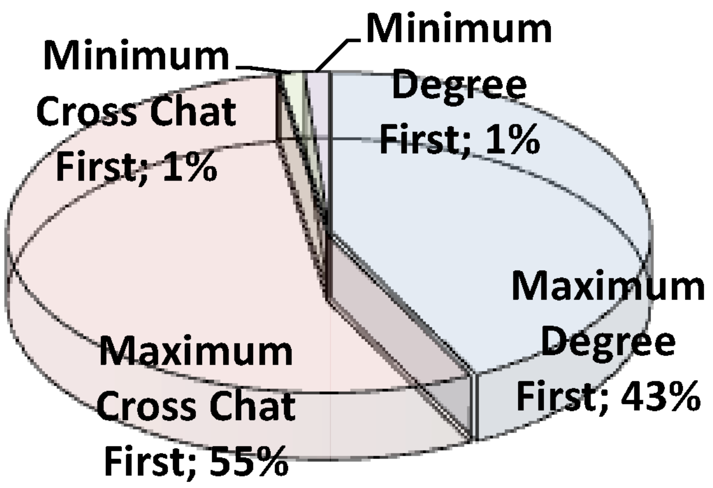


Figure 3.10: Percent Use of Task Selection Strategies

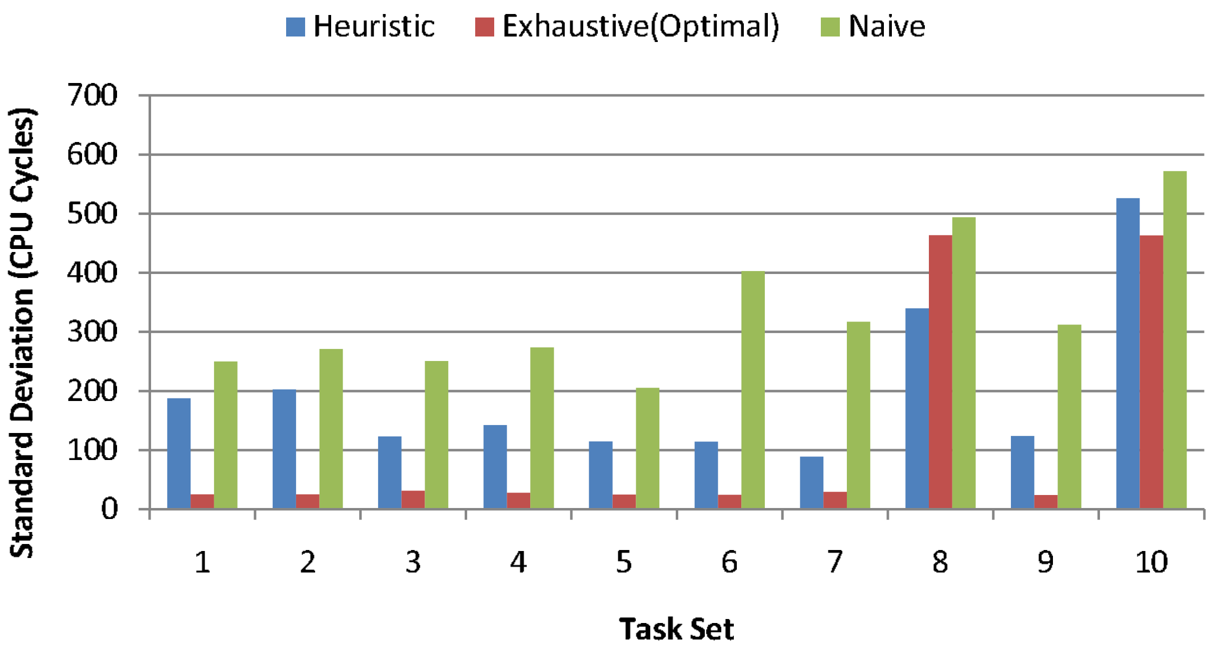


Figure 3.11: Contention Jitter on a 4x4 NoC

Table 3.2: Solving Times per Task Set for a 4x4 Mesh [hh:mm:ss.ms]

Test Case	Heuristic	Exhaustive(Optimal)
1	00:00:00.60	00:00:26.66
2	00:00:00.60	00:04:53.33
3	00:00:00.60	00:02:93.33
4	00:00:00.60	00:02:93.33
5	00:00:00.60	00:01:06.00
6	00:00:00.60	00:00:26.66
7	00:00:00.60	00:00:08.00
8	00:00:00.60	01:10:13.33
9	00:00:00.60	00:00:08.00
10	00:00:00.60	01:00:02.00

timing results for each configuration evaluated in this experiment. All results determined by the heuristic approach converged within fractions of a second. Using the exhaustive solver, convergence can take up to 70 of minutes for solutions with contention. As grid sizes grow, convergence grows exponentially for the exhaustive solver while HSolver's convergence times grow linearly. To help correlate the impact of the results, Figure 3.12 shows the contention jitter as a percentage of the measured worst-case transfer time for the same task sets and solver approaches. These results indicate that naïve mappings can result in jitter of almost 13% of the total worst case transfer time. Jitter of this scale could result in missed deadlines in hard real-time systems due to poor task placement. In comparison, heuristic scheduling reduces that impact by up to 50% while optimal task placement without contention shows less than 1% jitter.

The final experiments illustrate the impact of unavoidable contention on real-time predictability. We previously defined this as input contention (Section 3.2), i.e., two or more cores send to a single receiver in the same time frame. In this scenario, task placement may result in (unavoidable) contention imposed by the application code. Figures 3.13- 3.16 depict the cost for sends and receives for one-to-one and two-to-one pairing of senders/receivers. In taking these results we allowed a cache warm-up period prior to capturing the effects of contention. Senders under one-to-one (Figure 3.13) experience tight WCET send costs of exactly 515 cycles (irrespective of hop counts), only naïve has higher costs as it results in occasional link contention. Senders under two-to-one (Figure 3.15) experience overheads in three bands. Contention-free sends require 515 cycles. The other bands result from input contention for a single packet transfer. When a sender is blocked once under input contention, $\approx$ 1000 cycles are observed. The band around 1500 cycles is due to link contention and input contention. This effect is shown twice for both optimal and heuristic solutions and is a result of a single link contention and input

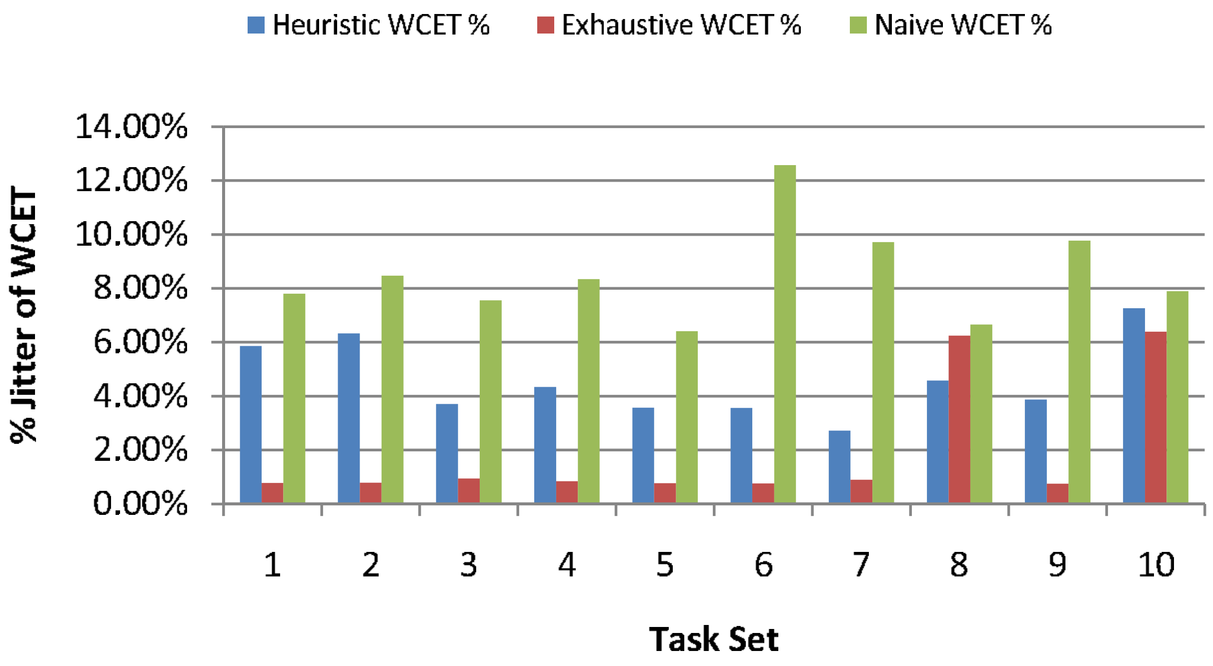


Figure 3.12: Jitter as a Percentage of Worst Case Transfer Time on a 4x4 NoC

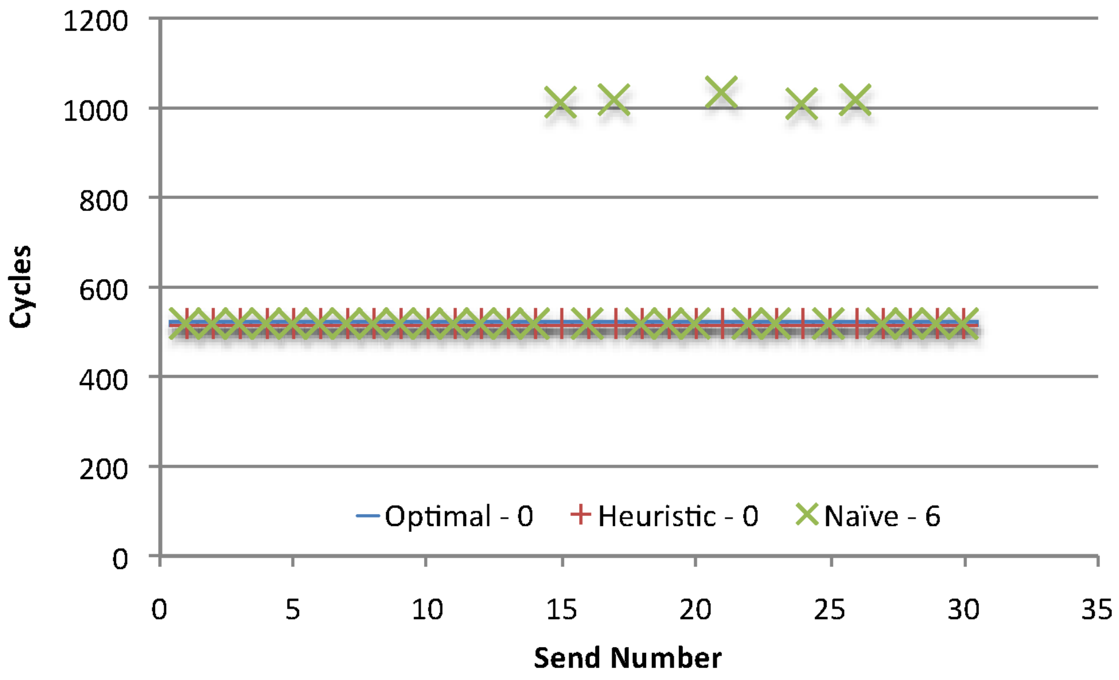


Figure 3.13: Scatter Send One to One on a 4x4 NoC

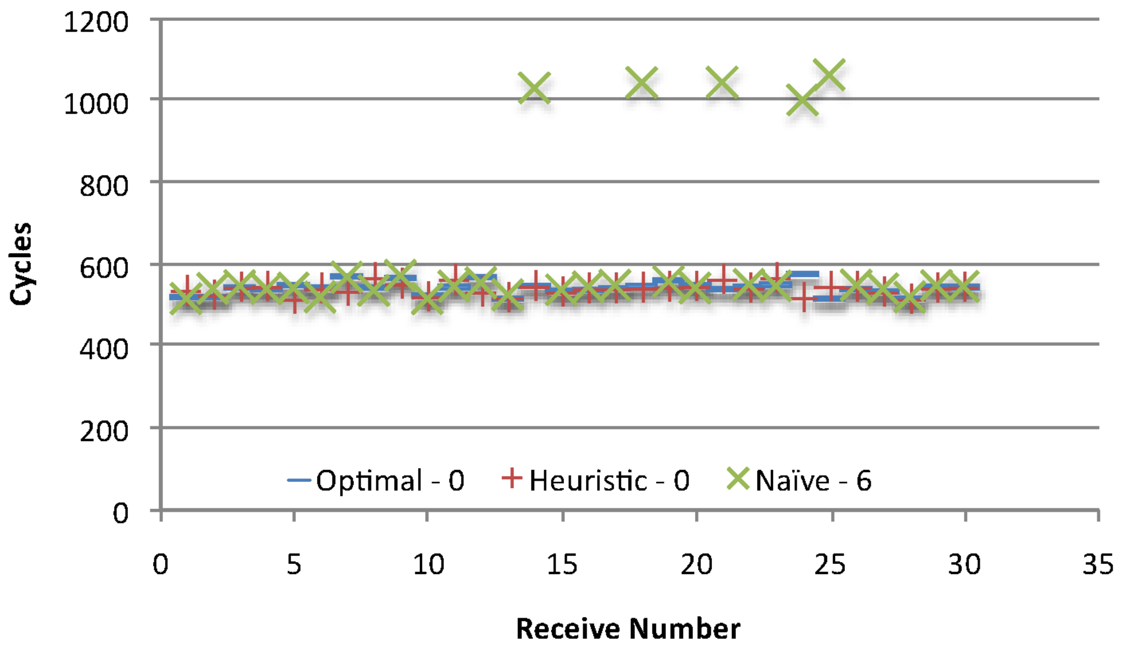


Figure 3.14: Scatter Receive One to One on a 4x4 NoC

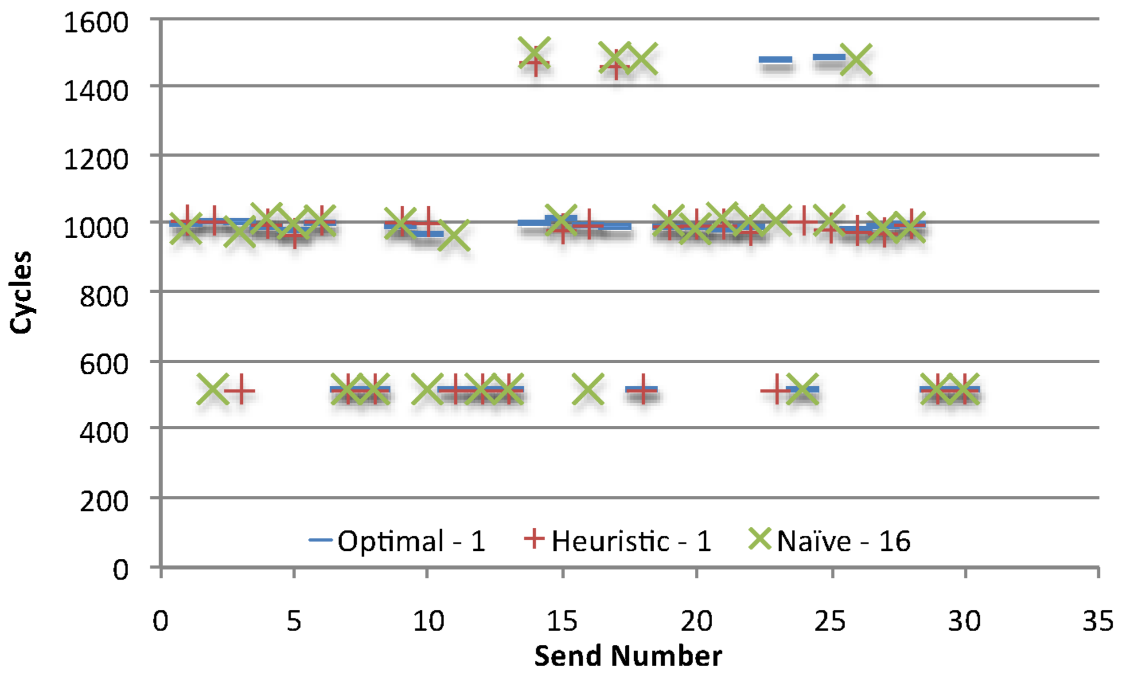


Figure 3.15: Scatter Send Two to One on a 4x4 NoC

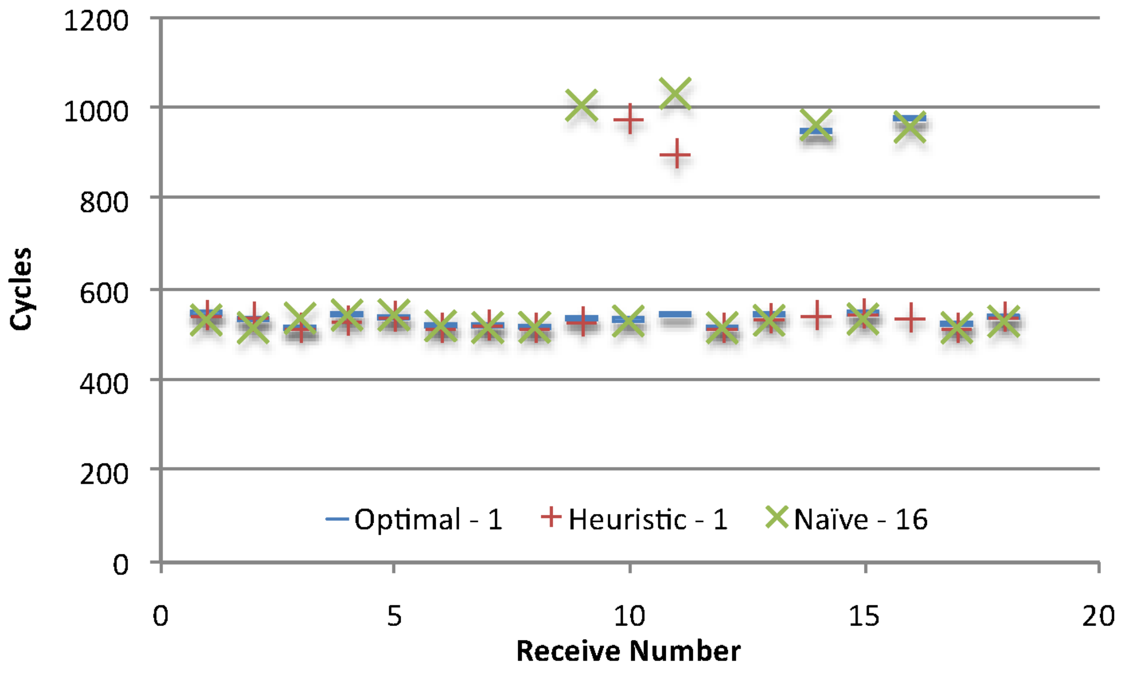


Figure 3.16: Scatter Receive Two to One on a 4x4 NoC

contention on separate cores. This example shows the worst-case experienced over multiple runs and emphasizes the significant impact that contention can have on bounding WCET. In this example the two types of contention discussed lead to an 191% increase in WCET. Figures 3.14 and 3.16 depict the receiver overhead for one-to-one and two-to-one transfers, which is quite uniform for heuristic and optimal (link contention free) with occasional higher cost for naïve due to link contention. Occasionally higher receive costs ($\approx$ 1000) for heuristic and optimal (two cases each) appear to correspond to the 1500 send cases. Overall, overheads can be safely bounded for sends (600/1500 cycles) and receives (600/1100 cycles) per packet (without/with input contention) for hard real-time systems. For soft real-time, tighter bounds of 600/600 and 600/1100 for send and receives, respectively, can be provided.

