ABSTRACT

RAMACHANDRAN, SUBRAMANIAN. Distributed Job Allocation for Large-Scale Many-cores. (Under the direction of Dr. Frank Mueller.)

As today's manycore processors already feature over 64 cores and as tomorrow's are slated to contain 1000s, it is important to design operating system techniques that can efficiently cope with this scale of resource coordination. The current state-of-the-art in manycore processor architectures has evolved from traditional bus-based architectures over rings to mesh-based Network-on-Chip (NoC) interconnects. This implies an increasing potential for scalable message passing. However, contemporary operating systems heavily rely on single system images with shared memory constructs that may not scale well to large core counts. To address these challenges, we devise a distributed message passing only system comprised of so-called "pico-kernels" per core. They are controlled by dedicated "micro-kernels" topologically centered within a set of cores that cooperatively comprise the overall operating system in a peer-to-peer fashion. Such a system promotes rethinking and redesigning of various operating system services focusing on scalability as the primary design constraint. We consider the challenges of distributed allocation of jobs, each comprised of a set of tasks to be mapped to disjoint cores. A naive solution performing fragmented allocations may quickly escalate to deadlocks, where jobs hold and wait for cores in circular dependencies. To tackle these challenges, we propose a deadlock free distributed job allocation protocol. We have devised two policies for avoiding deadlocks, namely *active cancellation* and *sequencer-based atomic broadcast*. The protocol and the two policies have been implemented and evaluated on a Tilera TilePro64 processor with 64 cores on a single socket. Results show that for sparse job allocations *active cancellation* provides less job allocation overhead while for denser job allocations the *sequencer-based atomic broadcast* provides less overhead.

Distributed Job Allocation for Large-Scale Many-cores

by
Subramanian Ramachandran

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

Dr. Vincent Freeh	Dr. Steffen Heber

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

To Amma and Appa.

BIOGRAPHY

The author was born in Madurai, a popular city in Tamilnadu, India. He completed his schooling at Ooty and Madurai. He went on to pursue his Bachelors in Electronics and Communication Engineering at Madras Institute of Technology (MIT India), Anna University. After his undergraduation he worked at IBM India Software Labs as a Senior Software Engineer in their Systems and Technology Group from 2007-2011. Later he moved on to Cisco Systems, India where he worked in their Mobile Internet Technology Group (MITG) from 2011-2012. In Fall 2012, he began his graduate studies at North Carolina State University, Raleigh in the field of Computer Science. He started working as a Research Assistant in the Systems Research Group under Dr. Frank Mueller from Fall 2013. During Summer 2013, Subramanian interned at Riverbed, CA and plans to join there after graduation.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Frank Mueller for trusting in my abilities and providing me an opportunity to work on this thesis. His constant flow of ideas, dedication and never say give up attitude inspired me a lot. His open door policy enabled me to discuss any idea or issue, without any inhibition. Thank you Dr. Mueller once again.

I would like to thank Dr. Vincent Freeh and Dr. Steffen Heber for agreeing to serve in my thesis committee; providing a fresh outlook to my work and guiding me throughout this process.

I would like to thank my parents, sister and little Harini for always being my constant source of encouragement and love. I am really blessed to have such a caring family.

I would like to thank my instructors, teachers and professors right from my childhood till now for whatever I am today is only because of them. I would like to thank my managers and colleagues at IBM and CISCO India who motivated me to pursue this journey.

This work would not have been possible without the help of some awesome people with whom I had a chance to interact during my two year stint as a Master's student here.

Chris Zimmer, thank you for the awesome code base of NoCMsg and helping me in any issue I faced on the Tilera board. Karthik Yagna, thank you for your valuable tips and suggestions.

My friends Madhavan, Shankar, Anerudhan, Mahesh, Sandeep and Kasyap, thank you for voluntarily helping me out in any issue, through whatever possible means. Thanks for providing valuable suggestions and an accommodating environment to work when I needed it the most.

Last but not the least; I would like to thank all my lab mates and friends who encouraged and motivated me to try harder till the end. Thank you everyone once again.

TABLE OF CONTENTS

LIST OF FIGURES . . . . . **vi**

Chapter 1 Introduction **1**
1.1 Motivation 1
1.1.1 An Era of Large-Scale Manycores 1
1.1.2 Scalability Challenges of Large-Scale Manycores 1
1.2 Hypothesis 2
1.3 Contributions 2
1.3.1 PICASO system 3
1.3.2 Distributed Job Allocation Protocol 3
1.4 Organization 4

Chapter 2 Design **5**
2.1 The PICASO System 5
2.2 The Distributed Job Allocation Problem 5
2.3 Deadlock free Distributed Job Allocation Protocols 9
2.3.1 The Main Scheduling Loop 9
2.3.2 Deadlock Avoidance by Active Cancellation 11
2.3.3 Deadlock Avoidance by Sequencer Based Atomic Broadcast 13
2.3.4 Pattern-Based Message Propagation Schemes 13

Chapter 3 Implementation **16**
3.0.5 Global Unique Ordering 16
3.0.6 Extensions to the NoCMsg Library 16

Chapter 4 Evaluation Framework **18**

Chapter 5 Experimental Results **20**
5.1 Performance Analysis 20
5.1.1 Micro-benchmark — Overhead for Sparse Job Allocations 22
5.1.2 Micro-benchmark — Job Allocation Overhead as Tile Size Increases . . . 24
5.1.3 Micro-benchmark — Worst-case Conflict Resolution Time for n Simultaneous Job Submissions 25
5.2 Experiments in Real Task Mode 26
5.2.1 Job Allocations that can execute in parallel 27
5.2.2 Job allocations which can only execute serially 27
5.3 Performance of the Pattern-Based Propagation Schemes 28

Chapter 6 Related work **31**

Chapter 7 Conclusion **33**

References **34**

LIST OF FIGURES

Figure 2.1 An example micro/pico kernel abstraction in a large many core processor . . 6
Figure 2.2 A probable deadlock involving two simultaneous job submissions at different micro-kernels 8
Figure 2.3 Periodic active cancellation procedure 10
Figure 2.4 Steps in processing core allocation request for active cancellation 12
Figure 2.5 Pattern-based request propagation schemes 14

Figure 4.1 PICASO system within a 6×6 tile on Tilera TilePro64 18

Figure 5.1 Micro-benchmark — Overhead for Sparse Job Allocations where each job requires a number of pico-kernels (cores) that is satisfied by: 21
Figure 5.2 Micro-benchmark — job allocations as tile size increases 24
Figure 5.3 Micro-benchmark — Worst-case conflict resolution time for n simultaneous job submissions 26
Figure 5.4 Real task mode — Parallel Job Allocations 27
Figure 5.5 Real task mode — Serial Job Allocations 28
Figure 5.6 Comparison of different request propagation schemes 29

Chapter 1

Introduction

1.1 Motivation

1.1.1 An Era of Large-Scale Manycores

Gordon Moore (co-founder of Intel) made this famous observation:
"The number of transistors incorporated in a chip will approximately double every 24 months."
For several decades, Moore's law [25] has held strong with continued chip performance increases every 18 months, even faster than what was predicted. Yet, uniprocessor scaling has reached its ultimate physical limits with increased power consumption and diminished performance returns. Nonetheless, multicore/manycore processors have the potential to enjoy continued performance increases to meet future processing needs while reducing/constraining power consumption. Hence, many chip vendors have abandoned uniprocessor scaling and have instead resorted to doubling the number of cores per chip. With current single microprocessor chips of up to 100's of cores on a die [1–4] available, a 1000 core chip might soon be reality [8, 30], and specialized computing devices, e.g., graphic processing units (GPUs), already support such scales today.

1.1.2 Scalability Challenges of Large-Scale Manycores

The current state-of-the-art for multicore chips has evolved from traditional bus-based architectures over rings to mesh-based Network-on-Chip (NoC) interconnects between processors. This implies an increasing potential for scalable message passing capabilities. However, to date multicore benefits have not scaled well. The primary reason for this stems from reusing conventional Single System Image (SSI) operating system designs for multicore architectures. With SSI, resources are aggregated to present a single view of the operating system environment while data access and communication are realized via shared memory over traditional bidirectional

buses. This approach delivers some performance increases in the natural evolution from single core up to 16 cores, but it deteriorates rapidly when the number of cores increases further [33]. Despite innovative designs of caching hierarchies and protocols (shared, L1, L2, etc.), the latency and contention for accessing shared memory with multiple cores limits performance gains at scale. Cache misses and coherence updates are also extremely expensive in SSI approaches on multicores since each core has its own cache that must be coherent with shared memory, as well as with other cores. Recent work by Baumann et al. [6], Wentzlaff et al. [29] and Zimmer et al. [33] show that coherent shared memory may not scale well to large core counts. They instead promote the usage of scalable message passing for OS communication in large-scale manycores.

Modern large-scale manycore processors resemble a distributed system [7]. Hence, it is of utmost importance that operating system techniques be revisited and redesigned to embrace the distributed nature of these manycore processors, with scalability as one of the primary design constraint.

1.2 Hypothesis

In this work, we attempt to address the above discussed challenges. We have devised a distributed message passing only system comprised of so-called "pico-kernels" per core. They are controlled by dedicated "micro-kernels" topologically centered within a set of cores that cooperatively comprise the overall operating system in a peer-to-peer fashion. Such a system promotes rethinking and redesigning of various operating system services focusing on scalability as the primary design constraint.

We consider one such operating system service, namely job allocation in a distributed system. Job allocation in such a system benefits from the delegation of scheduling capabilities to micro-kernels but has to tackle additional challenges due to the distributed nature of job generation. A naive approach allowing fragmented allocations could quickly lead to deadlocks in the job allocation algorithm. Distributed job allocation protocols could avoid deadlocks during the allocation process by loosely enforcing a globally unique order. Hence, the hypothesis of this thesis is:

A scalable approach to job allocation can be provided through a distributed protocol. Such a distributed protocol can avoid deadlocks in the job allocation process by loosely enforcing a globally unique order.