3.8 Related Work

Bender [11] uses mixed integer linear programming models of heterogeneous multi-cores to solve the task layout problem to guarantee execution time. The solver operates agnostically of the underlying communication mechanism and is unable to model delays due to contention. In contrast, our work operates on a precise model of the underlying network structure for the sole purpose of contention analysis and its effect on execution time. Communication size and time is considered in both their and our work. Murali and De Micheli [48] investigate splitting communication along alternate paths to avoid network based delays. Our work focuses on NoC architectures with static routing, i.e., without alternate path routing, as seen in current multi-cores. Hu/Marculescu as well as Kuchcinski investigate resource allocation of acyclic graphs to provide timing guarantees [30, 38]. These approaches address heterogeneous multi-core architectures and focus on mapping tasks to the correct processor types while our paper addresses homogeneous architectures and focuses on resource mapping to reduce communication overheads.

More recent work by Chou and Marculescu [16] explores intelligent task mapping to reduce contention in order to increase throughput and reduce the number of hops used per communication to reduce energy consumption. They also utilize a solver model but consider communication without temporal properties. This can lead to overly conservative solutions with less flexibility for mappings compared to our approach. Zhu *et al.* [83] create task to core mappings without considering communication. After the mapping is completed, scheduling is then performed on the communication to reduce contention. Both of these papers operate on a class of programs known as streaming data flow (SDF) programs that are generally related to media with soft real-time constraints. Our focus is on hard real-time systems and considers communication first rather than in a second step. Lee *et al.* [39] investigate reducing power and delays that result from contention on a NoC memory network. In contrast, our focus is on predictability.

Stuijk *et al.* [66] look into resource allocation for multi-processor SDFs to increase throughput. This approach uses TDMA to schedule communication into time slices but keeps communication physically disjoint via hardware. Our temporal frames are TDMA-like and require the programmer to comply with frame access constraints in software. As NoC sizes increase, they offer significant real estate to support several simultaneous messages without contention where TDMA approaches significantly limit available bandwidth. Another area in which our work differs from most of these works is that we do not consider the SDF model but instead focus on a hard real-time model. In SDF models, the impetus is on achieving maximum throughput; our work focuses on reduction of contention to increase predictability.

Goossens *et al.* [25] survey contention free routing via TDM slot reservation for both wires and buffers. TDM in the network is realized at the hardware level. Our work is implemented on top of an architecture that does not provide contention avoidance at the hardware level. Their model is more restrictive and costly in terms of power in that TDM hardware will even be used at times when there is no contention on the network. Our software model allows for variable frame sizing to avoid impeding performance in systems with little contention.

In the context of framing, NoC links and routers could potentially be gated when no messages are crossing them to reduce power consumption [28]. Our work could benefit from such an approach, but we focus on predictability for real-time systems instead of power, and we utilize currently available architectures instead of resorting to simulation.

3.9 Conclusion

In this chapter we have designed a temporally aware distributed real-time abstraction for creating contentionless task-to-core mappings. Using constraint programming techniques we modeled an exhaustive solver to determine optimal mapping for solvable NoCs. Based on heuristics derived from solutions to optimal task layouts, we were able to design a multi-heuristic solver, HSolver, that generates fast and low contention solutions for heavily contended NoCs. Compared with naïve and time-constrained exhaustive solving, HSolver was able to reduce aggregate contention by up to 70% while reducing jitter by up to 40% in our experiments. We additionally contributed a micro-benchmark of task system IPC, implemented and evaluated on a Tilera TilePro64, a state-of-the-art NoC processor with 64 cores. We evaluated 100 randomly generated task sets of increasing NoC size, some containing up to 600 messages in NoCs of 8x8 using 60 temporal frames on the TilePro64. Overall, we contributed a compelling method for a novel approach to contention-based modeling of real-time tasks that communicate via messages over a NoC on a multicore processor and demonstrated how predictability can be significantly improved in such environments.

Chapter 4

NoCMsg: Scalable NoC-Based Message Passing

4.1 Introduction

The future of computing is rapidly changing as multicore processors are becoming ubiquitous. While multicores offer tremendous opportunities to meet processing demand, they come at the expense of limited scalability due to on-chip (interconnect) and off-chip (memory) resource contention.

Contemporary shared memory techniques have been shown to fall short in scaling, particularly at the system level where a single system image (SSI) remains the traditional abstraction. SSI was a good match for bus-based multiprocessors in the past. However, bus-based designs do not scale well (even beyond four processors) and have been replaced by mesh interconnects (e.g., Hypertransport, Quick Path Interconnect) and, for high core counts, tile-based architectures with 2D meshed network-on-chip (NoC) interconnects [4, 6, 74, 60, 3].

Many embedded architectures increasingly feature multicores with 4-64 processors, such as ARM's MPCore, Cavium's Octeon, Freescale's QorlQ, and TI's TMS320C80 MVP. Even GPUs (mostly from NVIDIA and AMD) and heterogeneous APUs (AMD Fusion APUs, NVIDIA's Tegra3) with hundreds of compute elements have been considered for embedded/real-time systems [36, 35].

Mesh-based systems with MESI-style (Modified/Exclusive/Shared/Invalid) coherence protocols enhanced by coherence filters [47] may limit scalability in the number of cores. For example, the multikernel (aka. Barrelfish) follows a distributed kernel paradigm that employs messages in an off-chip mesh interconnect of Hypertransport links [10]. It shows that messaging can outperform shared memory for configurations of just eight processors.

Contributions: This work addresses the question if large core counts with 2D mesh NoCs scale better in performance under a shared memory protocol or under NoC-based message passing. It further identifies flow control as a major hurdle in gleaning additional performance from message passing. It presents techniques to identify areas within the implementation of an MPI-like runtime system [26] for NoCs where flow control is not needed, subsequently removes flow control and assesses the impact.

More specifically, the paper details the design and implementation of NoCMsg, a low-level message passing abstraction over NoCs. The design reduces the number of software layers compared to prior work. NoCMsg builds on the abstraction of a distributed memory architecture between cores, i.e., it does not utilize shared **data** memory at all. It is specifically designed for large core counts in 2D meshes.

Its design ensures deadlock free messaging for wormhole Manhattan-path (dimension-ordered) routing over the NoC. This is in contrast to low-level NoC messaging, where limited message buffer space may result in deadlock [7] when a pair of cores sends messages to each other, i.e., they may send flits of messages until all buffers overflow without ever draining them by issuing receives. This results in senders involuntarily stalling their processor pipeline until the transfer can complete. Instead of employing virtual channels that monopolize NoC links between end points, NoCMsg adaptively alternates between sending and receiving by sensing buffer thresholds. Additionally, NoCMsg is able to relax communication constraints by exploiting pattern-based communication common in MPI runtime system implementations to identify areas in which flow control is unnecessary, thereby significantly improving performance of communication, up to 86% for single packet messages and up to 40% for larger messages from our evaluations.

Experimental results on the TilePro hardware platform show that NoCMsg has lower latencies and provides higher throughput for small messages than past NoC-based messaging abstractions. NoCMsg furthermore is more scalable than prior messaging techniques, as shown for a subset of the NAS Parallel Benchmarks [9].

Another significant contribution is given by a comparison between message passing and shared memory on the same NoC architecture, where the former is supported in firmware while the latter is implemented by NoCMsg in software. Experiments demonstrate the potential of NoC messaging to outperform shared memory abstractions, such as OpenMP, up to 85% beyond 16 cores.

4.2 Motivation

In this section, we motivate the necessity of a low-cost deadlock free message-passing library for NoC architectures. We specifically discuss trade-offs that are made in favor of high-throughput communication and their effect on deadlock potentials for the NoC. We also discuss the current state of the art strategies for facilitating flow control.

NoCs utilize traditional network communication for inter-processor communication. Data transferred between cores takes the form of messages. These are broken into fixed sized packets composed of flow control digits (flits). Messages are packetized and transferred via XY dimension-ordered wormhole routing. This design is common to several NoC architectures such as the Godson-T [22] and Tilera Tilepro [6]. Contemporary NoCs feature increasing throughput in communication. This becomes feasible due to simplistic routing protocols to facilitate message passing with low per-flit transfer latencies in the order of a single cycle per hop (without contention).

Unfortunately, without more advanced hardware protocols or structured software libraries, bare-metal message-passing can can lead to deadlocks. As an example, consider wormhole routing on the TilePro64. Wormhole routing describes a packet transfer strategy, where pathways through the switching network are opened by the head of the packet and remain open until the final flit of the packet is seen. The ramifications of this are that packets utilizing switching paths along a currently open path is blocked until this wormhole is closed before continuing. This alone does not result in deadlock as long as packets transfer successfully. The problem arises when SRAM buffers reaches capacity on a receiving switch and its attached core is unable to drain the buffer. When this situation occurs, a packet will be stalled mid-flight, blocking the sender of the packet and any other cores sending data that share any portions of that packet's path.

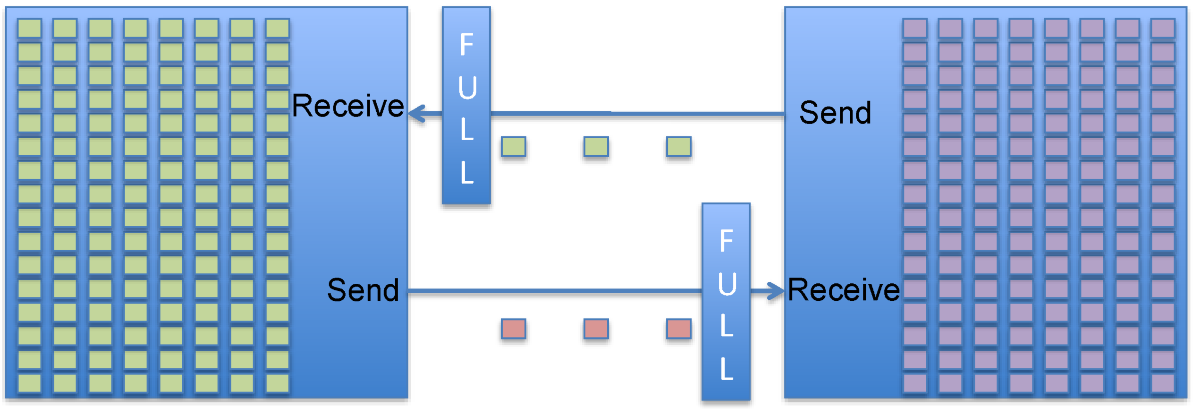


Figure 4.1: Message Passing Deadlock

As an example of deadlock, consider two tasks shown in Figure 4.1 transferring fixed size buffers to each other concurrently. In the Tilera architecture, the receiving tasks can buffer up to 127 words. However, when the buffer becomes full the switching network must wait until flits are drained before transferring any remaining flits. Exchanging contiguous buffers of size greater than 127 words will result in deadlock for the two cores and any messages that need to traverse the route between these two cores. Solutions to avoid blocking commonly involve interrupt-based channel creation to facilitate communication. The iLib communication library on the Tilera uses channel creation through protocol messages to establish a channel. Once the channel is created, the buffer can be transferred. Unfortunately, protocol messages are also subject to deadlock. Hence, the library must provide a timeout interrupt to break out of the communication so that the sending process can drain the network before continuing to send packets.

4.3 Design

Our work assumes a generic, generalized 2D mesh NoC switching architecture often used in many simulations and similar to existing fabricated designs with high core counts. Each core is composed of a compute core, network switch, and local caches. We describe the constraints of such an architecture in the following and discuss its relation to our design of a NoC message layer.

4.3.1 NoC Architecture

NoC architectures use the network-on-chip to replace the conventional system bus or other topologies of connecting cores. This means that all memory, messaging, and IO communication occur over the NoC, often through physically separate networks to reduce contention such as processors from Adapteva [1] features 3 networks and Tilera [6] with 5 networks. The Intel SCC [3] contains only has a single network and does not natively support coherence of this network. For the purpose of this work, we focus on the messaging network. In NoCs, messages are used for inter-processor communication. This deviates from system-bus networks that may only use shared memory as a means of communication. Similar to traditional networks, messages are split into packets containing information for routing within the switching network. A packet contains a payload of data for the recipient. Our work focuses on 2D mesh core layouts. Yet, our contributions to flow control operate irrespective of the switch topology developed.

4.3.2 Cores

A compute core interacts with its switch using input and output queues that are accessed via specialized registers as depicted in Figure 4.2. When the output queue from the core to the switch becomes full, subsequent writes to this queue will stall the pipeline until there is space

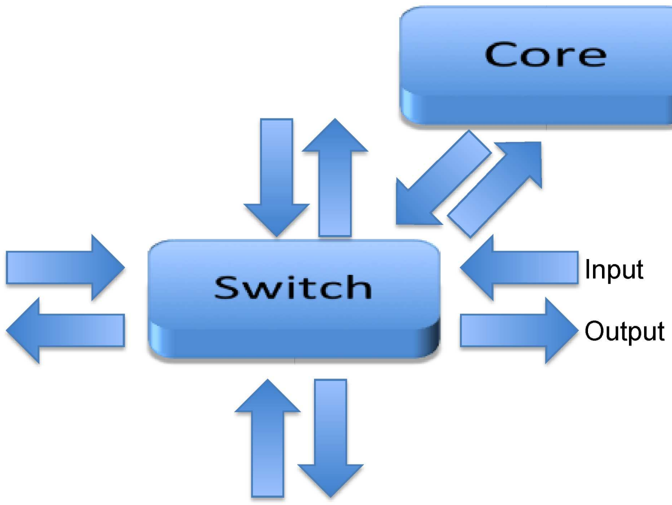


Figure 4.2: Core - Switch Topology

for the write. The inverse also holds: when the input queue is empty and the queue is read, the pipeline stalls until data is available.

4.3.3 Switches

Switches are generally composed of up to multiple sets (five in case of TilePro) of input and output queues attached to a crossbar switch. Each output queue is mapped to input queues of neighboring switches to support the flow of flits. In wormhole networks, header packets create mappings of output queues to input queues as they traverse the network. The mappings are revoked as a switch services the tail flit of the corresponding packet. To enable the detection of open input ports, output ports maintain a set of N transfer credits. When a flit of data is placed into an output queue, a credit is consumed. When that credit is transferred to the subsequent input port, either between the core and switch or between two separate switches, the credit is refunded. Credits are checked when an internal mapping is established between an input queue and an output queue. If the output queue is unable to receive any data due to a lack of credits, no additional output data may be transferred until credits are refunded. This is shown in Figure 4.3 where core one is sending a message to core four. Using XY dimension-ordered routing, this message passes from core one's output queue to switch one's east output queue. Each post decrements a credit when the flit of data enters the queue. Switch one's east output queue will then transfer to switch two's input queue, and switch two will set the cross bar to transfer the packet to switch two's south output queue, if enough credits exist in the south output queue. Subsequently, switch four's northern input queue will receive the flits from switch two's southern output queue and refund credits. Switch four will then create a mapping of the northern input queue onto the core's input queue. Incoming flits into core four's input queue will be automatically buffered in a larger SRAM FIFO buffer until no more data can be transferred.

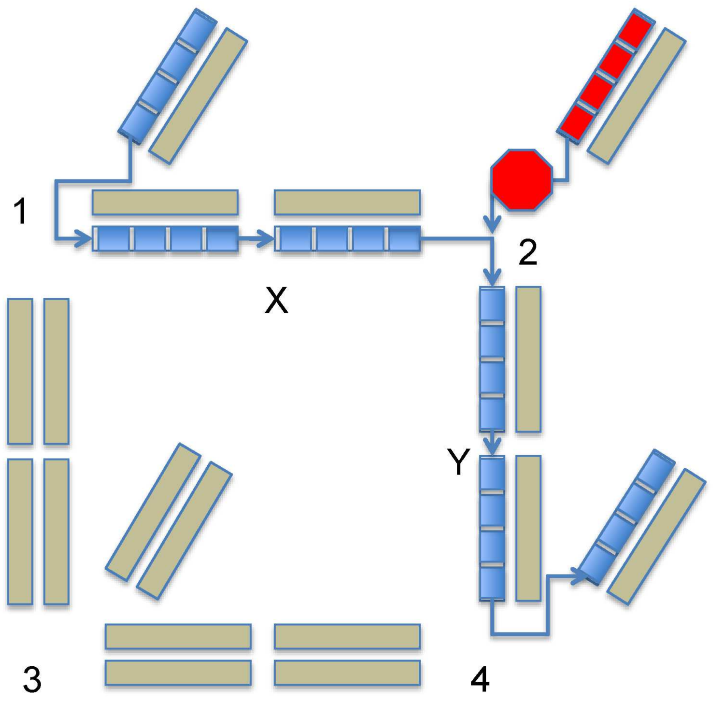


Figure 4.3: Path-Based Back Pressure

4.3.4 Back pressure Checking via Credit Monitoring

As discussed before, this work assumes that output queues maintain a series of credits. These must be greater than zero for more data to be added to the queue; otherwise, the pipeline will stall. *It is these credits that make up the basis of our flow control technique.* We assert that by checking credits on the sender side's output queue we can avoid deadlock and reduce the cost of sending messages over virtual channel flow control techniques. In the following, we characterize back-flow resulting from two types of blocking in the network. The first is receiver-side buffer blocking. In this situation, the receiver-side SRAM buffer has reached capacity and is unable to accept any more data. This implies that the receiver-side local input queues are unable to move any data into SRAM, effectively halting refunds of queue credits to the output queues on the previous core in the path. Figure 4.3 gives an example, where node four is unable to receive any more data. This backup can only be resolved if node four actively drains the network to free up space within its hardware buffers. The second type of back pressure in Figure 4.3 represents a switch unable to route a packet due to a open worm-hole path. The message sent between cores one and four is blocking a message being sent from cores two to four. In this situation, the only way to resolve blocking is to ensure the message between one and four completes.

The design follows this approach in a generalized fashion via a polling work loop that cycles between computation, sending, and receiving of data. The underlying credit checking scheme is specific to Tilera, other vendors may provide a different resource management approach that can be put in its place. For example, non-blocking processors may set co-processor failure registers when data cannot be added to the network. An equivalent resource monitoring approach can be applied to a variety of NoC architectures that provide architectural feed-back of failure conditions. Work loops provide a solid strategy for balancing communication and computation within the cores without costly interrupt service routines and buffers protected by locks.

4.4 Implementation

This section provides details on the implementation of NoCMsg. A central point is the integration of credit-checking flow control into an MPI-like API, yet with modified semantics. Such semantic changes enable selective flow control when the application or MPI runtime routines allow the omission of flow checking, which can result in significant performance improvements.

The difference between flow- and non-flow control communication is demonstrated in the following function prototypes. These prototypes also underline the close resemblance of the NoCMsg API compared to the MPI API. The NoCMsg prototypes following the signatures of MPI calls. A regular "Send" operation even mimics the flow control constraints (in terms of blocking requirements) of the equivalent MPI call. In contrast, "Xsend" eliminates flow control altogether, i.e., it differs fundamentally in the underlying semantics and operates at the architectural level instead of utilizing operating system / MPI runtime capabilities.

```
TILE_Send(void *buf, uint32_t size,
    TILE_Datatype dt, uint32_t dest,
    TILE_Comm comm)



TILE_Xsend(void *buf, uint32_t size,
    TILE_Datatype dt, uint32_t dest,
    TILE_Comm comm, bool sync)
```

We implemented NoCMsg on the Tilera Tilepro 64, yet our general design extends to any 2D mesh NoC architectures as described in Section 4.3.

4.4.1 Point-to-Point Messages

The basis of NoCMsg is factored around asynchronous work loops during which sends and receives are issued based on the availability of resources. Asynchronous communication forms the basis of all communication in NoCMsg as a building block for synchronous communication, collective operations, and barriers. As previously described, point-to-point messages are subject to deadlock in the absence of flow control due to the nature of the NoC switching architecture. As such, we employ back pressure monitoring to ensure absence of deadlock for any message transactions. This is ensured by implementing a work loop broken into two alternating operations for asynchronous communication. (1) Trysend implements conditional sending of a message. During the send of a packet of flits, the output queue's available credits are inspected. We then place as many flits in the output queue as credits are available, i.e., credits are queried for each an every transfer. If no credits are left, control is returned to the work loop. (2) Tryreceive implements conditional data reception. The MPI-1 ready-send specification for point-to-point sends and receives requires synchronization between any send/receive pairs [26]. For synchronous communication, this means a send will not be completed until the sender has seen an acknowledgment from the receiver. In asynchronous communication, send and receives will initiate communication, yet may complete before completing it. Should a matching sender-side MPI_Wait() call follow, then a similar acknowledgment has to first be seen by the sender. An MPI_Wait() after an asynchronous receive, on the other hand, simply indicates that the receive completed. These requirements for acknowledgments and completion of calls ensure ordering within the packet stream with respect to a given sender/receiver pair. When MPI_Wait calls are present, this can be exploited for flow control elimination.

NoCMsg implements introduces so-called synchronous non-flow controlled messages that diverge from MPI in terms of their semantics. Its objective is to exploit common communication patterns found in the implementation of collectives within the message-passing runtime but also in application codes. NoCMsg supports synchronous non-flow controlled communication for send and receive operations for (a) regions between collective communication and (b) within the implementation barriers if no flow control is required. We identify these patterns based on (a) the communication object of collectives and (b) the analysis of communication patterns in benchmarks.

The implementation of non-flow controlled transfers requires a small setup overhead to synchronize the sender and receiver if the buffer is larger than a packet. This is shown in the code presented above: The non-flow controlled calls feature a synchronization boolean and execute a send that by-passes any credit checking. This has the side effect of avoiding data congestion, which increases performance. After this synchronization, full messages can be transferred without the use of any interrupts or credit checking. A drawback of exposing flow-control free operations is that semantic correctness, when utilized, is not dynamically checked. A developer may choose this capability as a means of optimization and subsequently introduce errors to the program logic that may result in communication deadlock. Tools to validate correctness when substituting API calls with equivalent non-flow-controlled ones are beyond the scope of this work.

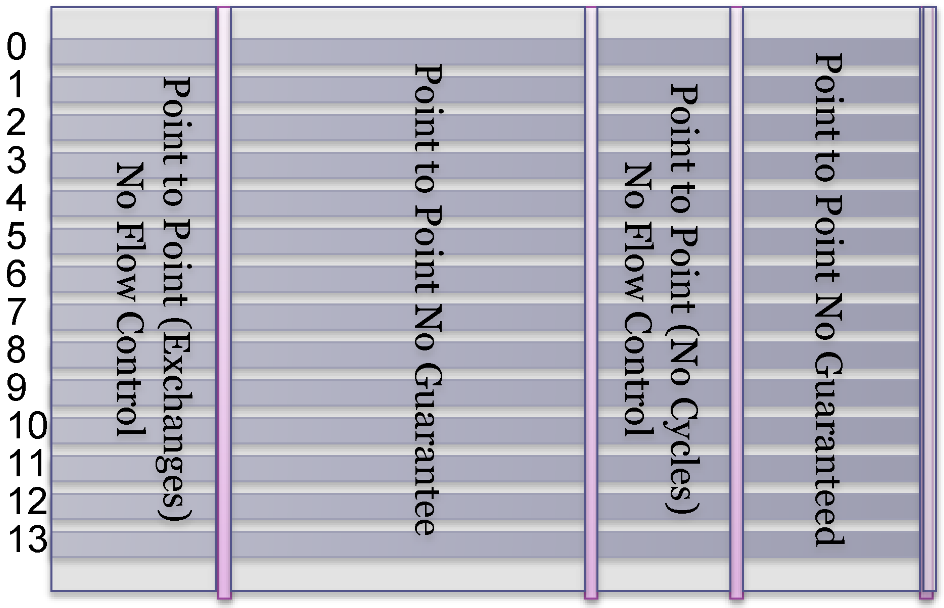

0
1
2
3
4
5
6
7
8
9
10
11
12
13
Point to Point (Exchanges) No Flow Control
Point to Point No Guarantee
Point to Point (No Cycles) No Flow Control
Point to Point No Guaranteed

Figure 4.4: Profile Detected Communication Regions

The challenge in non-flow controlled transfers is to identify if flow control can be safely removed. To this end, applications are profiled to identify regions of code with suitable communication patterns. A NoCMsg profiling run produces information about sender and receiver, code region mapping, and communication type, *i.e.*, synchronous or asynchronous. This data is then used to construct unique communication flow graphs for each region, where collective operations and barriers mark region boundaries. This data is then analyzed to detect communication patterns that inhibit flow-control elimination (e.g., due to cycles). The approach is conservative in that regions that *may* require flow control are excluded when in question. For example, asynchronous communication that crosses a collective, i.e., an asynchronous send before the barrier on one side with a matching receive after the corresponding barrier on the other side will be excluded. In contrast, special patterns, such as pairwise exchanges (send/receive pairs) are detected and subsequently optimized via ordering by rank to eliminate flow control. Figure 4.4 shows an example of a set of detected patterns. In the figure, bars show barriers or collectives that separate different regions of code. Regions of code are marked to indicate the type of communication they contain.

4.4.2 Collectives

Collective operations offer significant opportunities to eliminate flow control since one can make safe assertions about the content of the network messages in flight at a given point in time for NoC communication. This assumes that the NoCMsg program is the only program executing on the NoC (or, at the very least, is contained in a hard-walled NoC grid) effectively isolating the grid network ports. Collectives communicate data among all processes of a group. As an example, consider two common collectives, broadcast and reduction. Their semantics require no flow control to exchange messages. The first criterion for flow control elimination is that there is a single known sender or a single known receiver. Broadcast and reduction meet this criterion, respectively. The second criterion is the presence of synchronization prior to the collective and that no asynchronous communication is in flight. This guarantees absence of in-flight point-to-point messages before non-monitored message transfers begin.

Alltoall and alltoallv are the most demanding collectives, in terms of network contention, that we implement in NoCMsg. These collectives also allow the elimination of flow control. Based on the particular internal send and receive orders in these collectives, it is possible to guarantee flow-control free communication for the transfers from each core to every other core. In our implementation of both collectives, a single receiver is acquiring data from all cores at any given time. Thus, no deadlock is possible as all processes are involved in acyclic communication.

4.4.3 Barriers

Our current implementations of collectives requires prior synchronization of execution for deadlock free communication. We have created a new barrier interface specifically for this purpose that also improves performance over a shared memory barrier design. In order to provide scalable barriers, we implemented tree-based barriers that distribute the work evenly among nodes and to minimize the cycle differences upon barrier completion. Our Tilera implementation utilizes rooted n-ary trees to this end. The root of this tree is placed in the center of the NoCMsg grid to minimize latency (hops). The process of synchronization is simple: Children notify their parents when they have entered the barrier, up to the root. Once the root has received notifications from all children, it broadcasts a notification back down the tree by sending to its children and exits, as do the children. To guarantee isolation for processes that have not yet entered the barrier, we use a separate SRAM buffer. This also eliminates the need to use the standard packet header, which would unnecessarily increase the size of a synchronization packet. Flow control is not needed in the barrier as the prerequisite of entering into the barrier is that all outstanding sends and receives on the local core have completed. The synchronization packet is small enough to fit into the output queue, *i.e.* the core can drop an entire synchronization packet into its output queue. It can subsequently begin a blocking send operation that will halt the core's pipeline until synchronization packets become available. This technique significantly reduces the cost of synchronization when all cores are ready, as shown in the experimental results.

4.4.4 Network Partitioning

A consideration that must be made with the use of any flow-control elimination is network partitioning. The techniques discussed in this work assume run-to-completion tasks and absence of cross communication from outside task sets. This does not mean that outside task sets are unable to use these links, only that they cannot address messages to nodes within an external task set. However, utilization of these switches by outside task sets can create additional communication contention affecting performance. With this in mind, tasks are mapped within grids to reduce the possibility of perturbation of results by parallel task deployments in the experimental section.

4.5 Framework

Experiments were conducted on a Tilera TilePro processor, namely a 700MHz 64-core version (TilePro64) with floating point emulation in software. Programs were compiled with Tilera's MDE 3.03 tool chain at the O3 optimization level with Tilera's C/C++/Fortran compilers that also support OpenMP.

OpenMP experiments run with enabled coherence (L3 on). The L3 is called virtual since the processor has local L1 and L2 caches, where portions of the L2 caches of all cores can optionally be combined into a distributed (virtual) L3 cache. The L3 cache is directory based (uses address hashing) and supported by the memory dynamic network (MDN). Notice that the MDN has twice the bandwidth of the user dynamic network (UDN), which puts NoCMsg at a bandwidth disadvantage relative to shared memory.

Experiments with NoCMsg and OperaMPI were conducted under disabled cache coherence (hash-based distributed virtual L3 turned off). All messages are routed over the UDN.

OperaMPI [34] implements the MPI 1.2 standard [26] for C. It is layered over Tilera's iLib, an inter-tile communication library that communicates over the UDN. We ported iLib and OperaMPI from MDE 2.0 to MDE 3.03 to allow a fair comparison. We also extended OperaMPI with Fortran wrappers to make it Fortran compatible.

The iLib library is vendor-supplied and allows developers to easily take advantage of many of the features provided by the Tilera architecture. Message passing is one of these features. point-to-point messages are directly supported by iLib closely resembling the equivalent MPI semantics. Internally, iLib utilizes interrupt-based virtual channels and complex packet encodings to synchronize senders and receivers to set up such point-to-point communication. However, iLib only supports a limited number of collective operations, namely broadcast and barrier. Hence, OperaMPI creates virtual overlaps (e.g., trees for reductions) to implement more complex MPI collectives such as all-to-all communication, all-gather/scatters, reductions etc.

Experiments were conducted for the NAS Parallel Benchmarks (NPB) [9] Version 3.3 for OpenMP, OperaMPI and NoCMsg. Inputs were modified to allow weak scaling [27] within L2 sizes: As the number of cores is increased, overall problem input sizes are proportionally increased as well so that the core-specific data remains constant and fits into the L2 cache of a local core. Constraining the problem to L2 exposes the overheads of NoC-level communication for these benchmarks without being skewed by off-chip memory references, which otherwise dominate.

We also constructed a term frequency/inverse document frequency (TF*IDF) benchmark for document clustering based on prior work [82], which follows a map-reduce paradigm [20]. Its inputs also follow the weak scaling paradigm for L2 resident data sets.

4.6 Experimental Results

We evaluate NoCMsg by first comparing shared memory and message passing using micro benchmarks on the Tilera. More specifically, we refer to shared data memory whenever talk about shared memory here. In the evaluated benchmarks, instructions can still be shared with little to no impact on other executing applications since we warm up the instruction cache so that nearly all of the instruction references hit in L1 cache since it is sufficiently large.

Next, we compare NoCMsg with OpenMP using the NPB suite. We then compare NoCMsg to OperaMPI, an MPI library specific to the Tilera platform, for the NPB codes. Finally, we evaluate TF*IDF, a document clustering algorithm, by comparing NoCMsg to both OpenMP and OperaMPI.