1.3 Contributions

Following are the main contributions of this work:

- We propose the Pico-kernel Adaptive and Scalable Operating-system (PICASO) to address the scalability challenges of future manycore processors.
- We analyze the distributed job allocation problem and present a protocol with two policies, *active cancellation* and *sequencer-based atomic broadcast*.
- We evaluate the proposed solutions on the Tilera TilePro64 through a set of micro-benchmarks to analyze the performance and scalability.

1.3.1 PICASO system

We propose the Pico-kernel Adaptive and Scalable Operating-system (PICASO) for large-scale manycores. PICASO is a distributed message passing system devised to meet the scalability challenges of future large-scale manycore processors. The PICASO system consists of pico-kernels per core. These are normal worker cores where user tasks execute. A set of pico-kernels is managed by a micro-kernel. Micro-kernels are dedicated cores for control and management purposes. They are topographically centered within the set of pico-kernels that it manages. An advantage of such a system is the delegation of control and scheduling capabilities to micro-kernels for their respective set of pico-kernels. These micro-kernels, in concert, coordinate global scheduling in a decentralized manner across the entire manycore chip using distributed protocols. Hence, the operating system becomes a distributed system by design using message passing between cores and micro-kernels.

1.3.2 Distributed Job Allocation Protocol

We propose a novel protocol to tackle the challenges of job allocation in a distributed system. Allocating jobs of tasks on a partitioned system is known to be NP-Hard [11, 16]. The problem is further complicated in a distributed system due to the distributed nature of job generation. A naive approach allowing fragmented allocations could quickly lead to deadlocks in the job allocation algorithm. Our distributed job allocation protocol with two policies, *active cancellation* and *sequencer-based atomic broadcast*, takes a well disciplined approach in solving these issues. First, we avoid deadlocks by enforcing a globally unique order to resolve conflicting job allocations. Second, we split the job allocation problem into two subproblems: 1) query and reserve available resources, followed by 2) find a good task-to-core mapping. We believe such a split enables the best in class heuristics [5, 32] to tackle the NP-hard task-to-core mapping problem while our distributed job allocation protocol reserves available cores for the job.

Though our distributed job allocation protocol is generic in scheduling any application, in this work, we use Message Passing Interface (MPI) [26] applications as our standard workload for the following reasons: All ranks (tasks) of an MPI job need to start execution at the same

time. Such a workload demands guaranteed availability of cores to start execution or waits until they are available. This behavior enables us to model the job wait time as the overhead of the distributed job allocation protocol.

1.4 Organization

The rest of the thesis is structured as follows:

In Chapter 2 Section 2.1, we introduce the PICASO system for large-scale manycores. Section 2.2 describes the distributed allocation problem and the challenges in solving it. We describe our distributed job allocation protocol in detail in Section 2.3. Chapter 3 presents implementation details and Chapters 4 and 5 provide a detailed evaluation. We review the related work in Chapter 6 and conclude with Chapter 7.

Chapter 2

Design

2.1 The PICASO System

To meet the scalability challenges of future large-scale manycores, we have designed the Pico-kernel Adaptive and Scalable Operating-system (PICASO). PICASO features a distributed message passing system comprised of pico-kernels per core. Pico-kernels are worker cores on which user tasks of a job can be executed. A set of pico-kernels are managed by a micro-kernel. Micro-kernels are dedicated cores for control purposes, such as management of a set of pico-kernels and job scheduling in coordination with other micro-kernels. We use the term *micro-kernel domain* to refer to the set of pico-kernels governed by this micro-kernel. Micro-kernels are typically topographically centered within the set of pico-kernels that it manages.

A pico-kernel reports only to its parent micro-kernel. A micro-kernel, on the other hand, apart from controlling its set of pico-kernels, also co-ordinates with other micro-kernels. An advantage of such a system is the decentralization of control, where each micro-kernel may engage in fast and autonomous decisions on managing its set of pico-kernels. Since pico-kernels are just worker cores, we use the terms pico-kernels and cores interchangeably in this work. Figure 2.1 shows how a PICASO system with micro- and pico-kernel abstraction can be organized in a large-scale manycore system. In this figure, the available cores are partitioned into different domains represented by different colors. Each domain has a topologically centered core chosen to be the micro-kernel. The chosen micro-kernels shown in red are responsible for managing their set of pico-kernels and all external interactions occur only between micro-kernels.

2.2 The Distributed Job Allocation Problem

We use the following terminology in our discussion:

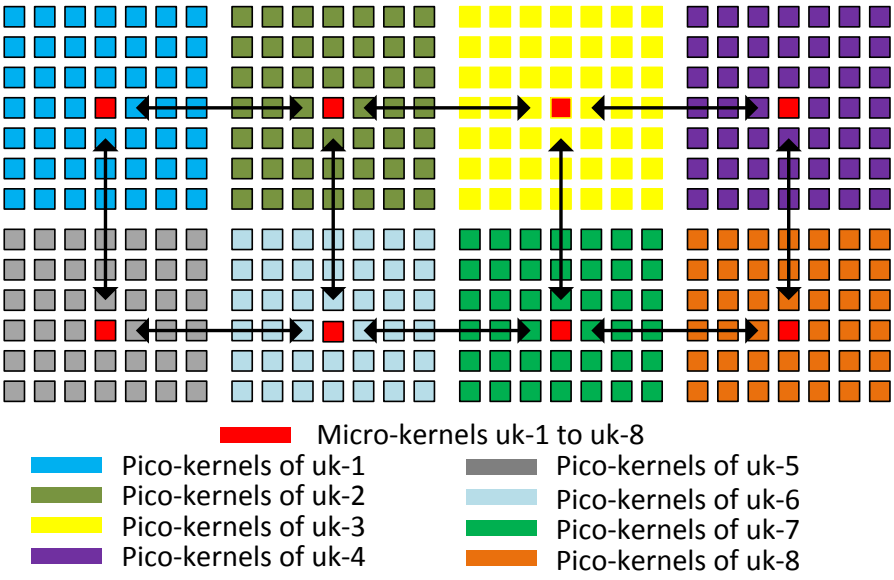


Figure 2.1: An example micro/pico kernel abstraction in a large many core processor

- A *task* is the basic unit of execution.
- A *job* consists of a collection of tasks.
- The *home micro-kernel* of a particular job is the micro-kernel where the job submission was initiated.

In this work, we consider jobs that require to be co-scheduled, i.e., these jobs consist of inter-dependent tasks that need to be concurrently executed on different nodes/cores. An example would be MPI jobs, where all associated ranks (tasks) need to start execution at the same time. We also assume that once tasks begin execution, they will run to completion without preemption. This assumption is in line with our vision of an era of dark silicon [22], where cores are abundant (in the order of the number of tasks). It will enable software tasks to be readily mapped onto cores. On such a system, scheduling of tasks amounts to core activation rather than context switching [6, 29]. Hence, techniques for devising performance efficient job allocations will be of importance. Such an allocation usually consists of two steps:

- Query available idle cores and reserve them for this job.
- Devise an efficient task-to-core mapping from the available cores.

Our focus in this work is on the former part. Once enough cores are reserved for a job, methods and results from prior work [5, 32] can be applied to find the best task-to-core mapping for a

given job. However, the problem becomes more complicated when extended to a distributed system due to the nature of job generation.

Conventional solutions involve a centralized resource manager that handles all job allocations. All cores continuously report their availability status to this entity. But such an approach does not scale to a large number of cores. One reason is the contention at the centralized entity because of the incessant status updates. The other reason is that this leads to a single point of failure. But more importantly, it allows for only a single job submission portal. These restrictions are undesirable for large core counts where jobs generate allocations and queue up for their execution at different cores of the system.

Our proposed pico-/micro kernel distributed system abstraction partitions the available cores between different micro-kernels. This domain-specific delegation of scheduling capabilities to micro-kernels enables jobs that can be locally satisfied within a single micro-kernel domain to be handled by fast and autonomous decisions. For jobs requiring more cores than can be locally satisfied, the *home micro-kernel*, where the particular job is submitted, co-ordinates with other micro-kernels to devise the allocation of cores to this job. Multiple job requests submitted at different micro-kernels could compete with each other for resources. Hence, we need a co-ordination protocol to resolve these conflicts and to choose the next job to execute loosely based on a globally unique order. This global unique order could be based on user-defined priority or a First Come First Serve (FCFS) policy. Such an ordering guarantees fairness and avoids starvation. Adhering to loose ordering rather than strict ordering allows non-conflicting job allocations to proceed in parallel, thereby increasing the system utilization.

But a lack of such co-ordination protocols may lead to potential deadlocks. Deadlocks can happen when multiple jobs submitted at different micro-kernels hold different subsets of cores and wait for more cores to become available. Yet, none are able to proceed because all cores have been allocated to jobs without meeting the full allocation request of any single job in full. Figure 2.2 shows a possible deadlock condition with two micro-kernel domains. Each domain has initially 8 pico-kernels (worker cores) available. In step 1, we have two job submissions requiring 12 and 16 cores, respectively. In step 2, each job first holds on to available local cores and sends out a request for more cores. In step 3, each micro-kernel is blocked waiting indefinitely for their job requests to be satisfied. Since none of the job requests are fully satisfied, the system remains deadlocked.