4.6.1 Microbenchmarks

We have conducted experiments to assess the tradeoffs between message passing and shared memory of a 64-core NoC (TilePro64) [6], which has multiple mesh networks (on chip). In a bandwidth micro-benchmark, we measured the transfer time in cycles for different data sizes. We compared message passing over the UDN with shared memory transfers over the coherence interconnect. Figure 4.5 indicates that shared memory incurs roughly twice the cost of message passing transfers (both without hashing). UDN messages follow a one-sided push model (sender initiated) while shared memory accesses are pull based (receiver initiated) and require at least two messages for a single transfer. Hash-based distributed caches reduce the shared memory overhead but the overhead still remains higher than sending messages without hashing, especially for larger transfers. (Notice that hashing interferes with larger messages while reducing overhead for shorter ones as long as the transfer data fits into local caches.) The differences between shared memory and message passing become even more significant as the distance (hop count) between cores in the NoC increases and as NoC contention increases. *These results indicate that message passing has the potential to outperform shared memory transfers and that the former has superior scaling characteristics than the latter.*

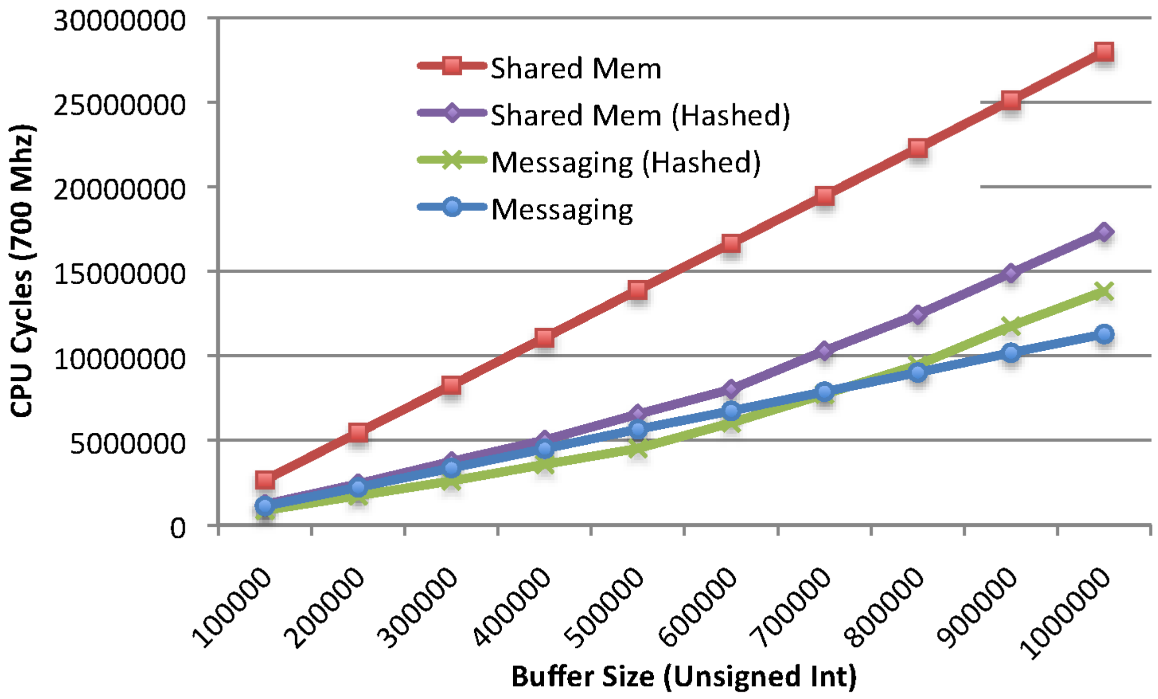


Figure 4.5: Shared Memory/Messages NoC Bandwidth

The Tilera processor designates a set of tiles as a distributed L3 cache by effectively "combining" multiple L2 caches via its coherence interconnect and protocol.[1] This results in a uniformly distributed address space over this virtual L3 where core affinity is determined by

[1]The vendor calls it a distributed L2 cache. Since the architecture can also be used with just the local L2 cache, we refer to it as a L3 cache to avoid confusion.

a hash function. Hashing can thus significantly increase the performance of shared memory by reducing the average distance to cached data and by increasing cache capacity of L3 to the aggregate of all L2 caches. However, this performance increase does not come for free. Even accesses to small data structures that might otherwise fit into L2 are redirected to remote L3. *This performance win comes at the cost of jitter since accesses to distributed L3 have variable hop counts over the NoC.* Figure 4.6 shows the effects of data transfer in terms of jitter. In both message and shared memory transfers where home cache hashing is turned on, there is noticeable jitter even in the absence of additional contention due to additional tasks. In a shared-memory system design, both the operating system and the application increasingly suffer from such latencies as the core count increases. While the total amount of jitter may be small for single-threaded code, jitter has the potential to aggregate as the number of cores increases. This may result in unbalanced execution where more and more cores remain idle prior to global synchronization (e.g., barriers). We term this effect *perturbation* and discuss it in the following set of experiments.

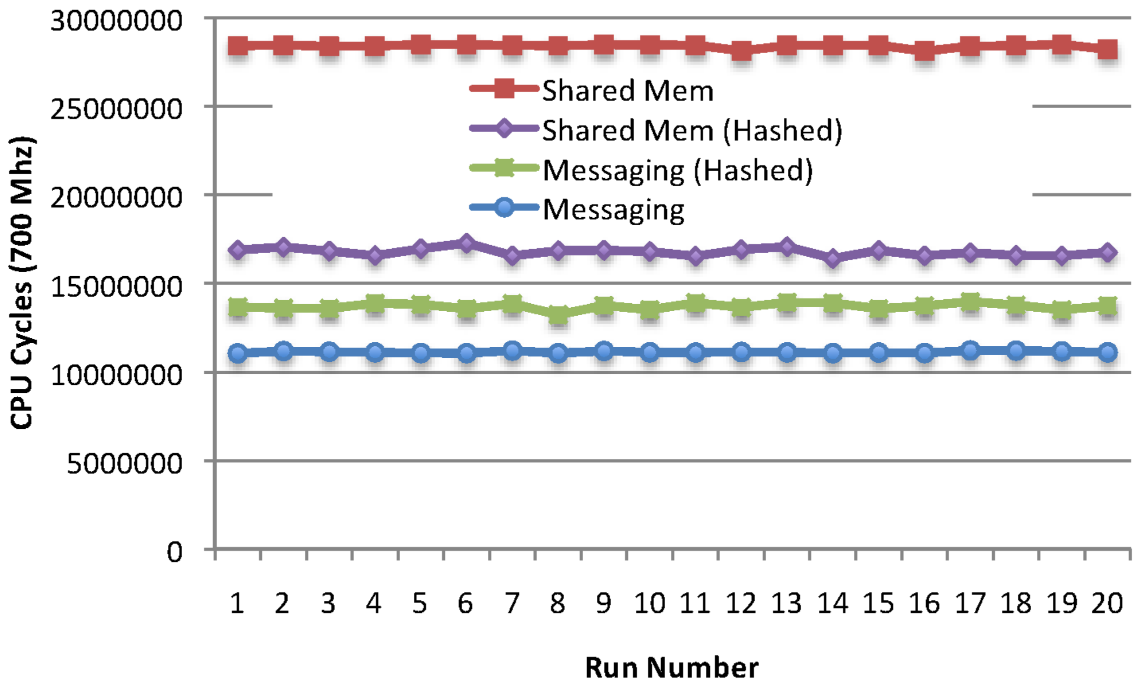


Figure 4.6: Shared Memory/Messages NoC Jitter [1MB Transfer]

4.6.2 NAS Parallel Benchmarks

The micro-benchmarks show the tradeoffs between shared memory and message passing. To assess these affects using real-world benchmarks, we evaluated several of the NPB codes on the TilePro 64 over (a) shared memory (OpenMP) and (b) NoCMsg. We chose NPB since OpenMP and MPI versions exist for each code, much in contrast to other parallel benchmarks that only provide shared memory codes. In contrast to NPB's default strong scaling inputs, we used our own weak scaling inputs [27] where the number of keys per core is a fixed size. This weak scaling input size is shown on the secondary Y-axis in each of the following figures. Weak scaling ensures that the computational work per core remains the same as the number of cores cooperating in a parallel application is increased. Note that all of these benchmarks except IS operate on floating point or complex data types. The TilePro 64 does not contain any floating point pipelines, *i.e.*, floating point calculations are performed using software emulation. This leads to more time spent in computation vs. inter-processor communication, which gives shared memory an advantage (due to a reduced fraction of communication) over message passing according to the micro-benchmark results.

Figures 4.7 and 4.8 show results for the two NPB codes LU and SP classified as pseudo-applications. LU and SP solve non-linear partial differential equations using standard solver techniques. In both benchmarks, the weak scaling input is 4KB per core. The shared memory network provides faster performance than message passing for low core counts. This is due to the fact that the shared memory network has twice the bandwidth of the UDN (for messages). At 16 cores, inter-processor communication and L3 contention start to hurt performance and begin to show the effects of perturbation, indicated by the range of execution times depicted through the error bars. For LU at 32 cores, perturbation becomes more frequent. For SP at 49 cores, the worst measured perturbation is almost 50% greater than the average performance. The perturbation shown across all of these results is caused by increased wait times for shared memory accesses as inter-processor communication increases with core count. It ultimately results in unbalanced computation and idle cores before global synchronization via collectives, e.g., barriers.

We next evaluate CG and FT. CG estimates eigenvalues using the conjugate gradient method. FT is a Fast Fourier Transform solver for partial differential equations. The results for CG and FT in Figures 4.9 and 4.10 are very similar to those of LU and SP. However, CG and FT exhibit less computation and more inter-processor communication. Both benchmarks show that inter-processor communication eventually dominates results under core scaling and considering the predictability (or lack thereof, i.e., perturbation) of these applications, even though a significant amount of computational power is expended on software emulation of floating point operations. OpenMP thus shows significantly worse performance and larger perturbation (er-

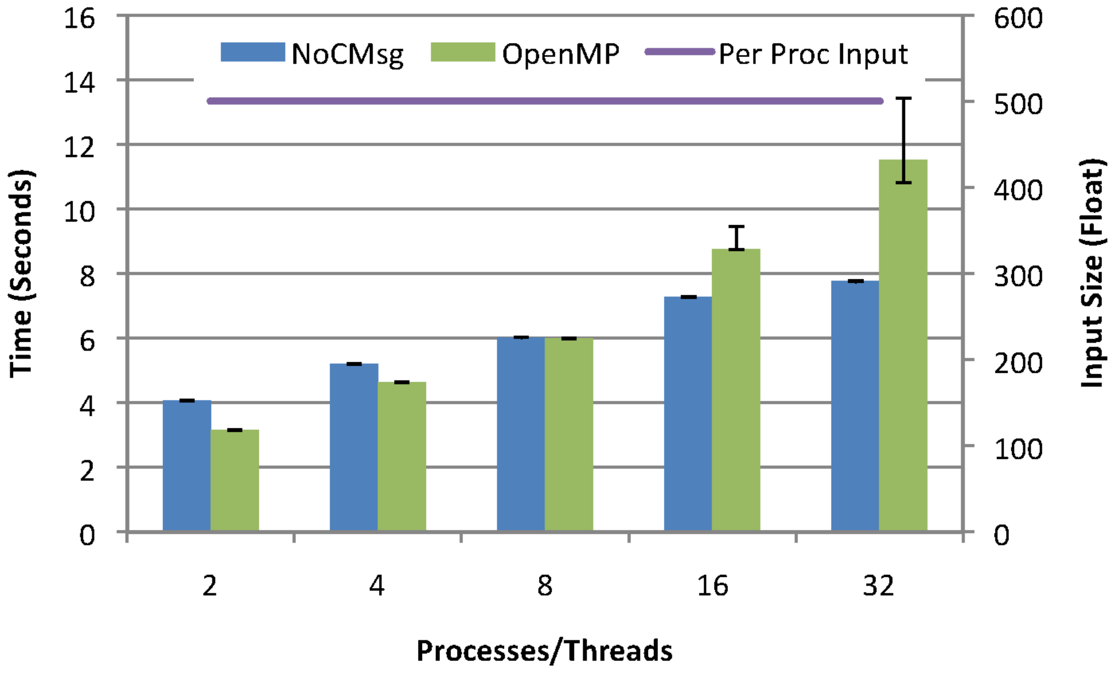


Figure 4.7: NPB LU: Weak Scaling

ror bars) for higher core counts. The perturbation results from L3 contention in both of these benchmarks that becomes dominant at 16 and 32 cores.

Figure 4.11 depicts the results for MG, a multigrid approximation benchmark for discrete Poisson equations. MG is the only benchmark without enough inter-processor communication to generate an affect on performance. This benchmark was extremely limited in sizes due to a communication pattern that grew with the number of processes. It is also an extremely memory intensive benchmark resulting in large performance benefits of OpenMP over NoCMsg at two cores. But their benefits rapidly diminish at larger core counts. While this benchmark did not show performance improvements when comparing OpenMP to NoCMsg, it did show a the trend toward high perturbation under OpenMP. This indicates that subsequent increases in process/thread count beyond 32 might lead to decreased performance for MG, just as in the other NPB codes. Unfortunately, due to hardware limitations and power of two limitations in core counts of the MG code, we were unable to test at 64 processes/threads.

Results for the final NPB code IS, an integer bucket sort benchmark, are depicted in Figure 4.12. IS features high amounts of collective inter-processor communication. It is also the only integer benchmark in the NPB suite. This explains the reduced amount of time spent in computationally demanding portions of the code for NoCMsg. It also shows that high inter-processor communication leads to significantly higher performance and lower perturbation of

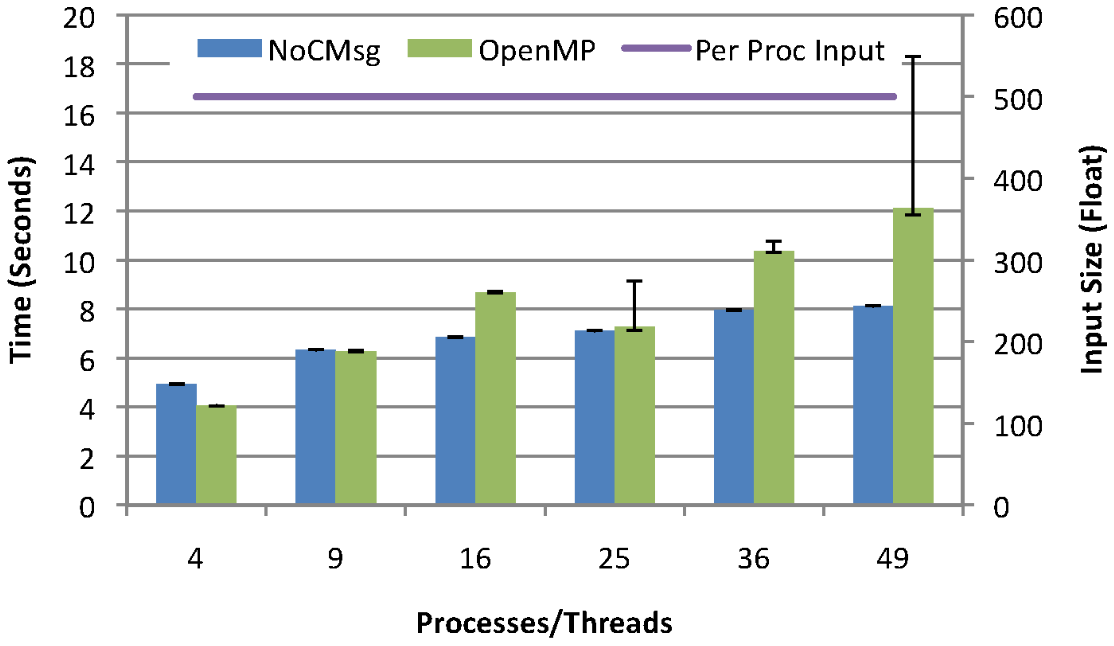


Figure 4.8: NPB SP: Weak Scaling

NoCMsg starting at just 8 processors. The execution time under OpenMP increases quadratically while that under NoCMsg remains close to linear as the processor count increases. The performance differences for IS underlines the potential of this architecture for other codes (NPB and beyond). If a pipelined floating point ALU were added, the computational performance of these benchmarks would increase significantly creating an even even wider gap between OpenMP and NoCMsg as communication would become more dominant relative to computation.

4.6.3 Flow Control Elimination

Our next set of experiments focuses on the elimination of flow control for several collectives for easily identifiable areas of the NPB codes. Initial findings indicate that while our flow-control method is portable, synchronization requirements within the MPI specification coupled with flow control resulted in NoCMsg and the interrupt based OperaMPI to perform at par for virtually all of the benchmarks. However, as detailed in the design section, MPI and point-to-point communication offers several opportunities to relax synchronization constraints by employing flow-control free communication. Figures 4.13, 4.14, and 4.15 show the benchmark results just for the communication time of FT, CG and IS after varying amounts of flow control were removed in a safe/conservative manner (cf. design section).

The primary communication in FT is an alltoall collective. Such collectives allow elimina-

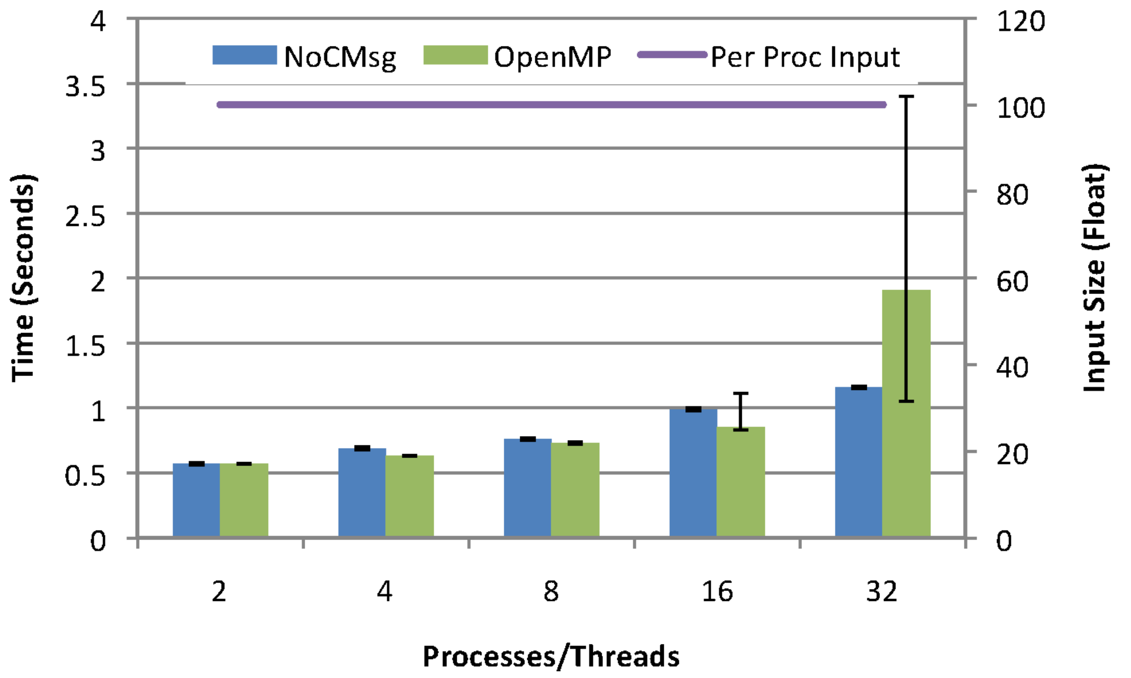


Figure 4.9: NPB CG: Weak Scaling

tion of flow control since all processes participate. After eliminating flow control, significant improvements to the communication performance of NoCMsg were observed (see Figure 4.13). The primary reason for the scalability of NoCMsg is that the minimum cost transfer is very small for flow-control free communication (on the order of just a few cycles). OperaMPI incurs much higher overheads (factor $7X - 8X$) due to interrupts and protocol messages.

Figure 4.14 shows communication time results for CG. CG has several regions where synchronized MPI communication can be replaced with flow-control free communication. Since CG exclusively transfers data as a series of exchanges, it can guarantee that flow control free communication can be utilized, i.e., message ordering is guaranteed due to the application and NoC characteristics. By replacing these regions with flow-control free exchanges, improvements up to 40% are observed for NoCMsg at 32 processes. Notice that there is a synchronization requirement in CG when transitioning from 8 to 16 processes due to a changing communication pattern resulting in a significant increase in communication cost due to additional synchronization messages. From 16 to 32 processors, communication times stabilize again.

Results for IS are depicted in Figure 4.15. IS features several patterns where the amount of flow control can be reduced without major modification to the application. The most significant one is in the implementation of the alltoallv collective. This function represents a majority of communication in IS. Our flow control elimination results in an 38% improvement

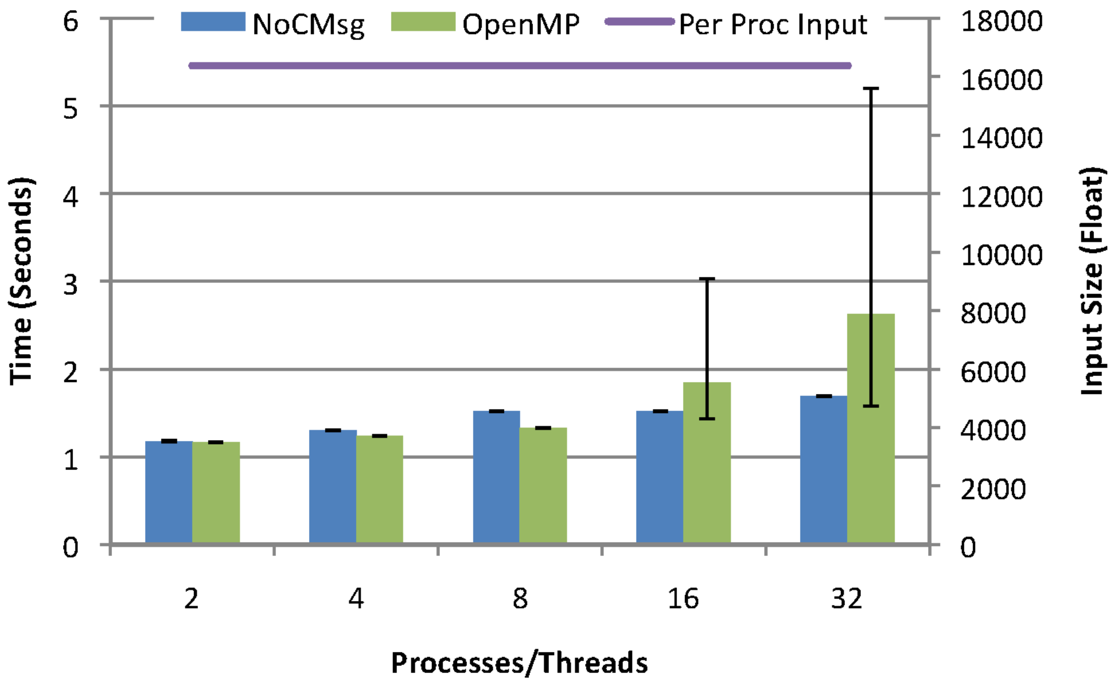


Figure 4.10: NPB FT: Weak Scaling

in communication performance at 32 processes.

For the remaining NPB codes, the communication patterns and use of collectives provided limited opportunity to eliminate flow control. In these codes, the performance behavior is dominated by MPI synchronization and flow control. Hence, we observe equivalent communication times for OperaMPI and NoCMsg for SP, LU, and MG and omit figures due to that fact.

In our final result, we evaluate TF*IDF. TF*IDF is a document classification technique for identifying important terms over large amounts of documents. TF*IDF is broken into two separate algorithms. TF (term-frequency) classifies unique terms and their occurrence frequencies on a per-file basis. IDF (inverse document frequency) combines TF data and accounts for term frequencies over the full set of documents. This problem is traditionally used in data mining. Two challenges in this problem are the large amount of required dynamic memory allocation and the reduction of IDF data in a parallel implementation.

In our first TF*IDF experiment, we compared the wall-clock time for NoCMsg to OpenMP (see Figure 4.16). We observe a disparity between performance that is almost a factor of 9X at 20 processes. This is primarily due to the required synchronization for heap allocation (C++ new) of STL calls for OpenMP. Heap allocation is protected by a lock to ensure thread safety. This lock contention results in inferior scalability under OpenMP due to increasing number of threads contending for the lock by spinning on a shared memory location, which results in

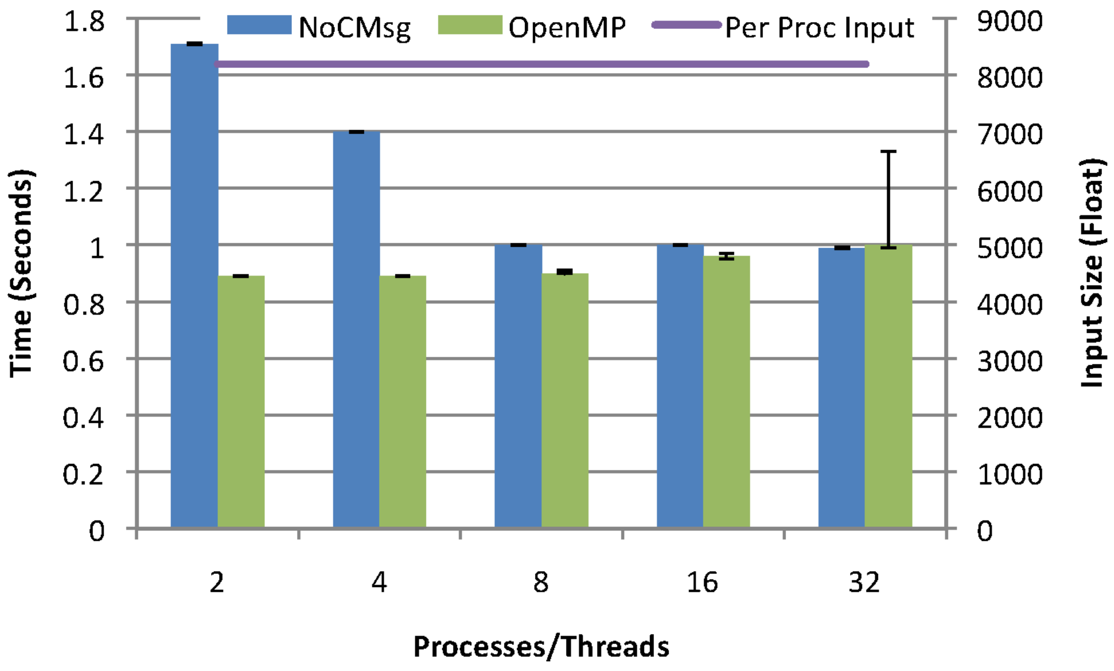


Figure 4.11: NPB MG: Weak Scaling

high coherence protocol traffic. NoCMsg does not experience this problem since its execution paradigm is a distributed one with separate address spaces.

The problem for OpenMP could be addressed algorithmically by pre-allocating heap data at initialization time (such as for the NPB codes). Unfortunately, the nature of the TF*IDF algorithm does not adhere itself to pre-allocated data as data structures are dynamically determined and allocated, which is common for many C++/STL codes. One could implement private heaps non-shared heap strategy until the for the TF calculation, yet would have to switch to a global one for IDF, where the problem remains for the latter. We did not go this route as we wanted to assess the benefits of TF*IDF without excessive changes to the application or system libraries.

Our second TF*IDF experiment compares the communication costs (see Figure 4.17) between NoCMsg and OperaMPI. Since TF*IDF largely works on map-type data structures containing terms and their frequencies, the data must be serialized for interprocessor communication. This communication is structured as a tree-based reduction where flow control is not necessary. This is largely responsible for the 12% improvement for NoCMsg at 20 processes.

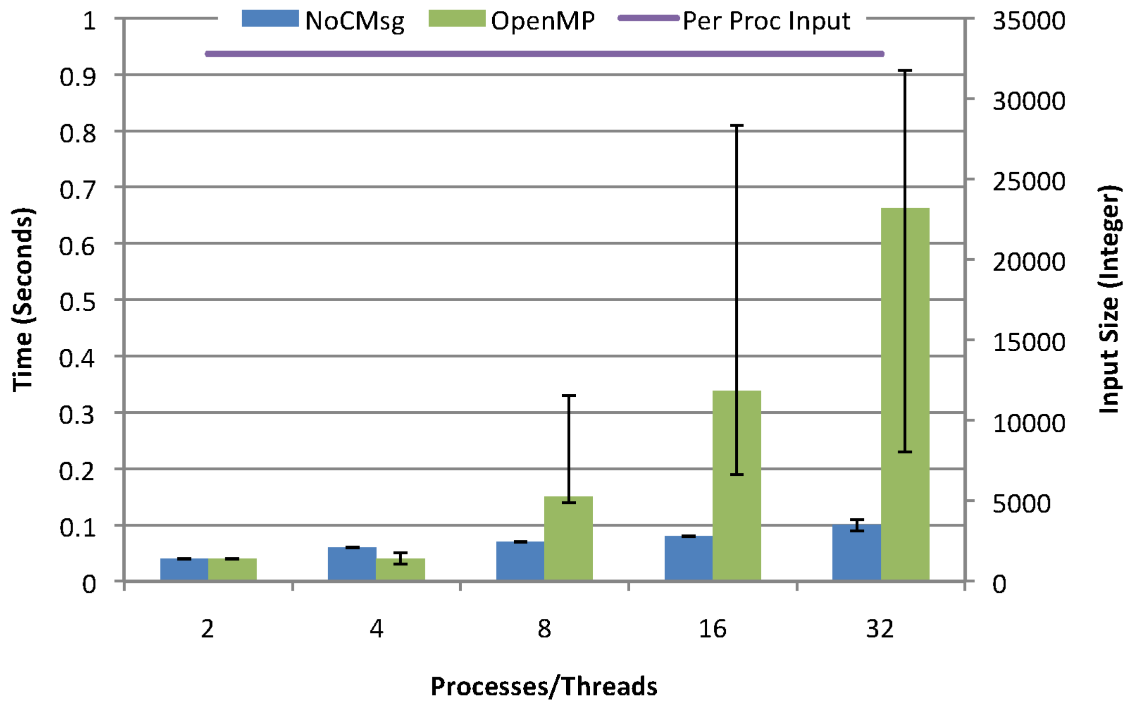


Figure 4.12: NPB IS: Weak Scaling

4.6.4 FPU Experiments

We also conducted experiments to evaluate the benefits of NoCMsg in hardware environments supporting dual-precision floating point units. The Maestro [15] project is a radiation hardened 49-core Tilera board with integrated FPU. Unfortunately the Maestro board does not support software for a Fortran compiler, and the Tilera compiler is limited to software floating point emulation.

Due to these limitations, we found it necessary to perform experiments that would approximate these effects on the Tilepro64 from our original results. We evaluated this by using dual-loop timing for native and soft floating point operations on the Maestro board to ascertain the performance disparity between them. We then used cross-product ratios to extrapolate the potential performance of FPU operations for a 700Mhz Tilepro 64. The results are shown in Table 4.1 depict the cycle latencies between soft and native floating point operations on both the Tilepro and the Maestro boards. It is important to note that the variation between soft-float cycles when comparing Maestro to the Tilepro is due to compiler version differences generating slightly improved code on the Tilepro. We were unable to eliminate these differences due to the age of the only available version of the compiler for the Maestro and a variance in executable format prohibiting us from evaluating code on the opera board using the newer compiler.

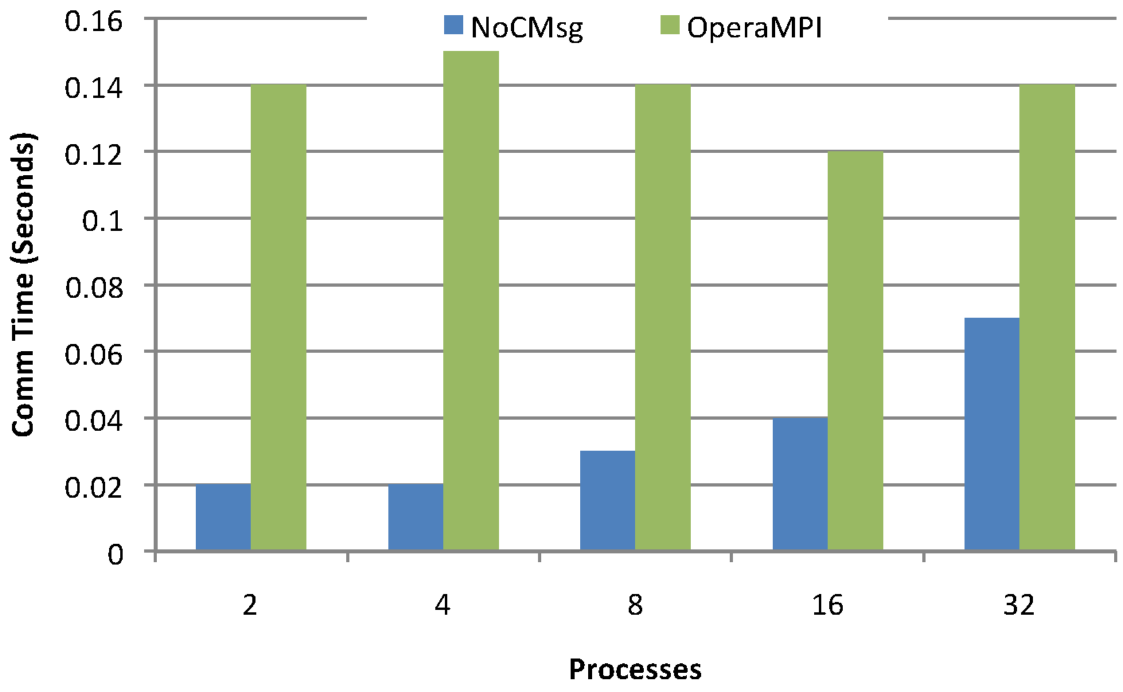


Figure 4.13: NPB FT: NoCMsg vs. Opera MPI

We next extracted performance counter metrics from the floating-point NAS benchmarks using the likwid performance tool that extracts counters on the x86 architecture. We used simple aggregate counters to determine the specific number of floating point adds/multiplies used within each benchmark. We assert that these values should be mostly architecture agnostic for computational kernels as each floating point operation should be derived from a single, high-level floating point calculation performed in the source code. Places where this may potentially deviate would be in code reordering or if variations in the compilers were able to eliminate unused blocks. We do not believe NAS contains many opportunities for doing this and were careful to guarantee the same code path between both the x86 and Tilera evaluations of the code. Using these metrics we where then able to assess the total cost difference for the computation kernels of the code and determine the approximate performance difference.

Figure 4.18 and Figure 4.19 show the impact that might be seen on results containing significant computational overhead. Both SP and LU results were dominated by computational overhead under FPU emulation. In these results, computational overhead was significantly reduced by approximating the affects of using a hardware FPU. This improves the results for NoCMSg and creates a larger performance gap between OpenMP and NoCMsg.