Random back-off schemes could be used to recover in case of potential deadlocks. In such a method, different micro-kernels yield their cores and retry their job allocations after waiting for a randomly chosen back-off time. This probabilistically avoids a deadlock again, but fails to guarantee a bound on completion time for the allocation algorithm. Hence, such schemes may not be applicable to real-time systems where upper bounds on completion times are mandated. A more serious issue is potential starvation of jobs that require large allocations as they might

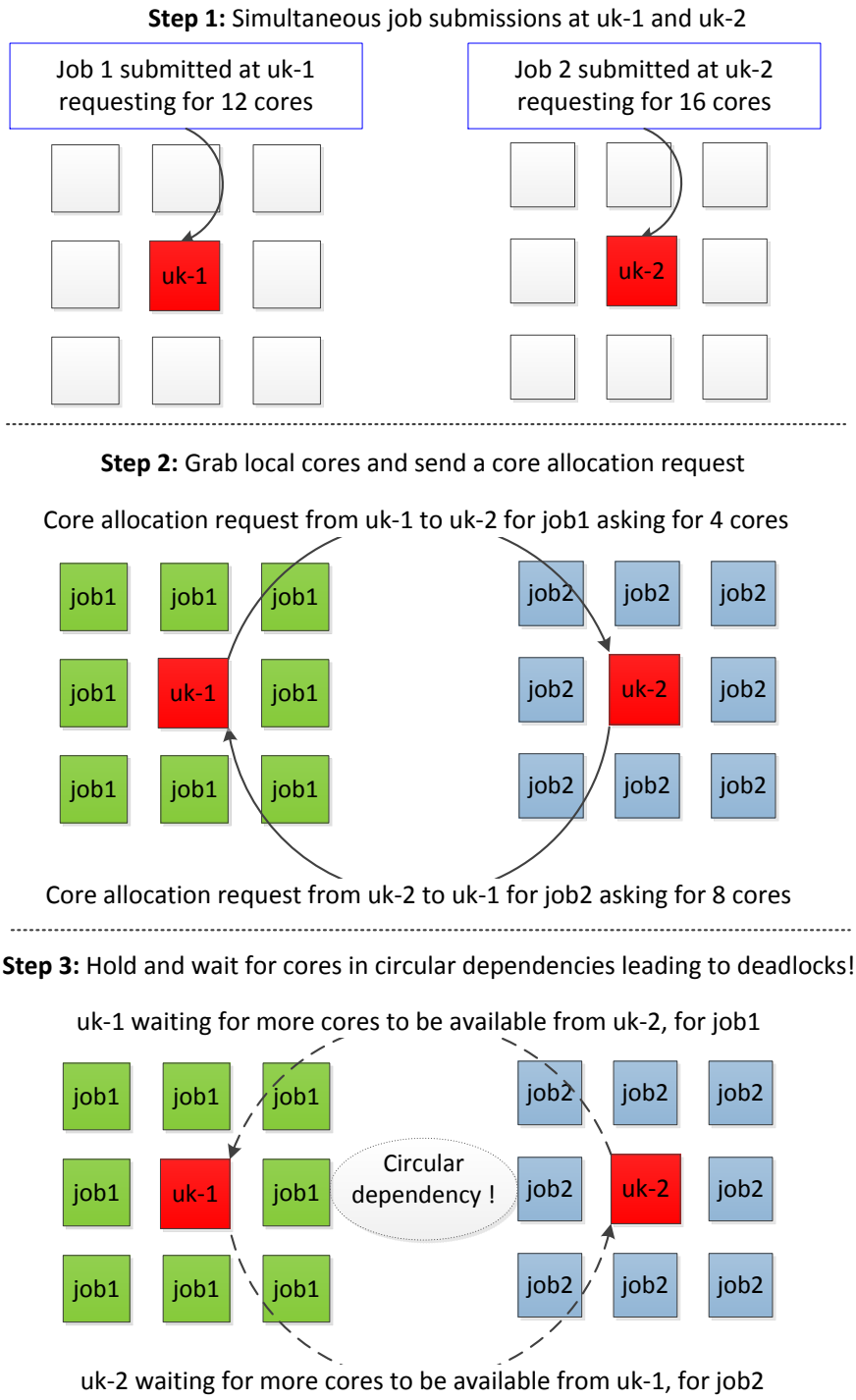


Figure 2.2: A probable deadlock involving two simultaneous job submissions at different micro-kernels

never be satisfied. Therefore, a job allocation algorithm that avoids starvation with an upper bound on completion time is required.

2.3 Deadlock free Distributed Job Allocation Protocols

We have devised a distributed job allocation protocol for large-scale manycores. Two policies for deadlock avoidance are proposed, namely

1. active cancellation and
2. sequencer-based atomic broadcast.

Both of these policies require that a globally unique order be established. For example, we could use timestamps of the job submission time along with the micro-kernel identifier to devise a globally unique job identifier, or we could use user-defined priorities in conjunction with a method to break ties for matching priorities. For the discussion in this work, we will refer to job priority based on a globally unique job ordering rather than a user-defined priority. In the following sections, we examine the two different approaches, compare their capabilities and finally conclude with a detailed performance evaluation.

2.3.1 The Main Scheduling Loop

Irrespective of the policy used, each micro-kernel runs an event-based main scheduling loop. To implement such a main scheduling loop in an MPI-like fashion (rather than a socket based communication framework) poses two challenges. The micro-kernel should be able to:

1. perform wild-card receives from any arbitrary micro-kernel or one of the pico-kernel it manages;
2. receive messages of arbitrary length. To support this, messages are constructed in two parts: a fixed sized header indicating message type and message length, followed by the actual message body.

Algorithm 1 shows the main scheduling loop. It performs two main functions:

1. Process any incoming message, and
2. in the absence of an incoming message, schedule pending job requests submitted at this micro-kernel.

Algorithm 1 Scheduling loop at each micro-kernel

```
while TRUE do
  Post a nonblocking receive for the fixed size header
  repeat
    if policy == active cancellation then
      call periodic active cancellation specific procedure
    else if policy == atomic broadcast then
      call periodic atomic broadcast specific procedure
    end if
  until fixed size header is received
  Receive the entire message body blocking
  call respective message handler routine
end while
```

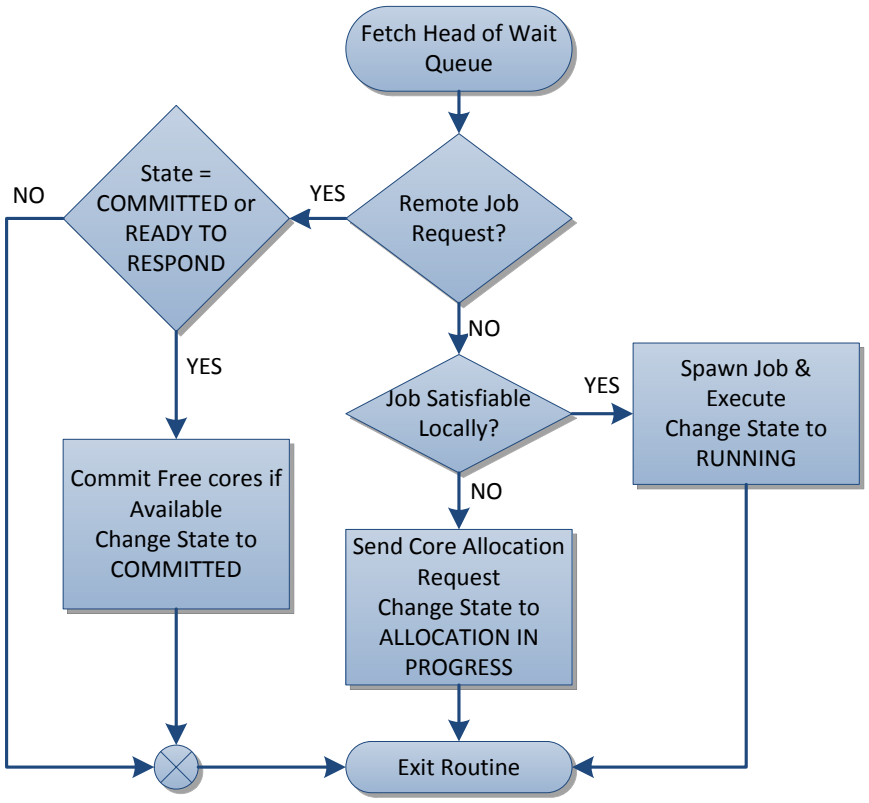


Figure 2.3: Periodic active cancellation procedure

The scheduling loop uses message passing as the only means of communication between micro-kernels, and between a micro-kernel and its set of pico-kernels. There can also be architecture-specific optimizations for micro- to pico-kernel communication, not shown here.

The significant message types of the distributed job allocation protocol are as follows:

Core Allocation Request:

Sent by the *home micro-kernel* of the job. The request is propagated to all micro-kernels via an efficient request propagation scheme.

Core Allocation Response:

Sent by a micro-kernel when it commits certain cores to a particular job.

Job Spawn Request:

Sent by the *home micro-kernel* when it devises the best task allocation for the given job. This request follows the same propagation path earlier traversed by the *Core Allocation Request*. Micro-kernels that are not part of an allocation, release their reservations for this job when they receive this request.

Job Cancel Request:

When the *active cancellation* policy is used, this message is sent by the *home micro-kernel* if it determines that there is an higher priority job to be satisfied first.

Submit Job to Sequencer:

Under the *sequencer-based atomic broadcast* policy, all micro-kernels use this message to submit their job requests to the fixed sequencer (see Subsection **??**).

2.3.2 Deadlock Avoidance by Active Cancellation

The periodic *active cancellation* procedure is depicted pictorially in the flowchart shown in Figure 2.3. In this method, any micro-kernel that launches a job requiring more than the locally satisfiable cores sends a core allocation request to all its neighbors. This request is propagated to all other micro-kernels via an efficient request propagation scheme as explained in Section 2.3.4. A greedy policy is employed, wherein the request to each micro-kernel always asks for the maximum number of cores needed for the job. This policy frequently allocates more cores than needed for a job, but guarantees a successful allocation.

The flowchart in Figure 2.4 depicts how active cancellation is triggered when a high priority core allocation request arrives. Each micro-kernel maintains a wait queue based on the globally unique order consisting of both the job requests it has sent out and the job requests it has received. All incoming job requests are inserted in the wait queue as per the globally unique ordering. If the new request happens to be the head of the wait queue, it first checks if this

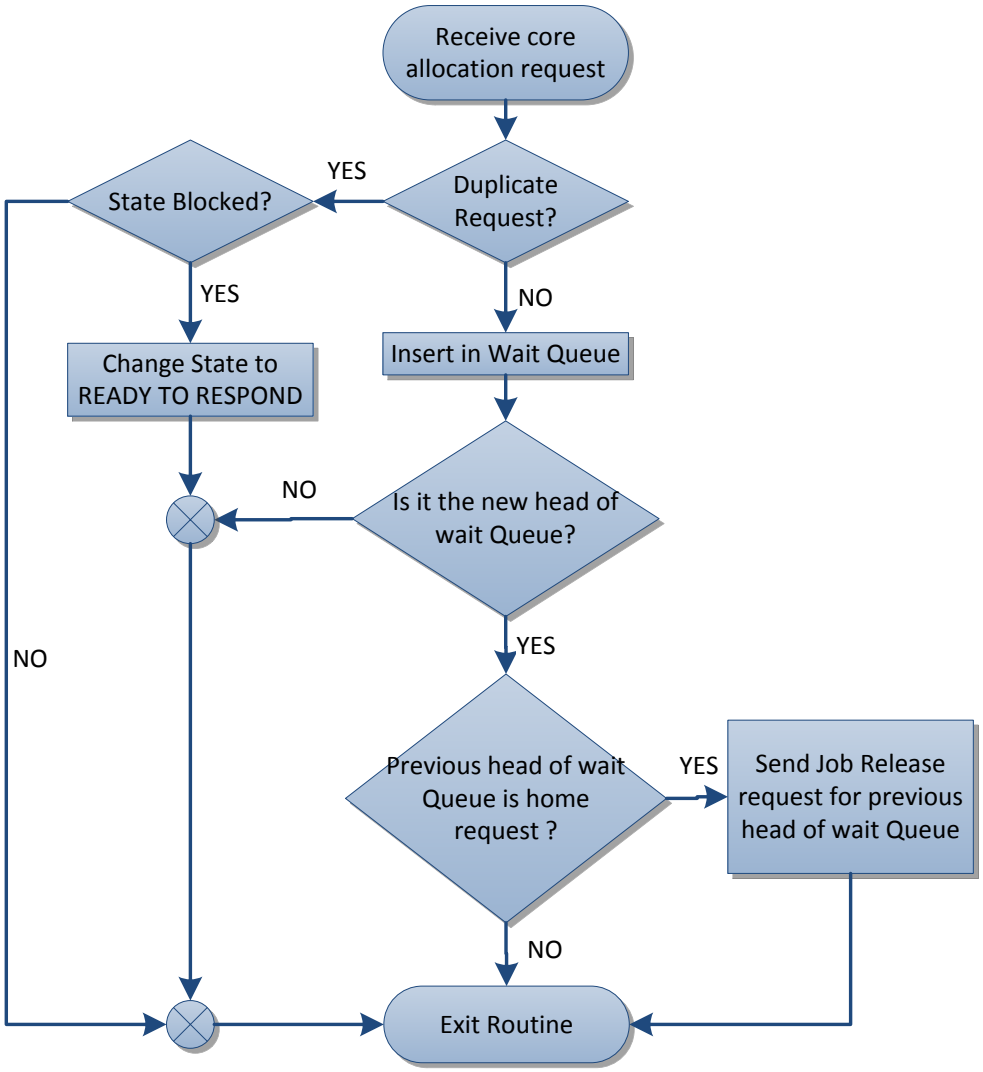


Figure 2.4: Steps in processing core allocation request for active cancellation

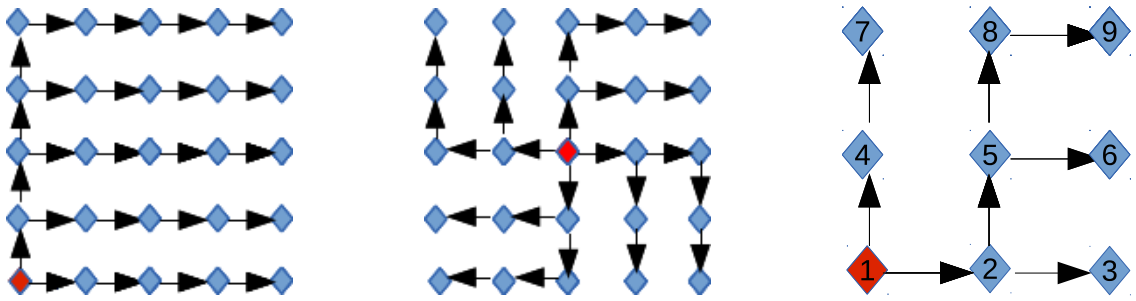
request has a higher priority than any job request it has sent out earlier. If so, it engages in active cancellation of the lower priority job changing it to the BLOCKED state pending a renewed request. This frees up resources otherwise allocated to unsuccessful lower priority job requests. Finally, the micro-kernel commits how ever many cores it can afford for this job request by responding with the committed cores to the *home micro-kernel* of this particular job request. The micro-kernel contributes new cores to this commitment whenever its resources become free. This scheme satisfies multiple job requests *loosely* based on the global ordering but also offers a relaxation to this hard criteria by allowing a lower priority request to proceed if its allocation is satisfied quickly enough before a higher priority job overrides it in the wait queue. This relaxation is allowed under the assumption that any job using a successful allocation will *eventually* complete, after which time the resources it was given becomes available for the next high priority job in the wait queue (bounded by the longest job).

2.3.3 Deadlock Avoidance by Sequencer Based Atomic Broadcast

This method is inspired by the sequencer based atomic broadcast as explained in Xavier et al. [13]. In this method, a micro-kernel is elected to be the single sequencer of the system. All job requests, even if submitted at different micro-kernels, are in turn submitted to the sequencer. The sequencer ensures the globally unique ordering and sends the request to all the micro-kernels only when it determines which job to execute next. Our approach differs here. Instead of broadcasting the request, we use a custom built request propagation scheme as explained in Section 2.3.4. This ensures that the job allocations happen in order without any collisions. Less conflicts directly translate to fewer messages compared to *active cancellation*. But since each micro-kernel has to send requests to the sequencer, it leads to contention at the sequencer and additional delays even for small allocation requests, which could have been solved with just a few neighboring micro-kernels. As we show in Section 5, this additional overhead translates into real performance benefits only in case of dense and large job allocations.

2.3.4 Pattern-Based Message Propagation Schemes

An efficient method for propagating request messages, such as core allocation and job spawn requests from any given source to all other micro-kernels in a 2D mesh topology, is required. Multi-casting messages from a given source to all micro-kernels is inefficient as this involves sending individual messages to each micro-kernel, unless hardware support for multi-casting exists [12]. Therefore, we have designed and implemented two alternatives: 1) a fixed pattern-based propagation scheme and 2) an adaptive pattern-based propagation scheme. We use the term *nodes* when introducing these schemes, as these schemes not only apply to micro-kernels but any set of nodes in a 2D mesh topology. The adaptive pattern-based propagation scheme



(a) Fixed pattern source at lower left

(b) Fixed pattern source at center

(c) An example adaptive pattern

Figure 2.5: Pattern-based request propagation schemes

has the advantage that it does not expect nodes to be arranged in a 2D mesh topology.

Fixed pattern-based propagation

When a message needs to be sent to all nodes in a 2D mesh processor NoC, the source sends the message only to its neighboring nodes. Each neighbor in turn propagates the request to its next set of unvisited neighbors following a predefined pattern. The pattern depends on the placement of the initial source of the message. Consider Figure 2.5a. The source initially sends the request to all its neighbors with an embedded information to propagate the request toward the East direction. Each node receiving this message propagates the request as per the embedded information. Similarly, if the source is located at bottom-right, the propagation will be toward North; if located at the top right, toward West; and if located at the top left toward South. Figure 2.5b shows the pattern when the source is located at the center, in which case each arm takes the responsibility of propagating the request in all four directions. Following such a predefined pattern avoids duplicate requests, which waste link resources and increase processing time at the nodes.

Adaptive pattern-based propagation

This scheme involves an initialization phase responsible for forming the adaptive pattern. In this phase, an empty message is forwarded from the given source to all its neighbors. Each neighbor in turn broadcasts the message to all of its next set of neighbors until all the nodes have been visited. At this point, each node has received the given message from multiple sources. It chooses one among these sources as a preferred source and informs it. The preferred source remembers this decision and forwards all messages it receives to this node. The criteria to choose the preferred source can be based on various policies, e.g., the first received request or shortest distance from the source to this node, to name a few. At the end of the first phase, every node

has identified its preference from which source it wishes to receive a request in the future; or, alternatively, each node has remembered a list of neighbors to forward a message to that was received from a particular source. This forms an adaptive pattern ensuring each node receives a message only once. An advantage of such adaptive patterns compared to fixed patterns is that the patterns could be adaptively rearranged in case of link failures. The initialization phase needs to be run only once during the system startup or when recovering from faults, hence reducing the overhead by amortizing the costs.

As an example, consider the pattern shown in Figure 2.5c for a 3×3 tile with numbered nodes. This pattern is formed with 1 as the source node and forwarding paths from nodes 1 to 2 & 4, 2 to 3 & 5, 4 to 7, 5 to 8 & 6 and 8 to 9.