CG Figure 4.20 and FT in Figure 4.21 show similar scalability trends. In these results, the performance benefits increase to similarly high levels as the integer-based IS benchmark shown

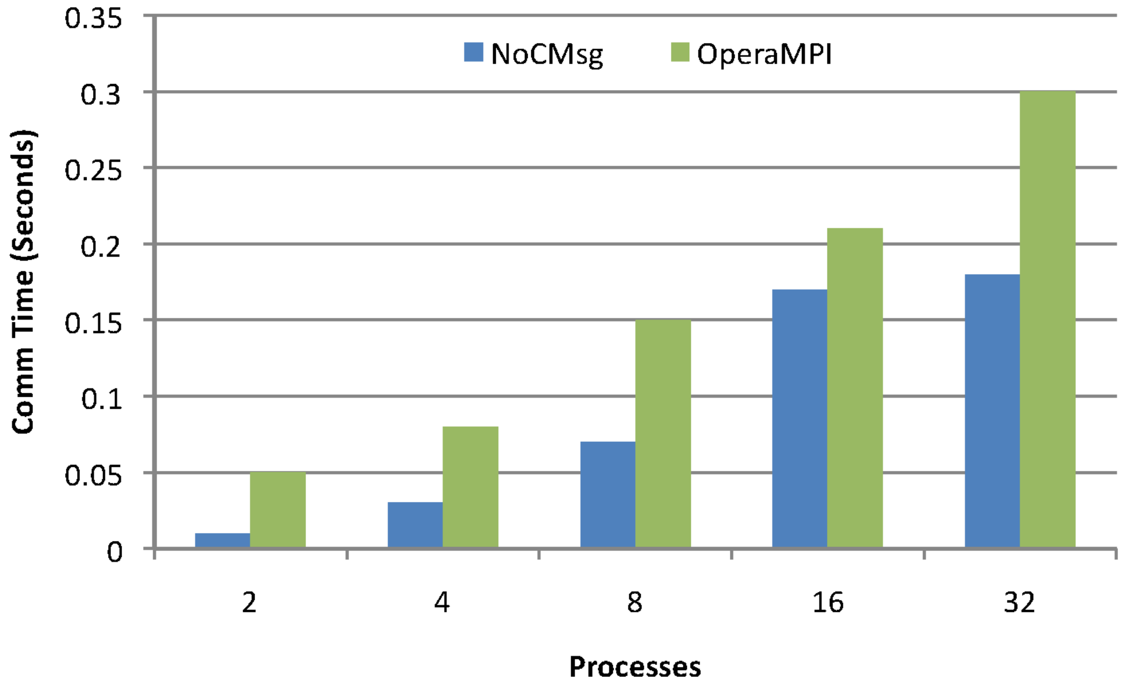


Figure 4.14: NPB CG: NoCMsg vs. Opera MPI

in Figure 4.12. In these results, CG runs 64% faster than its OpenMP comparison when both are using approximated hardware FPUs at 32 cores.

MG in Figure 4.22 again shows interesting scaling performance. Due to the communication pattern changes that occur as the problem size of MG changes, MG shows better performance than NoCMsg at small core counts. This offsets the weak scaling used but the trend is clear and promising: as core counts increase, NoCMsg will outperform OpenMP. Using hardware floating-point operations serves to guarantee this as NoCMsg outperforms OpenMP at 32 cores in this experiment, albeit at small margin.

Table 4.1: Floating Point Performance Metrics

Operation	Native	Soft
Add Opera	20	105
Mult Opera	21	112
Add Tilepro	14	74
Mult Tilepro	17	91

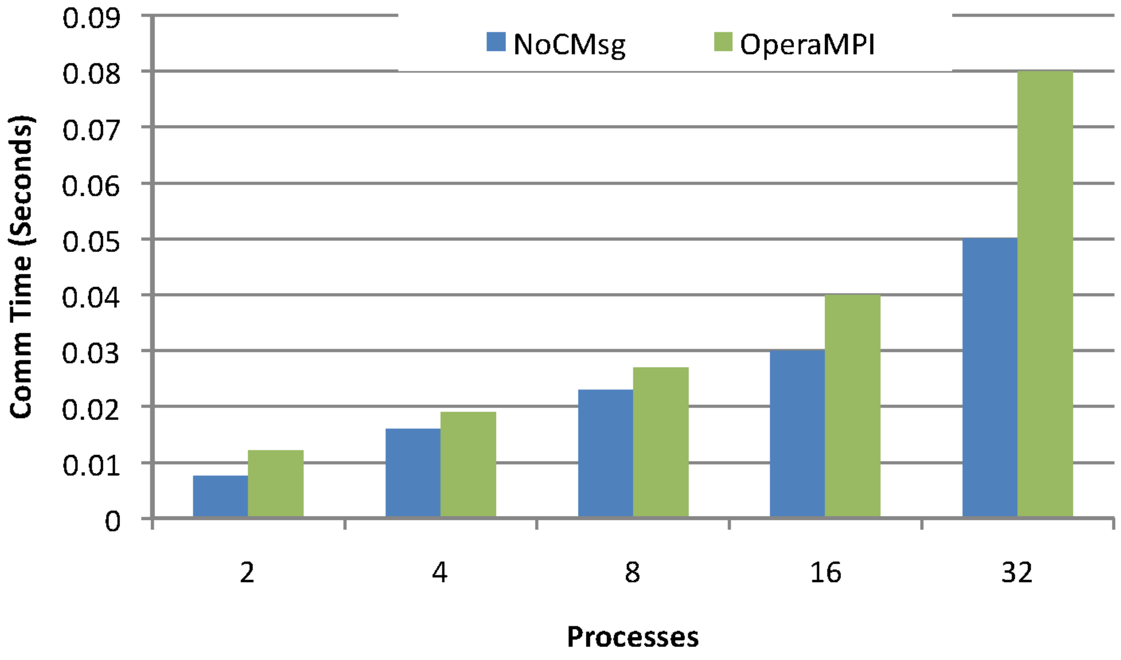


Figure 4.15: NPB IS: NoCMsg vs. Opera MPI

The most important take-away from these results is that as computational overhead decreases NoCMsg sees larger performance gains over OpenMP. This is consistent with our original experiments from the previous section where the integer workloads of IS showed the most significant performance gains. This was largely due to IS being heavily influenced by IPC over computational overhead and the fact that IS was an integer-only benchmark meaning it contains no software emulation. It is also important to remember that the results presented in this section are an approximation and do not account for several important changes that would occur if the computational overhead was reduced. This includes increased contention on IPC pathways and controllers due fewer stall cycles.

4.7 Related Work

Singh et al. [64] and Suh et al. [67] report the performance of FFTW and FFT/CRBlaster, respectively, on the Tilera Maestro platform.

Serres et al. [61] report on the performance of UPC implemented over GasNet plus Pthreads/OperaMPI on a TilePro64. UPC versions of NPB 2.2 under class A show better performance for Pthreads than MPI for benchmarks with significant communication components under strong scaling experiments (input class A).

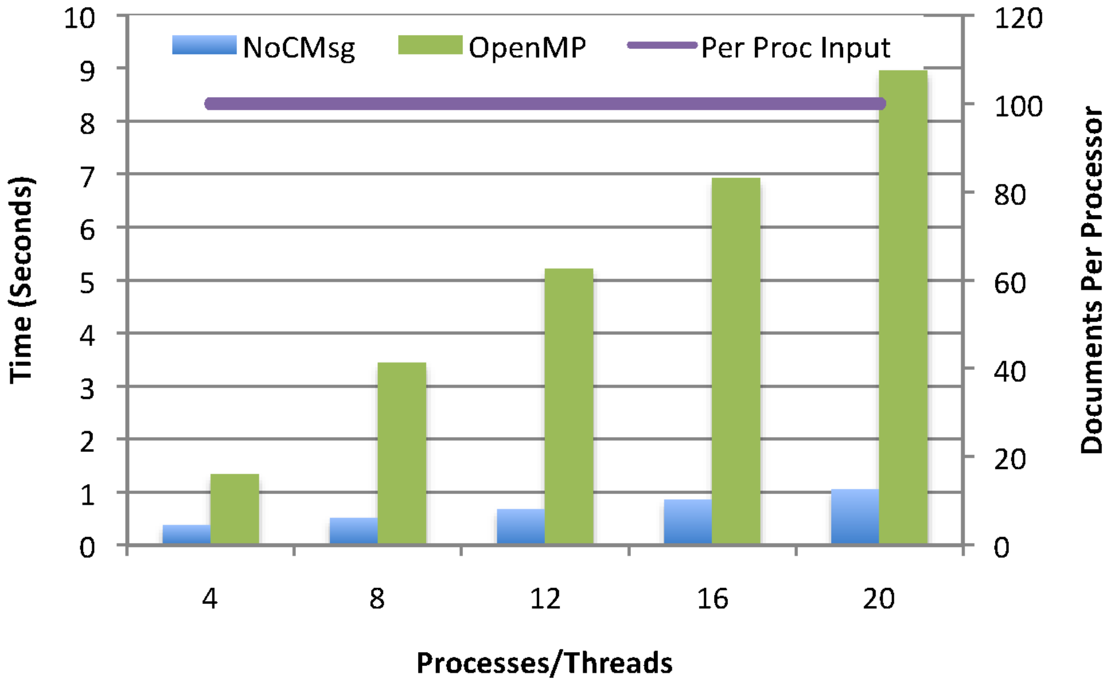


Figure 4.16: TF*IDF: OpenMP vs NoCMsg

Martin et al. [43] report on technique for integrating coherence state and semantics into shared caches to increase scalability. However, the authors themselves acknowledge that these techniques will not improve scalability for all algorithms and that techniques such as message passing are here to stay. Additionally, this paper focuses solely on coherence with little mention of additional performance degradation due to NUMA architectures integrated within many-cores.

This may seem superficially similar to the iWarp[2] protocol that works at an OS level to reduce the overhead of TCP. NoCMsg eliminates the necessity for software flow-control when safe, just as iWarp. The major difference is that NoCMsg operates directly at the hardware level without OS intervention and that NoCMsg is a library, not a protocol.

We compare shared memory against message passing and, in contrast to this past work, assess the effect of enabling coherence for the former while disabling it for the latter. Furthermore, we conduct weak scaling experiments, which reveal the potential and limitations of multicore architectures in terms of parallelization speedup in scenarios where on-chip caches are fully utilized. Finally, we determine the benefits of message passing at the lowest possible level in software instead of multi-layer protocols.

Prior work has compared MPI and OpenMP for shared-memory multiprocessors [37] but not for on-chip NoCs of multicores, which is our focus.

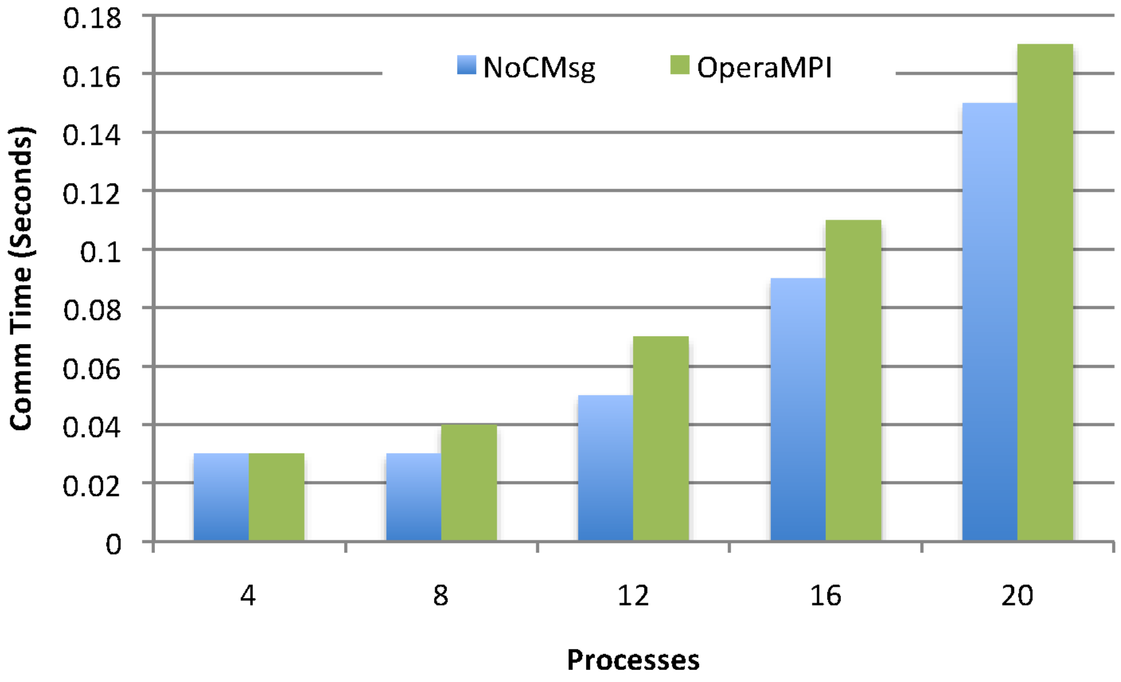


Figure 4.17: TF*IDF: NoCMsg vs OperaMPI

NoCMsg follows the paradigm of addressing scalability problems via message passing, not just for shared-memory multiprocessors as the Multikernel [10] but for multicores in our case. It takes ideas such as NoC-level message passing from Factored Operating Systems [73] to another level in supporting low-level NoCMsg as a basis for scalable NoC communication without deadlocks.

4.8 Conclusion

This chapter presents NoCMsg, a specialized MPI library designed to take advantage of network-on-chip architectures to improve scalability and performance, over a base MPI implementation up to 86% and, more significantly shared memory abstractions such as OpenMP up to 85%. NoCMsg improves scalability by providing a polling-based message passing implementation. Our results indicate that as processor counts and problem sizes increase, even on-chip solutions that employ shared memory are not as scalable as their message passing counterpart. We further develop methods for synchronization and flow control that guarantee deadlock free communication, both of which are essential to communication performance. We demonstrate that communication analysis and pattern-based code replacement around collectives and other code regions of benchmarks allows eliminate flow control in a safe but conservative manner.

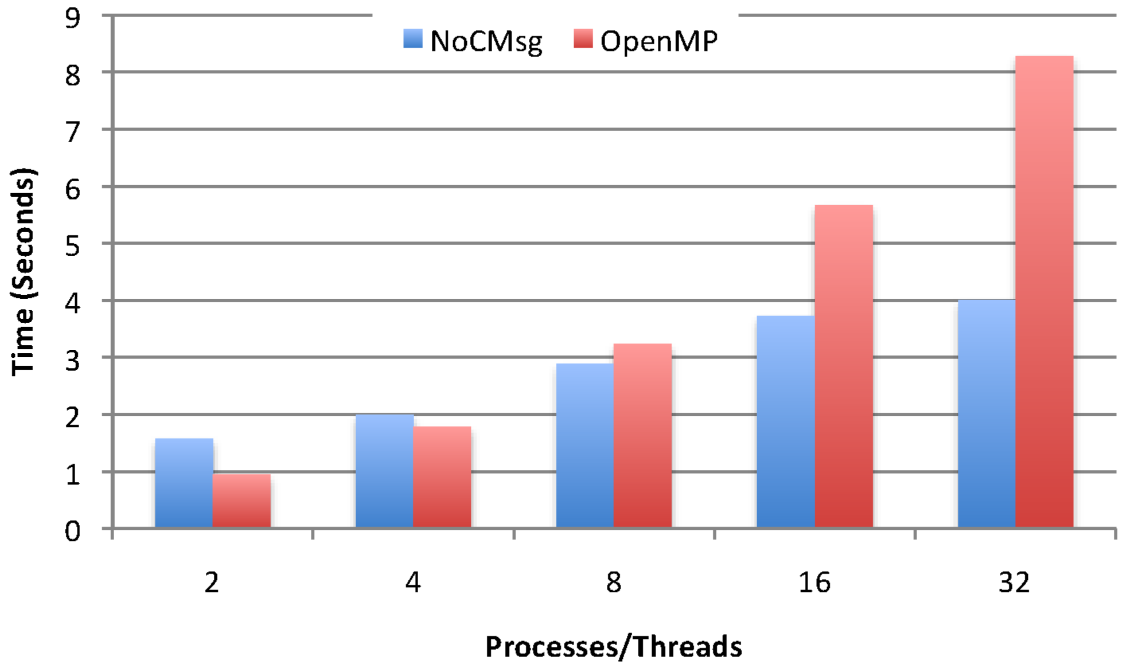


Figure 4.18: LU Native Weak Scaling

These contributions provide significant benefits in performance in terms of wall-clock time, particularly with respect to communication overheads.

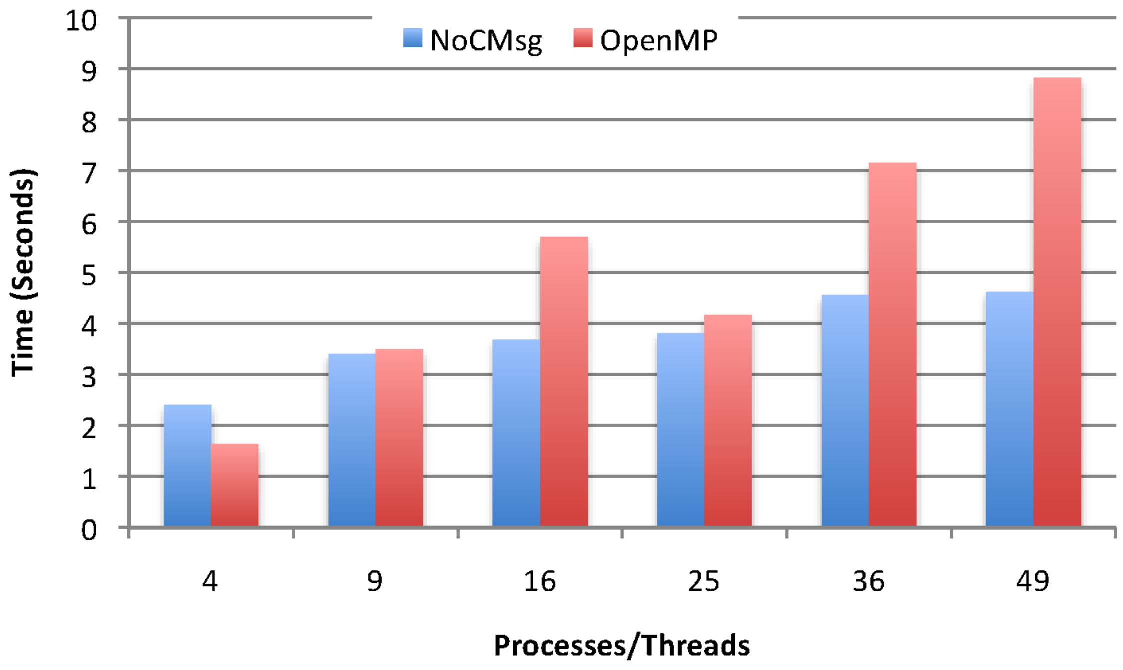


Figure 4.19: SP Native Weak Scaling

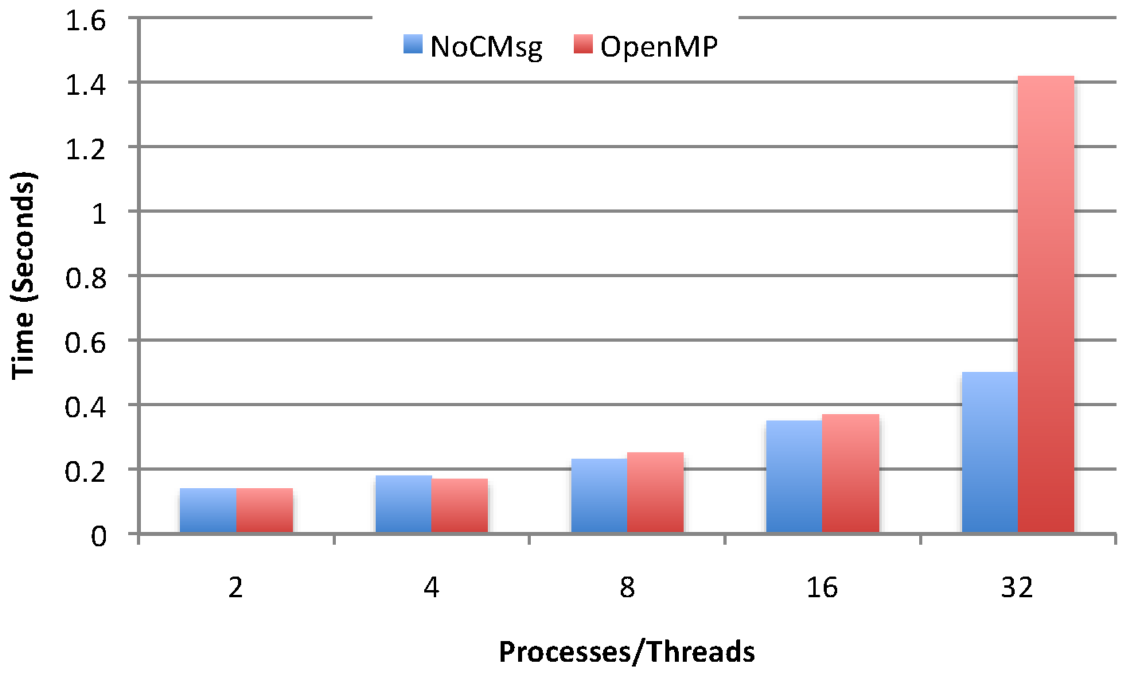


Figure 4.20: CG Native Weak Scaling

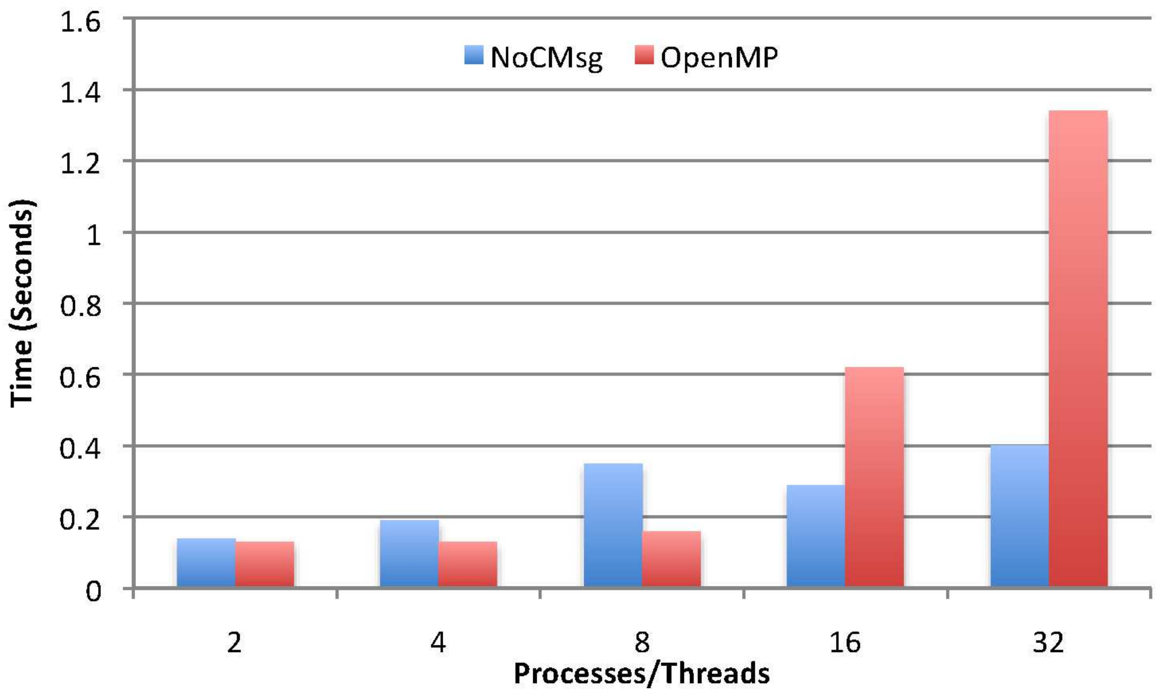


Figure 4.21: FT Native Weak Scaling

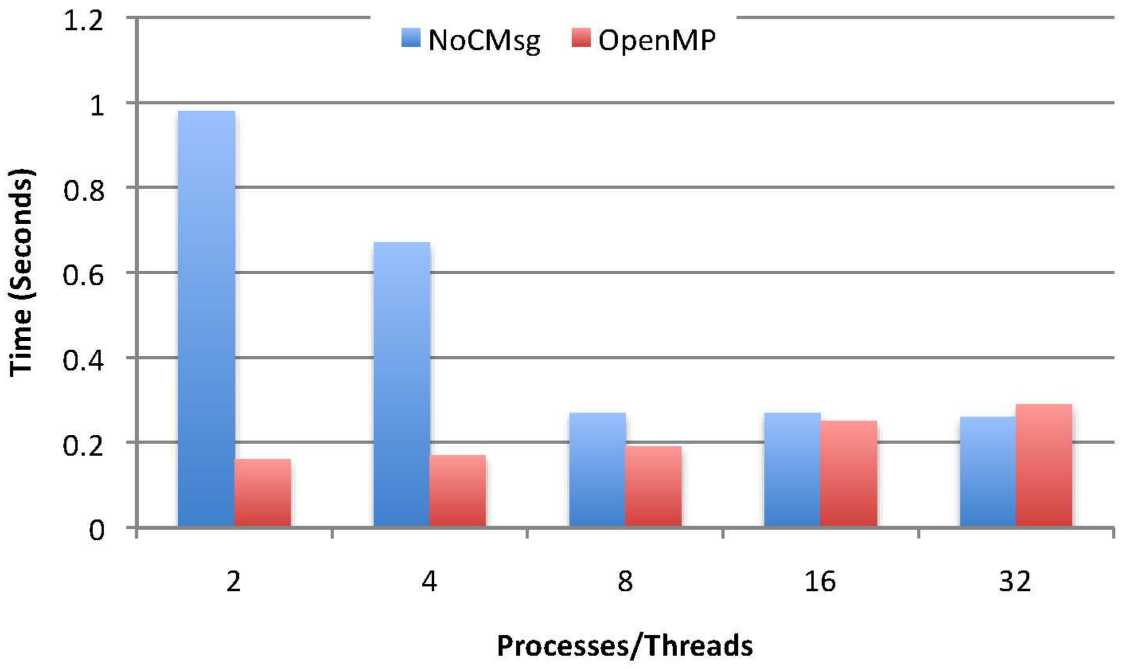


Figure 4.22: MG Native Weak Scaling

Chapter 5

Future Work

In this chapter, we present ongoing and future research directions.

5.0.1 Operating System Abstractions

In Chapter 4, we took our first steps towards new operating system abstractions to support the notion of a distributed system on chip. NoCMsg provided a hardware abstracting pico-kernel that we plan to expand into a hierarchical distributed operating system. We assert that the future of operating systems (OS) is rapidly changing as multicore processors are becoming ubiquitous. It is clear that many-core architectures offer tremendous processing power to meet tomorrow's computational demands. However, there are significant challenges in harnessing this power. As we have shown in the some of our work, shared memory is a limiting technology. Message passing may remedy these limitations for certain applications.

In future work, we plan on evaluating this strategy at the operating system level. Current generation chips contain up to 64 cores on a single processor, with up to 100 cores expected soon. Architecture designers are designing processors containing thousands of cores in the near future. This is in contrast to current operating systems for SMP systems using single system image designs. We expect pitfalls in resource management to become prevalent as core counts increase. In particular, we see challenges in single system image operating systems maintaining efficient scheduling for thousands of separate hardware contexts. This is exacerbated as task layouts over the NoC impact performance. Another area of concern is shared data amongst kernel threads. Current implementations focus largely on shared data objects amongst the kernel objects. As the number of hardware resources increases, this could potentially result in a state explosion that could cause shared objects to incur severe performance penalties. Considering this the impact of operating system design on system performance could prove very important for next generation many-cores.

Our proposed solution to these challenges is an entirely distributed hierarchical, operating

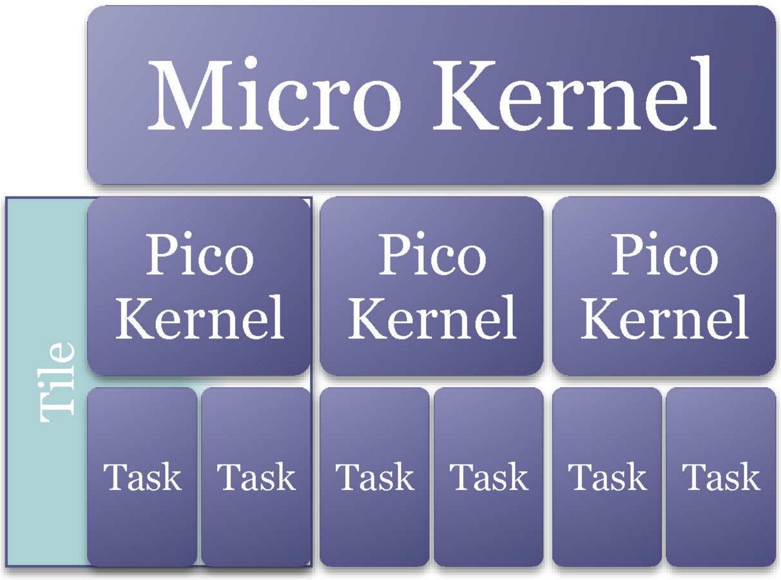


Figure 5.1: Hierarchical Distributed Operating System Abstraction

system (see Figure 5.1) designed from the ground up for network-on-chip many-core architectures. This operating system will do away with traditional mechanisms for IPC at the OS and application level and instead utilize only message passing for communication. A hierarchical OS is broken into two components a pico-kernel and micro-kernel, as shown in Figure 5.1. As mentioned our previous work, NoCMsg falls into the first level of the OS hierarchy as it is a pico-kernel designed to abstract from the hardware and take operational commands. The main idea of future work is in the design and implementation of the micro-kernel. The micro-kernel will be the primary resource management entity in the defined OS hierarchy. We plan on using the micro-kernel abstraction to define partitions into the many-core processor to reduce per OS resource management for both cores and memory controllers. This will impose several challenges from the core scheduling perspective because of the necessity of running highly parallel processes. Because of this, the micro-kernel will be forced to employ distributed scheduling algorithms that are yet to be designed and deployed to work predictably, efficiently, and scalably.

5.0.2 Trade-Off Assessment of Hybrid Programming Models

Results from Chapter 4 show improved scalability for message passing models at or above 16 concurrent cores for most of the NPB suite. However, it is important to realize that these processors are often optimized for memory-based inter-processor communication. This is clear in the results for many of the benchmarks where OpenMP outperforms message passing at smaller core counts. These optimizations come in the form of distributed L3, address hashing distribution, and a wider bus doubling bandwidth for memory transactions.

To this effect, it is important to determine techniques for assessing the trade-offs of mixed OpenMP/MPI implementations. This technique has been utilized in the high-performance computing domain for several years and has shown great potential for performance optimization and scalability.

To assess this technique using our current tools, we would leverage NoCMsg and existing OpenMP compilers using existing benchmarks containing mixed IPC codes. The challenge in this approach would be in determining the partitioning scope and identifying techniques to limit contention effects from shared memory use. This is doubly challenging because previous results on scaling pairs of data exchanging tasks have shown adverse performance effect even when IPC is limited between individual tasks. We believe the cause of this is due to the coarse-grained distributed L3 and false sharing overhead. This may be remedied by determining techniques for localizing multiple L3 caches and utilizing cache ownership strategies to reduce overhead.

5.0.3 NoC Partitioned Scheduling and Load Balancing

With today's many-core processors containing up to 100 cores and tomorrow's slated contain thousands, it is important to design operating system scheduling techniques that will be able to accommodate this scale of resource coordination. Making this challenge even more difficult is the necessity of task scheduling to be power aware. To this end, we aim to investigate strategies for process and application scheduling at large scale. Using the hierarchical OS structure, we plan on investigating two coupled strategies for scheduling.

The first is coordinated pico-kernel scheduling for highly scalable tasks. Considering that this scheduling would be occurring within distributed micro-kernels, this requires modeling the coordination and communication necessary for the micro-kernels to determine what to run when and where. This is further exacerbated by the challenge of coordinated decision making that is necessary for distributed systems to converge.

The second is power-aware micro-kernel scheduling. As mentioned previously, a major goal in a distributed scalable system is the optimization of power. With this in mind, it will be necessary for micro-kernels to alternate power states depending on the requirements of the process load of the system. This task will require deep investigation into

1. strategies for micro-kernels to boot quickly to allow for the deployment of time-sensitive processes;
2. process partitioning and scheduling analysis to reduce the frequency of power-state changes;
3. power-sensitive testing environments to evaluate the improvements that these modifications are capable of at scale; and

4. metrics and benchmarks for evaluating this approach.

To assess these techniques, small-scale evaluations can be performed using existing hardware, but further analysis will require theoretical frameworks or scalable simulation environments.

5.0.4 Run-time Contention Avoidance

Our previous work investigating real-time scalability through message passing showed that message-based contention would influence run-time without intervention. We want to eliminate this perturbation. Such perturbation also affect performance based applications due to load imbalance at collectives (e.g. barriers). Our research in this area showed that by intelligently mapping tasks to cores based on communication analysis we were able to significantly reduce the amount of link based contention and perturbation in many instances. The approach used heuristics and greedy algorithms in a best effort to reduce contended links based on static communication.

We would like to expand these approaches to dynamic run-time systems and relax some of the assumptions made in our previous work. Our goal is to apply this approach to scalable task sets such as those seen in the NoCMsg work where the scope of the task may change during run-time to meet a computational goal. We intend to change our model from static temporal framing for communication and instead exploit pattern-based communication, which can often be statically extracted from these types of applications. In [58], Riesen et al. extracted communication patterns from NAS parallel benchmarks. These benchmarks show communication partner scaling that can be exploited from a scheduling perspective to eliminate contention.

Our work could integrate these techniques into the micro-kernel design and be run dynamically, shortly before dispatching a task to evaluate task-to-core mappings that have the potential to improve performance in critical application sections via reduced packet contention. To evaluate this work, we plan to demonstrate a proof-of-concept implementation by integrating task mapping into NoCMsg and evaluating run-time performance of naive vs. intelligently scheduled benchmarks.

5.0.5 Memory Controller Aware Provisioning

We seek to address a recent trend where the number of memory controllers is increasing for multicore chips. Currently, the Tilera Tilepro 64 contains 4 separate memory controllers that are managed via OS-based NUMA techniques. Unfortunately, at large-scale, as shown in the HPC domain NUMA techniques can result in an order of magnitude differences in performance for memory accesses. In [86], we seek to address this problem via intelligent partitioning and mapping of memory within the OS. Using these techniques we seek to address several challenges:

First, we seek to increase locality by enabling the operating system to manage a memory controller that is physically located close to the running pico-kernels. This may result in improvements in performance due to decreased distances, *i*.e., lower memory latencies. Additionally this can eliminate the necessity for look-up dictionaries that determine which memory controller contains a particular address.

Second, we can internally apply this technique to employ micro-kernel paging separation within the OS to eliminate latency in page table walks due to maintaining a single set of page tables for each micro-kernel. We would again expect to see performance improvements based on improved locality of OS structures compared to those of a single-system image OS using NUMA.

Chapter 6

Conclusion

In this chapter, we summarize our efforts and present conclusions drawn from this research.

Chapter 2 introduces shared memory as a limiting factor to scalability for many-core NoC architectures. Using a micro-benchmark exercising inter-processor communication via message passing and shared memory, we are able to show that scaling independent pairs of tasks still leads to memory contention. This memory contention is the result of the use of distributed L3 caching that uses portions of each cores L2 to improve memory bandwidth. Unfortunately, this can lead to non-uniform cache distances for accesses and results in resource contention on cache controllers. Our micro-benchmark results show increasing perturbation under scaling. This perturbation is a limiting factor for many improvements that NoC architectures may be able to achieve, including task duplication-based resilience. Fortunately, our results also show that if we replace shared memory with message passing, perturbation can be reduced.