Chapter 3

Implementation

The distributed job allocation protocols are applicable to any system of inter-networked cores, even heterogeneous cores [1, 3, 4, 12]. But for the purpose of implementation and experimentation alone, the job allocator has been optimized for a 2D-mesh architecture, such as the Tilera TilePro64 [4, 30]. The Tilera TilePro64 processor has 64 tiles interconnected with a 2D-Mesh Network-On-Chip (NoC) interconnect. Each tile has a processor engine running at 700 MHz, a switch engine for routing on the NoC over five different network interconnects and a cache engine. The User Dynamic Network (UDN) interconnect is the only one available for user-generated messages. We use the services of the NoCMsg [33] library. NoCMsg provides a deadlock free, scalable and efficient low-level message passing layer over UDN with an MPI like interface. This motivated our design choice and, hence, our scheduling loop. The protocols and the messages were designed entirely around these MPI like interfaces. This, in itself, makes our design generic enough to be ported to other message passing libraries as well.

3.0.5 Global Unique Ordering

For our experiments, we use an ordering based on a FCFS policy. Each tile on the TilePro64 has synchronized clocks. Hence, we use the time-stamp of the job submission along with the unique micro-kernel identifier of the job's home micro-kernel as a tie breaker for job submissions.

3.0.6 Extensions to the NoCMsg Library

We extended the NoCMsg library to support the following features:

- ability to perform wild-card receives;
- ability to post multiple non-blocking sends and receives.

Our Main Scheduling loop requires wild-card receives. Hence, NoCMsg was extended to support this functionality by providing *MPI_ANY_SOURCE/ MPI_ANY_TAG* parameters. We introduce two queues, namely the unhandled message queue and unmatched request queue. Whenever a new incoming message arrives, it is checked for a matching *MPI_Recv/MPI_Irecv* request posted earlier by scanning the unmatched request queue; otherwise, the new message is queued in the unhandled message queue. Similarly, all *MPI_Recv* and *MPI_Irecv* requests, when posted, are checked against the unhandled message queue for a potential match. Upon a mismatch, the request is queued in the unmatched request queue. Similarly, queues were introduced for send operations to support multiple non-blocking send requests to the same source/tag combination. Processing messages via these queues provides support for both our requirements.

Chapter 4

Evaluation Framework

We use the TilePro64 processor [4] for our evaluation. While the TilePro64 supports 64 tiles, at least two tiles are reserved exclusively by Tilera's hypervisor for administrative tasks and Input/Output operations. The maximum square tile size that can be reserved for user tasks is 7×7. We choose a square tile size so as to eliminate possibilities of discrepancies due to other asymmetric tile sizes. We support two different experimental frameworks for testing the performance of the job allocator,

1. a *real task* mode, and
2. a *partial simulation* mode.

The *real task* mode, supports execution of MPI jobs, such as a subset of NAS Parallel benchmarks (NAS-PB). Figure 4.1 shows the *real task* mode on the Tilera TilePro64 processor. This small PICASO system on a 6×6 tile has been divided into four regions. Each region has a topologically centered micro-kernel managing a set of eight pico-kernels. Using this platform, a

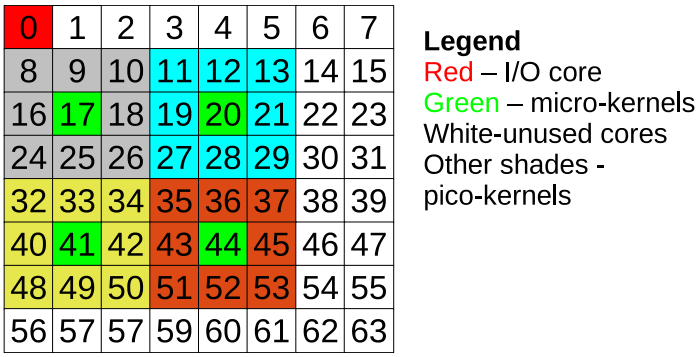


Figure 4.1: PICASO system within a 6×6 tile on Tilera TilePro64

combination of NAS-PB of power of two sizes (1,2,4,8,16 and 32) can be executed. This platform is primarily used to assess the schedulability of real user tasks.

The limited number of usable cores on the TilePro64 constraints our scalability tests on the *real task* mode. To overcome this, we have developed a *partial simulation* framework, where we consider all cores in the reserved tile as micro-kernels without pico-kernels. Task execution is simulated by timers triggering a job completion message after a certain user-defined execution time. This simulation platform is justified by the fact that the distributed job allocation protocol requires only micro-kernel interaction. Our results could be directly translated to the *real task* mode combining them with the pico-kernel management overheads obtained in the *real task* mode. This *partial simulation* mode provides the ability to assess our protocol with up to 49 micro-kernels on a 7×7 tile.

The following chapters detail the experiments/results under the *real task* mode and the *partial simulation* mode for different job allocation mixtures.

Chapter 5

Experimental Results

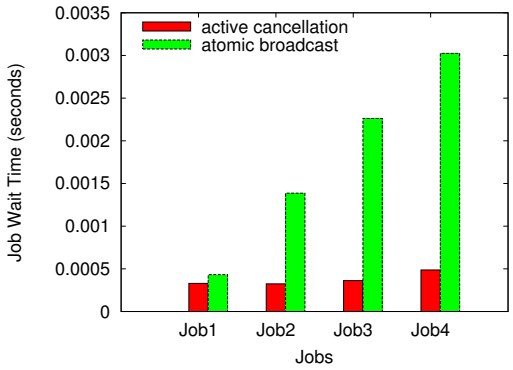
The distributed job allocator and all user programs are compiled as applications with Tilera's MDE 3.03 tool chain at the O3 optimization level using Tilera's C/C++/Fortran compilers.

5.1 Performance Analysis

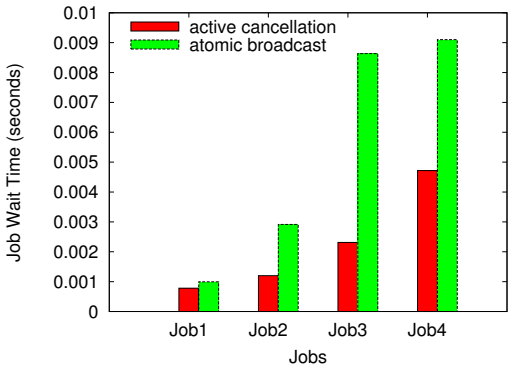
To analyze the performance of the two proposed schemes, we first use the *partial simulation* mode. We execute a set of micro-benchmarks. For each job, we measure the job allocation overhead as the wait time of the job from the time of submission to the time it receives all the resources to execute. This wait time includes both the overhead of the distributed job allocation protocol and the time spent waiting for the earlier job allocations to terminate and to release its cores. Our focus is to measure the overhead of the distributed job allocation protocol in isolation. Hence, for performance tests, we use an initial state where no jobs are active. We then trigger simultaneous job submissions from different micro-kernels as they have the highest probability to result in fragmented allocations. This creates a workload for our protocol triggering its deadlock avoidance subsystem. Note that all our experiments cover cases where the job allocations require large number of cores that need more than one micro-kernel domain to be fully satisfied. Recall that job allocations, which could be satisfied within a single micro-kernel domain, have a constant overhead.

For all our experiments, the reported job wait times are averaged over 15 runs. The maximum relative standard deviation observed in all these experiments was less than 20%, except for the experiment shown in Figure 5.1b, where we observed relative standard deviations of up to 41%. We discuss this exception and other significant experimental details in the following relevant sections.

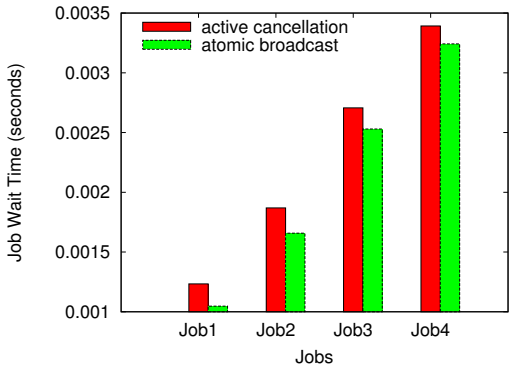
In our experiments, we compare both our proposed polices, *active cancellation* and *sequencer-based atomic broadcast*, against one another. When reporting the relative performance improve-



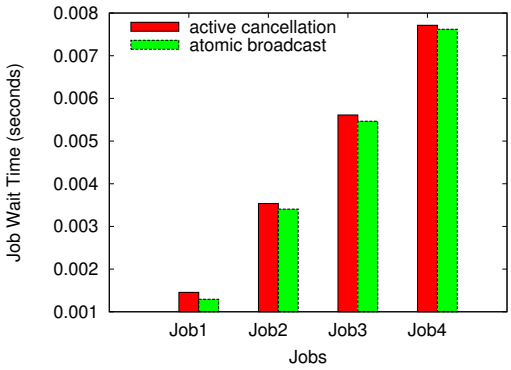
(a) 3 micro-kernels



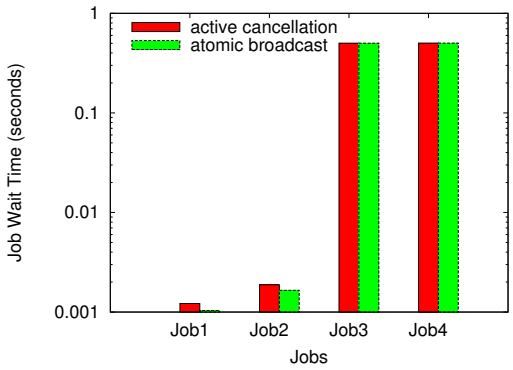
(b) 12 micro-kernels



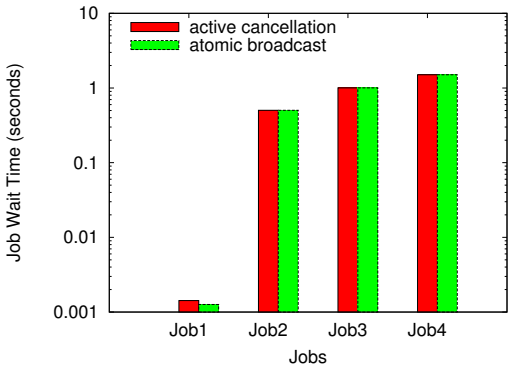
(c) 24 micro-kernels (short runs)