Using message passing, we propose a new programming paradigm that is ideally suited to allow NoC architectures to operate with long-term resilience in hazardous environments. By changing the traditional real-time feedback loop into a series of chained distributed/message passing tasks, we are able to effectively scale the amount of concurrently running tasks without affecting predictability as is the case under shared memory. In doing so, we introduce the fault observant real time embedded (Forte) framework for running concurrent real-time systems. The Forte programming paradigm enables fault tolerance through modular redundancy. It eliminates concerns that concurrent models will suffer from losses of predictability due to strain on memory controllers because the model only uses message passing for IPC.

Forte duplication takes advantage of the well known simplex design. By reducing task and data complexity, real-time tasks are able to reduce their code footprint and improve the probability of fail-safe operation. Forte accommodates multi-complexity task redundancy by checking for coherence between data instead of strict equivalence. We show that the Forte framework using multi-mode execution is also able to support the notion of task rejuvenation and correct data failures when they occur.

In Forte, we experimentally evaluated well-known UAV codes using this programming paradigm on the Tilera architecture. We showed that by using message passing instead of shared memory, we can reduce latencies by up to an order of magnitude. We further showed that by enabling rejuvenation with data correction, we are able to increase the mean time to failure (MTTF) when using systems with triple modular redundancy. MTTF increases from 157 days to $2.05x10^9$ days under single event upsets and a 250 ms repair period.

Chapter 3 builds upon a message-passing model loosely correlated to Chapter 2. It investigates the impact of task placement on network contention and its impact on predictability. Using small scale message passing experiments for NoC-based real-time messaging, we identify that worm-hole based packet routing can result in delivery delays when multiple messages contend for the same resources. The outcome is perturbation that could potentially affect the worst case execution times of tasks deployed in such systems. We further break this contention into two types, link based, and core based contention. We identify that core based contention can only be dealt with through message scheduling and instead choose to seek techniques for remedying link-based contention.

To alleviate link-based contention, we propose the temporal frame graph abstraction model. Similar to TDMA, temporal frame graphs statically limit when messages can be sent but differs in that it does not limit the hardware resources to only be used by one task at a time. Instead, it uses the graphs as an analysis platform to perform constraint-based optimal scheduling on the TFG and determines the layouts of tasks that result in the least amount of contention. In substantiating this work, we demonstrate on actual hardware the impact that link contention can have on predictability. We identify the limits of exhaustive solutions using our model and use analysis of these solutions to identify heuristics. To extend this work, we introduce a multi-heuristic contention solver known as HSolver that makes it possible to derive low contention solutions for NoC processor meshes of size 5x5 and greater.

This work is evaluated on a Tilera Tilepro 64 network-on-chip processor. We developed a framework for real-time task deployment for evaluation of our technique. The framework is capable of performing the task-to-core binding and evaluation of our synthetic task-sets using real messages on the Tilera hardware. Using cycle-accurate hardware counters, we are able to attain fine grain timing results to demonstrate the reduction in perturbation from using this technique. Using HSolver for 8x8 meshes, we were able to reduce the number of contended links by up to 60% when compared against a naïve solution and a timed exhaustive solver.

Chapter 4 demonstrates the versatility of NoC message passing architectures. Following the hypothesis of our work, we identify limitations within shared memory as a major stop-gap in the goal of extracting performance through parallelism. In this work, we investigate the use of message passing for performance critical applications exhibiting significant parallelism. We use previous work to support this argument and focus our design on identifying mechanisms for extracting parallelism exploiting message passing that can overcome the limitations of the underlying hardware.

We motivate this work with a discussion of common design patterns in simulated and existing NoC processors. We identify through put optimized network hardware as the primary hurdle. Switches in such networks are optimized for the transmission of packets and very simple in design. This puts the responsibility for flow control on to the software. Not accounting for flow control could lead to network-based deadlock, largely caused by cycles in IPC chains. Similar to resource-based deadlock in shared memory, processes cannot make further computational progress (deadlock).

To address these challenges, we design a new message passing kernel that abstracts hardware features from the programmer, thereby easing the development of message passing codes. Through analysis of common hardware patterns, we seek to transparently eliminate flow control concerns from designers. We do so by designing a method of evaluating network lock conditions from sending cores. Using a light-weight check prior to flit transfers, cores are able to avoid unnecessary pipeline blocking and instead invoke receiving or computational routines. With flow control concerns eliminated for base transfers, we are able to supply common routines that exist within industry standard MPI, such as asynchronous communication and collective operations.

We extend this work further by identifying that the MPI semantics offer a great deal of opportunity to relax flow control constraints. We prototype two techniques of relaxation. We first use profile-based path extraction from the profiling layer of our implementation. Using path and pattern analysis, we then determine if point-to-point messages can be replaced using versions without flow control checks. This is particularly effective in reducing the latency costs for large messages where aggregate flow control checks can get expensive. We expound upon this by engaging in similar static analysis for collectives and identify common pre-conditions in which collective operations can be applied without the use of flow control. Collective operations are generally expensive and decreasing their cost can potentially show significant performance improvement.

To evaluate this work, we use the NAS parallel benchmark suite. We use kernels from LU, SP, CG, FT, MG, and IS to evaluate our implementations. We compare against OpenMP and an existing MPI implementation for the Tilera many-core processor. Our comparisons against OpenMP show significantly improved performance under weak scaling for NoCMsg against the 6 compared kernels. This is particularly true at large core counts of 16 or more processors where contention effects inhibit OpenMP performance. Note that all of these benchmarks, except IS, are floating point codes and are not natively supported by the TilePro architecture. This leads to a greater computational cost and greater separation of inter-processor communication.

Results from IS, without software emulation, show an 85% reduction in execution time at 32 processors. When compared with the alternative message passing library, our approach results in similar performance except when compared against cases where flow-control removal can be applied. In these latter cases we see improvements of up to 40% in performance at 32 cores.

In the work presented in Chapter 2, Chapter 3, and Chapter 4, we provide a model of software design that most closely resembles on-chip distributed programming. We show that that the hypothesis holds, i.e., message passing approaches are a feasible solution to reducing reliance on shared memory abstractions and result in increased program reliability, predictability, scalability, and performance.

REFERENCES

[1] Adapteva processor family. www.adapteva.com/products/silicon-devices/e16g301/.

[2] A remote direct memory access protocol specification. tools.ietf.org/html/rfc5040.

[3] Single-chip cloud computer. blogs.intel.com/research/2009/12/sccloudcomp.php.

[4] Tera-scale research prototype: Connecting 80 simple cores on a single test chip. ftp://download.intel.com/research/platform/terascale/terascaleresearchprototypebackgrounder.pdf.

[5] Tilera gx processor family. http://www.tilera.com/products/processors/TILE-Gx_Family.

[6] Tilera processor family. http://www.tilera.com/products/processors.php.

[7] Tilera user architecture reference. www.tilera.com.

[8] Hisashige Ando, Yuuji Yoshida, Aiichiro Inoue, Itsumi Sugiyama, Takeo Asakawa, Kuniki Morita, Toshiyuki Muta, Tsuyoshi Motokurumada, Seishi Okada, Hideo Yamashita, Yoshihiko Satsukawa, Akihiko Konmoto, Ryouichi Yamashita, and Hiroyuki Sugiyama. A 1.3ghz fifth generation sparc64 microprocessor. In *Design Automation Conference*, pages 702–705, New York, NY, USA, 2003. ACM Press.

[9] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[10] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Symposium on Operating Systems Principles, pages 29–44, 2009.

[11] A. Bender. Milp based task mapping for heterogeneous multiprocessor systems. In *Proceedings of the conference on European design automation*, EURO-DAC '96/EURO-VHDL '96, pages 190–197, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[12] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symposium*, December 2002.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[14] V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. *International Software Quality Week*, May 1997.

[15] Manuel Cabanas-Holmen, Ethan H. Cannon, Charles Neathery, Roger Brees, Bryan Buchanan, Anthony Amort, and AJ Kleinosowski. Maestro processor single event error analysis.

[16] Chen-Ling Chou and R. Marculescu. Contention-aware application mapping for network-on-chip communication architectures. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 164 –169, oct. 2008.

[17] M. Cirinei, E. Bini, G. Lipari, and A. Ferrari. A flexible scheme for scheduling fault-tolerant real-time tasks on multiprocessors. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8, March 2007.

[18] C. Constantinescu. Trends and challenges in vlsi circuits reliability. *IEEE Micro*, pages 14–19, July-August, 1996.

[19] J. T. Daly. Running applications successfully at extreme scale: What is needed? ANL-TR LA-UR-09-1800, ASCR Computer Science Research, Principal Investigators Meeting, 2008.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

[21] John Donovan. Arm cto warns of dark silicon.

[22] Dong-Rui Fan, Nan Yuan, Jun-Chao Zhang, Yong-Bin Zhou, Wei Lin, Feng-Long Song, Xiao-Chun Ye, He Huang, Lei Yu, Guo-Ping Long, Hao Zhang, and Lei Liu. Godson-t: An efficient many-core architecture for parallel program executions. *Journal of Computer Science and Technology*, 24:1061–1073, 2009. 10.1007/s11390-009-9295-3.

[23] M.J. Flynn and P. Hung. Microprocessor design issues: thoughts on the road ahead. *Micro, IEEE*, 25(3):16 – 31, may-june 2005.

[24] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(3):272 –284, March 1997.

[25] Kees Goossens, John Dielissen, and Andrei Radulescu. &#198;thereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22:414–421, September 2005.

[26] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[27] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[28] Kyle C. Hale, Boris Grot, and Stephen W. Keckler. Segment gating for static energy reduction in networks-on-chip. In *Workshop on Network on Chip Architectures*, pages 57–62, 2009.

[29] Ching-Chih Han, K.G. Shin, and Jian Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *Computers, IEEE Transactions on*, 52(3):362 – 372, March 2003.

[30] Jingcao Hu and Radu Marculescu. Energy- and performance-aware mapping for regular noc architectures. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 24(4):551–562, 2005.

[31] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton. Software rejuvenation: analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381 –390, June 1995.

[32] V. Izosimov, I. Polian, P. Pop, P. Eles, and Zebo Peng. Analysis and optimization of fault-tolerant embedded systems with hardened processors. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 682 –687, April 2009.

[33] Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin Schoeberl. Exhaustive testing of safety critical java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, New York, NY, USA, 2010. ACM.

[34] M. Kang, E. Park, M. Cho, J. Suh, D.-I. Kang, and S. P. Crago. Mpi performance analysis and optimization on tile64/maestro. In *Workshop on Multi-core Processors for Space — Opportunities and Challenges*, July 2009.

[35] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, , and R. Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *IEEE Real-Time Systems Symposium*, pages 57–66, 2011.

[36] Shinpei Kato, Karthik Lakshmanan, Yutaka Ishikawa, and Ragunathan (Raj) Rajkumar. Resource sharing in gpu-accelerated windowing systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 191–200, 2011.

[37] Géraud Krawezik and Franck Cappello. Performance comparison of mpi and openmp on shared memory multiprocessors: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(1):29–61, January 2006.

[38] Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8:355–383, July 2003.

[39] Ser-Hoon Lee, Yeo-Chan Yoon, and Sun-Young Hwang. Communication-aware task assignment algorithm for mpsoc using shared memory. *Journal of Systems Architecture*, 56(7):233 – 241, 2010. ¡ce:title¿Special Issue on HW/SW Co-Design: Systems and Networks on Chip¡/ce:title¿.

[40] Witold Lipski, Jr. An o(n log n) manhattan path algorithm. *Inf. Process. Lett.*, 19:99–102, September 1984.

[41] J. Liu. *Real-Time Systems.* Prentice Hall, 2000.

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

[43] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[44] Sibin Mohan and Frank Mueller. Preserving timing anomalies in pipelines of high-end processors. Technical Report TR 2007-13, Dept. of Computer Science, North Carolina State University, 2008.

[45] Sibin Mohan, Frank Mueller, William Hawkins, Michael Root, Christopher Healy, and David Whalley. Parascale: Expoliting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *IEEE Real-Time Systems Symposium*, pages 233–242, December 2005.

[46] Matteo Monchiero, Gianluca Palermo, Cristina Silvano, and Oreste Villa. Exploration of distributed shared memory architectures for noc-based multiprocessors. *Journal of Systems Architecture*, 53(10):719 – 732, 2007. Embedded Computer Systems: Architectures, Modeling, and Simulation.

[47] Andreas Moshovos, Gokhan Memik, Alok Choudhary, and Babak Falsafi. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *International Symposium on High Performance Computer Architecture*, pages 85–96, 2001.

[48] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto noc architectures, 2004.

[49] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany.

[50] R. Obermaisser, H. Kraut, and C. Salloum. A transient-resilient system-on-a-chip architecture with support for on-chip and off-chip tmr. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 123 –134, May 2008.

[51] N. Oh, P. Shirvani, and E. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[52] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[53] M. Pignol. Cots-based applications in space avionics. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1213 –1219, march 2010.

[54] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 148–157, March 2005.

[55] Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante, and Marco Torchiano. A source-to-source compiler for generating dependable software. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 35–44, 2001.

[56] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[57] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *International Symposium on Computer Architecture*, pages 148–159, 2005.

[58] R. Riesen. Communication patterns [message-passing patterns]. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006.

[59] J. G. Rivera, A. A. Danylyszyn, C. B. Weinstock, L.R. Sha, and M. J. Gagliardi. An Architectural Description of the Simplex Architecture. Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pennsylvania, 1996.

[60] K. Sankaralingam, R. Nagarajan, P. Gratz, R. Desikan, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, W. Yoder, R. McDonald, S.W. Keckler, and D.C. Burger. The distributed microarchitecture of the trips prototype processor. In *International Symposium on Microarchitecture*, November 2006.

[61] Olivier Serres, Ahmad Anbar, Saumil Merchant, and Tarek El-Ghazawi. Experiences with upc on tile-64 processor. In *2011 IEEE Aerospace Conference*, pages 1–9, 2011.

[62] P. Shirvani, N. Saxena, and E. McCluskey. Software-implemented edac protection against seus. *IEEE Transactions on Reliability*, 49(1):273–284, 2000.

[63] C. Singh. Reliability modeling of tmr computer systems with repair and common mode failures. *Microelectronics and Reliability*, 21(2):259 – 262, 1981.

[64] Karandeep Singh, John Paul Walters, Joel Hestness, Jinwoo Suh, Craig M. Rogers, and Stephen P. Crago. Fftw and complex ambiguity function performance on the maestro processor. In *IEEE Aerospace Conference*, pages 1–8, 2011.

[65] Joel R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, 1976.

[66] S. Stuijk, T. Basten, M.C.W. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 777 –782, june 2007.

[67] Jinwoo Suh, K.J. Mighell, Dong-In Kang, and S.P. Crago. Implementation of fft and crblaster on the maestro processor. In *IEEE Aerospace Conference*, pages 1–6, march 2012.

[68] Ian Troxel, Eric Grobelny, Grzegorz Cieslewski, John Curreri, Mike Fischer, and Alan D. George. Reliable management services for cotsbased space systems and applications. In *Proc. International Conference on Embedded Systems and Applications (ESA), Las Vegas, NV*, 2006.

[69] Rajesh Venkatasubramanian, John P. Hayes, and Brian T. Murray. Low-cost on-line fault detection using control flow assertions. In *International On-Line Testing Symposium*, pages 137–143, 2003.

[70] Villalpando, C.Y., Johnson, A.E., Some, R., J. Oberlin, Goldberg, and S. Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander. In *Aerospace Conference, 2010 IEEE*, pages 1 –9, march 2010.

[71] V.Narayanan and Yuan Xie. Reliability concerns in embedded system designs. *IEEE Computer magazine*, pages 106–108, January, 2006.

[72] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, November 2001.

[73] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009.

[74] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, September 2007.

[75] J. Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction.* PhD thesis, University of York, May 2008.

[76] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, April 2008.

[77] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *International Symposium on Computer Architecture*, pages 24 –36, june 1995.

[78] Jun Yan and Wei Zhang. Compiler-guided register reliability improvement against soft errors. In *International Conference on Embedded Software*, pages 203–209, 2005.

[79] Y. C. (Bob) Yeh. Design considerations in boeing 777 fly-by-wire computers. In *IEEE International High-Assurance Systems Engineering Symposium*, page 64, 1998.

[80] Y.C. Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307, 1996.

[81] W. Yurcik and D. Doss. Achieving fault-tolerant software with rejuvenation and reconfiguration. *Software, IEEE*, 18(4):48 –52, 2001.

[82] Y. Zhang, F. Mueller, Xiaohui Cui, and Thomas Potok. Large-scale multi-dimensional document clustering on gpu clusters. In *International Parallel and Distributed Processing Symposium*, April 2010.

[83] Jun Zhu, I. Sander, and A. Jantsch. Constrained global scheduling of streaming applications on mpsocs. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 223 –228, jan. 2010.

[84] C. Zimmer and F. Mueller. Fault resilient real-time design for noc architectures. In *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pages 75 –84, april 2012.

[85] Christopher Zimmer and Frank Mueller. Low contention mapping of real-time tasks onto tilepro 64 core processors. *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 0:131–140, 2012.

[86] Christopher Zimmer and Frank Mueller. Operating System Abstractions for Large-Scale Multicores. In *The Real-Time Linux Workshop*, October 2012.