(d) 49 micro-kernels (short runs)



(e) 24 micro-kernels (long runs)



(f) 49 micro-kernels (long runs)

Figure 5.1: Micro-benchmark — Overhead for Sparse Job Allocations
where each job requires a number of pico-kernels (cores) that is satisfied by:

ment or degradation, we always follow the convention of comparing *active cancellation* against *sequencer-based atomic broadcast* as follows:

Let the overhead of *active cancellation* be denoted as O_{ac} and the overhead of *sequencer-based atomic broadcast* be denoted as O_{ab}. Then the relative performance change of *active cancellation* is given by:

$$\frac{(O_{ab} - O_{ac})}{O_{ab}} \times 100\%$$

5.1.1 Micro-benchmark — Overhead for Sparse Job Allocations

This experiment uses the *partial simulation* mode. Job allocation requests are generated simultaneously from the four extreme corners of a 7×7 tile. These requests can be satisfied with just a few nearby micro-kernels even before the conflicting job requests arrive from the other corners. Hence, in most of these cases, cancellation of the lower priority job request may not even be required as all the simultaneously submitted jobs are satisfied without the need for global ordering. Conversely, with *sequencer-based atomic broadcast*, all requests have to still go to the single sequencer, which can only serve one request at a time so that serialization delays impact these small job allocations. This experiment proves that *active cancellation* provides best performance in scenarios where sparse job submissions can proceed in parallel.

In the following set of experiments, we consider two scenarios: Jobs that can be execute in parallel and jobs that need to be executed serially one after another.

Jobs that can execute in parallel

Figures 5.1a and 5.1b depict the scenario where each job can proceed in parallel. For the four jobs shown on the x-axis, their corresponding job wait times are depicted on the y-axis. The job wait time does not include execution times of prior jobs as all these jobs execute in parallel. Hence, the measured job wait time can be considered as the exclusive protocol overhead. We observe a relative decrease in the job wait times for *active cancellation* when compared to *sequencer-based atomic broadcast*.

In the first experiment (see Figure 5.1a), each job requires a number of pico-kernels (cores) that is satisfied with available cores from 3 out of the total 49 micro-kernel domains. We observed a relative performance improvement for *active cancellation* over *sequencer-based atomic broadcast* of 23% for the first job, 76% for the second job and 83% for the third and fourth jobs.

In the second experiment (see Figure 5.1b), each job requires a number of pico-kernels (cores) that is satisfied with available cores from 12 out of the total 49 micro-kernel domains. The relative performance improvement of *active cancellation* over *sequencer-based atomic broadcast*

for the four jobs were: 21% for the first job, 58% for the second job, 73% for the third job and 48% for the fourth job. For *active cancellation*, we observe a maximum relative standard deviation of 41% in this experiment, which is explained as follows: The wait time of each job depends on how many cancellations are required after the first job has been successfully allocated. In some runs, we observe that a lower priority job request propagated fast enough to succeed in its allocation before a higher priority job triggers the cancellation procedure. In these cases, the job wait times for the lower priority jobs are reduced. They are otherwise above average if more cancellations are involved.

Jobs that require serial execution

When jobs execute serially, job wait times depend largely on the execution times of preceding jobs. When the execution time of prior jobs is high, this becomes the main contributor to the job wait time. Conversely, when the execution time is lower than the minimum job allocation overhead, then the overhead of the distributed job allocation protocol is the main contributor to the job wait time. Hence, for the next two experiments, we consider both short and long running jobs. Short running jobs help assess the actual overhead of the two polices. Long running jobs demonstrate that for serially executing jobs, this performance improvement is not entirely carried over as a reduction in the job wait times.

Short Running Jobs: We set the job execution times to 0.001 seconds, which is below the minimum overhead observed. Figure 5.1c depicts a case where each job requires a number of pico-kernels (cores) that is satisfied by exactly 24 out of the total 49 micro-kernel domains. Hence, two out of the four jobs can run in parallel. As not all jobs can run in parallel, allocations of the lower priority jobs require cancellation so that the allocation of higher priority jobs is satisfied. This results in an additional overhead for *active cancellation* compared to *sequencer-based atomic broadcast* of around 17% and 12%, respectively, for the first two jobs, but considerably less for the next two jobs (7% and 4%, respectively). Figure 5.1d depicts the case where all jobs require a number of pico-kernels (cores) that is satisfied by exactly all available 49 micro-kernel domains and, hence, execute serially one after another. Here, *active cancellation* incurs an additional overhead as lower priority job allocations need to be canceled to enforce the globally unique order. This additional overhead for *active cancellation* is around 12% for the first job and reduces considerably to 4% for the second job, and then to around 1% for the fourth job.

Long Running Jobs: For these experiments, we set the job execution times to 0.5 seconds, which is much higher than the overhead of the distributed job allocation protocol. Hence, in these cases, the execution time is the main contributor to the job wait time. During the initial execution delay for the spawned jobs, the job allocation protocol reorders the job wait queue.

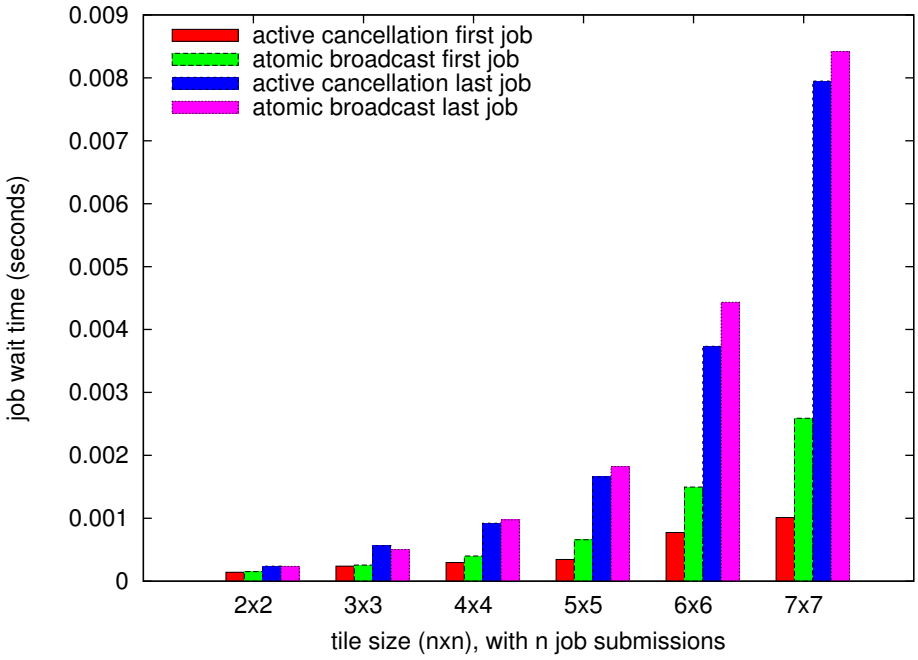


Figure 5.2: Micro-benchmark — job allocations as tile size increases

Therefore, subsequent jobs are spawned as soon as the earlier jobs complete with a minimal overhead. Figure 5.1e depicts a case where each job requires a number of pico-kernels (cores) that is satisfied by exactly 24 out of the total 49 micro-kernel domains. Hence, two out of the four jobs can run in parallel. Job wait times are depicted on the y-axis on a *logarithmic* scale. Here, *active cancellation* incurs an additional overhead of 17% and 13%, for the first two jobs, respectively. The additional overhead for the next two jobs is very minimal (0.03% to 0.05%). Figure 5.1f depicts the case where all jobs require a number of pico-kernels (cores) that is satisfied by exactly all available 49 micro-kernel domains and, hence, execute serially one after another. Job wait times are depicted on the y-axis on a *logarithmic* scale again. Here, *active cancellation* incurs an additional overhead of around 12% for the first job, but very minimal for subsequent jobs (0.01% to 0.04%). Hence, the above experiments show that for long running jobs, which execute serially one after another, the performance gain achieved by *sequencer-based atomic broadcast* is minimal.

5.1.2 Micro-benchmark — Job Allocation Overhead as Tile Size Increases

In this experiment, we scale the tile size ($n \times n$) from 2×2 to the maximum supported size of 7×7. For each tile size, we generate n simultaneous job requests, each requiring pico-kernels (cores) that is satisfied by exactly n micro-kernel domains. For example, in a tile size of 2×2,

there will be 2 simultaneous job requests requiring pico-kernels (cores) that is satisfied by 2 micro-kernels each, and in a tile size of 7×7, there will be 7 simultaneous job requests requiring pico-kernels (cores) that is satisfied by 7 micro-kernels each. Thus, this experiment shows the additional overhead for jobs that can ideally execute in parallel. The results depicted in Figure 5.2 compare the job wait times of the first and last jobs for *active cancellation* and *atomic broadcast*. Here, the job wait times are depicted on the y-axis for different tile sizes on the x-axis. We observe that the wait time for the first among the n jobs is consistently lower for *active cancellation* as it does not incur the overhead of submitting all job requests at the sequencer. We observe a reduction in the job wait time of the first job from 6% for a tile size of 2×2 to up to 60% for a tile size of 7×7. For the sake of analysis, let us assume that the highest priority job overrides all other jobs in their home micro-kernels before any of the lower priority jobs gets a chance to execute. In this case, there will be one initial request sent for the highest priority job. For all other lower priority jobs, there will be $n-1$ initial requests plus $n-1$ cancel and finally $n-1$ repeat requests sent in total. Thus, all subsequent jobs incur this additional overhead. Notice that significant performance gains in spawning the first job compensates for this additional overhead for subsequent jobs to a large extent. When compared to *sequencer-based atomic broadcast*, we observe a slight increase in the job wait times for *active cancellation* (in the range of 1% to 12% for smaller tile sizes, i.e., 2×2 and 3×3). But for larger tile sizes, we observe a more significant reduction in the overhead for *active cancellation* of up to 15%. This experiment reinforces our earlier finding that as long as multiple simultaneous job submissions can execute in parallel, *active cancellation* has a lower overhead compared to *sequencer-based atomic broadcast*.

5.1.3 Micro-benchmark — Worst-case Conflict Resolution Time for n Simultaneous Job Submissions

In this experiment with n simultaneous job submissions, we measure the conflict resolution time for the first job to execute. We use a fixed tile size of 7×7 in the *partial simulation* mode. As all the cores are considered to be micro-kernels in this mode, a maximum of 49 micro-kernels are available. All job submissions require a large number of pico-kernels (cores) that can only be satisfied by the cores available in all the 49 micro-kernel domains. In this worst-case scenario, the *sequencer-based atomic broadcast* scheme provides the best performance. The *sequencer-based atomic broadcast* scheme just has to wait for the job allocation request with the highest priority to arrive. It can then send out core allocation requests one after another. The maximum overhead occurs when the highest priority job request is the one that reaches the sequencer last. Compare this to the considerable overhead in *active cancellation*. Here, in the worst-case, the $n-1$ lower priority job requests together could have reserved all available

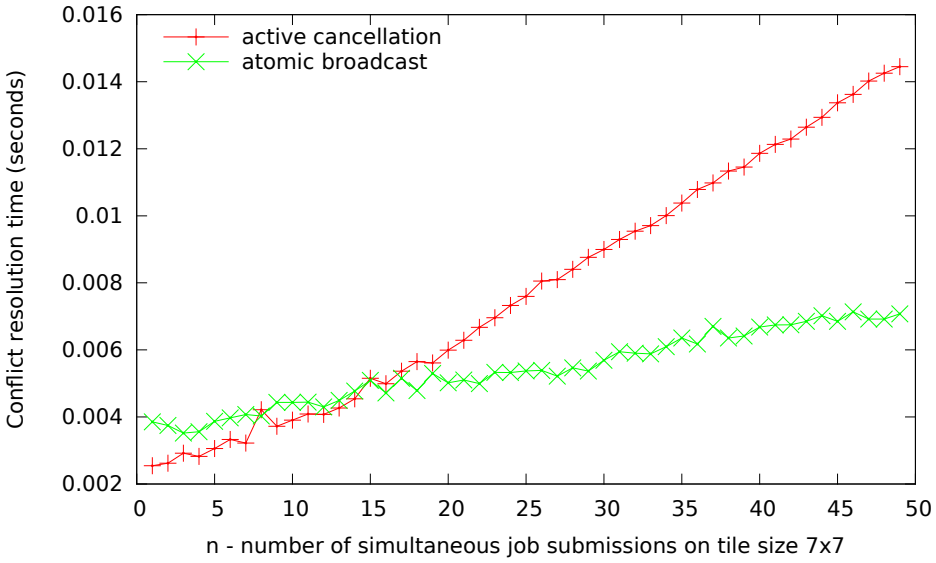


Figure 5.3: Micro-benchmark — Worst-case conflict resolution time for n simultaneous job submissions

cores in all micro-kernels. But none would have reserved enough to proceed executing. Hence, for the highest priority job request to execute, it has to override each of the lower priority job request in all other micro-kernels by sending job cancel requests. In the worst-case, $n-1$ cancellation requests need to be sent before the first job can get enough cores for its allocation to be satisfied. We see this reflected in Figure 5.3. The wait time for the first job is shown on the y-axis and x-axis depicts n, the number of simultaneous job submissions. We observe that the worst-case performance is better for the *sequencer-based atomic broadcast* scheme once the number of micro-kernels simultaneously requesting allocations exceeds $1/4^{\text{th}}$ of the total number of micro-kernels.

5.2 Experiments in Real Task Mode

The *real task* mode on the TilePro64, introduced in Section 4, consists of 4 micro-kernels, each managing a set of 8 pico-kernels. We can execute jobs that require a maximum of 32 cores in this mode. To confirm the pattern observed under the *partial simulation* mode, we conduct similar, yet scaled down experiments in *real task* mode.

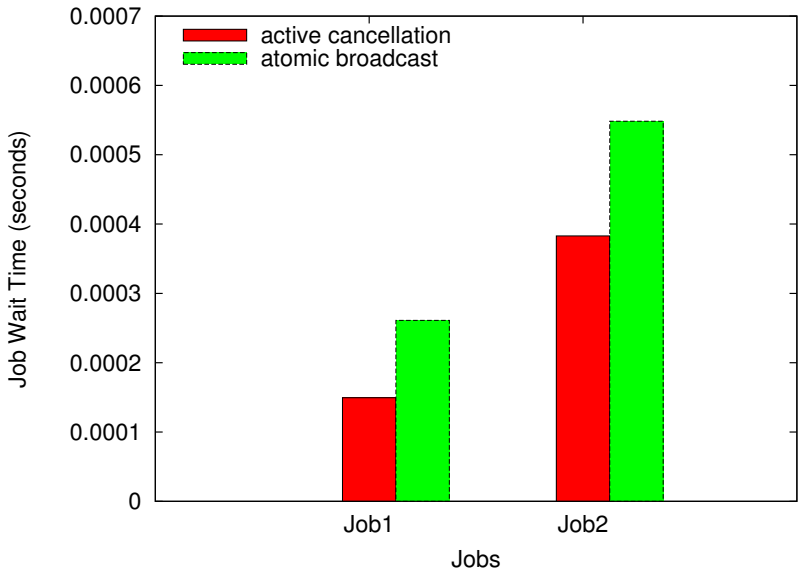


Figure 5.4: Real task mode — Parallel Job Allocations

5.2.1 Job Allocations that can execute in parallel

In this experiment, two jobs (NAS Parallel Benchmark FT Class=S size=16) run in parallel in two different micro-kernel domains. Each job requires 16 cores, which can be satisfied in parallel. We measure the average job wait time. Here, the wait time is exclusively due to the protocol overhead as it does not include any resource wait time. This experiment is an approximation of the sparse job allocations explained in the context of the *partial simulation* mode. We observe results following the same pattern: Under *active cancellation*, less overhead is incurred compared to *sequencer-based atomic broadcast*. These results are shown in Figure 5.4 where y-axis depicts the job wait time in seconds for the two jobs executing in parallel (on the x-axis).

5.2.2 Job allocations which can only execute serially

In this experiment, four jobs (NAS Parallel Benchmark FT Class=S size=32) requiring all the 32 cores available from all of the four micro-kernels are submitted simultaneously. These job submissions compete for all resources and are eventually serialized to execute one after another. Thus, this experiment is similar to the *partial simulation* mode experiment in Section 5.1.3, which measured the worst-case conflict resolution time for n simultaneous job submissions. We obtain similar results, where *sequencer-based atomic broadcast* performs much better than *active cancellation*. Figure 5.5 shows these results with the exclusive job wait time on the y-axis for the four jobs on the x-axis. Exclusive job wait time is calculated here as the actual job wait

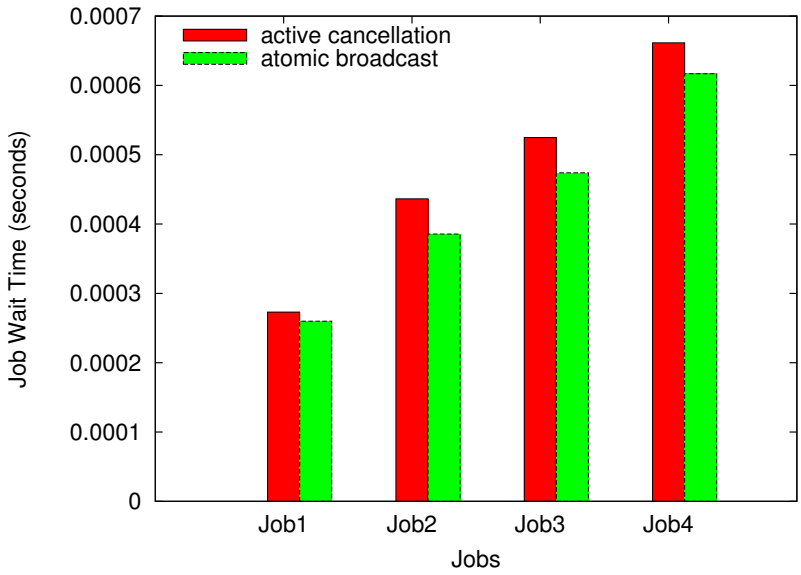


Figure 5.5: Real task mode — Serial Job Allocations

time minus execution times of all prior jobs. This metric provides the job allocation overhead in isolation.

5.3 Performance of the Pattern-Based Propagation Schemes

To evaluate the pattern-based message propagation schemes, a simple experiment was devised. A request is broadcasted to all nodes (cores in this case) in a 7×7 tile (maximum nodes = 49). The time spent to broadcast this request and to receive a reply from all endpoints in the reverse path of broadcast is measured. The results in Figure 5.6 compare the time taken on the y-axis against the number of nodes to which the message is broadcast on the x-axis. Four different schemes are compared:

1. A naive broadcast scheme: The source sends m individual messages to m recipients.
2. Distributed flooding: The source sends the message to all its neighbors and each node receiving the message again multi-casts the message to all its neighbors until all the nodes have received the message.
3. Fixed pattern-based propagation: Explained in Section 2.3.4.
4. Adaptive pattern-based propagation: Explained in Section 2.3.4.

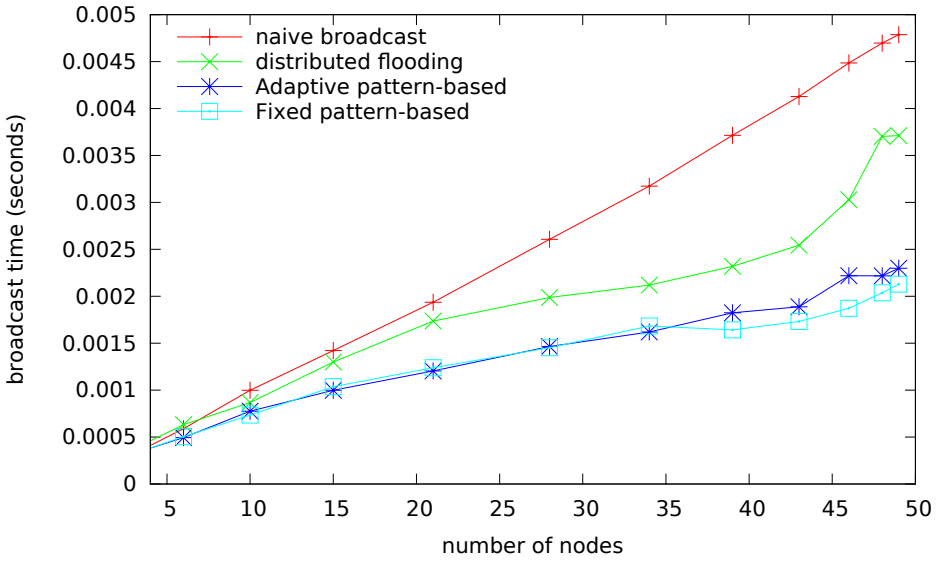


Figure 5.6: Comparison of different request propagation schemes

For our analysis, let us assume a tile size of $n \times n$. One sender needs to broadcast the message to the remaining $n^2 - 1$ recipients.

Among the different schemes, the naive broadcast scheme tends to be the most time consuming. In this scheme, a single source node sends the message to all the recipients and waits for replies from each of them. This increases the load on the single source. The number of individual end-to-end messages on the NoC equals the number of recipients of the broadcast, i.e., $n^2 - 1$. But it is important to note that, on a 2D mesh topology with X-Y dimension ordered routing, the messages are sent over the same link multiple times resulting in unnecessary link utilization. We can easily observe that as the same X-Y path is traversed multiple times, there is heavy contention on a few links that become the bottleneck.

Distributed flooding performs slightly better. In this method, the load on the single source node is reduced as all nodes contribute to forwarding the message. Also, the message is sent exactly once over each link. But the number of individual messages on the NoC is comparatively larger than that of the naive broadcast scheme. For a tile size of $n \times n$, the total number of messages equals the total number of links on the NoC, i.e., $2 * n * (n-1)$. Hence, after a threshold point, the cost of distributed flooding tends to increase and is as costly as the naive broadcast scheme. This trend was observed in Figure 5.6, when the number of nodes is greater than 43.

Fixed pattern-based propagation, where messages propagate in a predefined pattern, uses the least number of individual messages, namely $n^2 - 1$. The fixed pattern reduces the number of links used to $n^2 - 1$ and the message is sent exactly once on each link. Also, the load on

the single source node is considerably reduced as each recipient forwards the message further. Hence, pattern-based propagation consumes the least amount of time (see Figure 5.6).

In the adaptive pattern-based propagation scheme, the number of individual messages is $n^2 - 1$, which is the same as in the fixed pattern-based scheme. Also, the scheme ensures that the message is sent only once per link. Even the additional cost in setting up the adaptive pattern is amortized over multiple runs. Hence, the adaptive pattern-based scheme performs as good as the pattern-based scheme (see Figure 5.6). The adaptive pattern-based scheme is only slightly costlier than the fixed pattern-based scheme. This is explained as follows: Depending on the adaptive pattern formed, certain nodes may need to forward the message to more than one recipient (unlike the fixed pattern-based scheme). For example, nodes 2 and 5 incur this additional processing time in Figure 2.5c.

Chapter 6

Related work

There has been renewed research interest in academia and industry in redesigning operating systems for the future manycore architectures [6, 7, 9, 15, 21, 23, 29]. Among these, the most closely related work to ours is that of the Factored Operating Systems (FOS) [29] and the Barrelfish multikernel [6], which share our vision of redesigning the operating system services by embracing the distributed nature of future large-scale manycore processors. Our micro-kernel and pico-kernel abstraction is inspired by FOS [29], where application and operating system services run on physically separate cores. But our work differs from FOS in that we benefit more from spatial locality as pico-kernels (cores) only need to communicate with their parent micro-kernel. The parent micro-kernel engages in scheduling and/or other control decisions in concert with its peer micro-kernels. We show that this delegation of control (e.g., scheduling capabilities) to micro-kernels enables fast and autonomous decisions in managing the set of the cores belonging to a single micro-kernel domain. We follow the core of the design principles postulated by Peter et al. [24] for designing multi-core schedulers. We even go one level further and take a purely distributed message passing approach as the primary means of communication enabled by our adoption of NoCMsg [33] as our low-level messaging library.

The compute chip in the BlueGene/Q [10, 17] supercomputer has 16 cores for executing application tasks, one core dedicated for operating system services and one core for redundancy. This is similar to our approach of dedicated micro-kernels for operating system services and applications. Our design differs here as we propose multiple dedicated micro-kernels managing the cores in a manycore chip rather than across nodes.

Job schedulers for HPC clusters, such as the TORQUE resource manager [27], SLURM [31] and the Maui scheduler [19], use similar algorithms for resource allocation and employ backfilling algorithms to increase utilization. Though these schedulers in principle tackle similar problems, these solutions to HPC systems do not directly apply to large-scale manycore systems due to completely different communication to computation ratios, job completion deadlines and the

type of applications used.

Job co-scheduling for High-end computing (HEC) systems often use a single job submission portal [18, 20]. But such approaches using a centralized resource manager do not scale. Tang et al. [28] propose a distributed job co-scheduler for HEC systems. They propose to resolve deadlocks by yielding the resources after a predefined wait time. Though such a mechanism might be acceptable for HEC systems, this approach will tend to have the same starvation issues as the random-back off scheme on a large-scale manycore system with real-time deadlines (see Section 2.2). Our approach differs as we avoid deadlocks in job allocation and guarantee a definite completion time for the distributed job allocator.

Some NoC architectures, such as the Kalray MPPA-256 [12], have specialized support for multi-casting. The performance of our distributed job allocation protocol can vastly improve on such architectures as the job requests can be propagated fast leading to fewer cancellations in the *active cancellation* scheme. But most other NoC architectures ([3, 4]) lack hardware support for multi-casting while our efficient pattern-based request propagation schemes can be applied to them.

Chapter 7

Conclusion

We introduce PICASO, a distributed message passing system, to meet the scalability challenges of future manycore processors and demonstrate the ease and usability of such a system in managing large numbers of cores on a single chip. We study the distributed job allocation problem and propose a protocol with two policies, *active cancellation* and *sequencer-based atomic broadcast*. Both of these policies avoid fragmented allocations that would otherwise lead to deadlocks and provide guaranteed allocation loosely following a global order. Experimental results on the Tilera TilePro64 platform indicate that for sparse job allocations the *active cancellation* scheme provides lower overhead while for denser job allocations the *sequencer-based atomic broadcast* scheme provides lower overhead. The results obtained show that our distributed job allocation protocol is scalable and avoids deadlocks during the job allocation process, which confirms the hypothesis.

REFERENCES

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

[2] SCC External Architecture Specification (EAS) Revision 0.94.

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

[4] Tilera processor family. `www.tilera.com`.

[5] T. Agarwal, A. Sharma, and K. Kale. Topology-aware task mapping for reducing communication contention on large parallel machines. In *International Parallel and Distributed Processing Symposium*, April 2006.

[6] 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.

[7] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn't your os? In *HotOS*, 2009.

[8] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[9] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[10] P Boyle. The bluegene/q supercomputer. *PoS LATTICE2012*, 20, 2012.

[11] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

[12] Benot Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clment Lger, Benjamin Orgogozo, Jrme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, 18(0):1654 – 1663, 2013. 2013 International Conference on Computational Science.

[13] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[14] Yoav Etsion and Dan Tsafrir. A short survey of commercial cluster batch schedulers. *School of Computer Science and Engineering, The Hebrew University of Jerusalem*, 44221:2005–13, 2005.

[15] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, volume 99, pages 87–100, 1999.

[16] Michael R. Garey and David S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[17] Ruud A Haring, Martin Ohmacht, Thomas W Fox, Michael K Gschwind, David L Satterfield, Krishnan Sugavanam, Paul W Coteus, Philip Heidelberger, Matthias A Blumrich, Robert W Wisniewski, et al. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48–60, 2012.

[18] Eduardo Huedo, Ruben S Montero, and Ignacio M Llorente. A framework for adaptive execution in grids. *Software: Practice and Experience*, 34(7):631–651, 2004.

[19] David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the maui scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.

[20] Subramanian Kannan, Mark Roberts, Peter Mayes, Dave Brelsford, and Joseph F Skovira. Workload management with loadleveler. *IBM Redbooks*, 2:2, 2001.

[21] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanovic, and John Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. *HotPar09, Berkeley, CA*, 3:2009, 2009.

[22] M. Muller. Dark silicon and the internet. Keynote at EETimes Virtual Conference: Designing with ARM, March 2010.

[23] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[24] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *2nd Workshop on Hot Topics in Parallelism, Berkeley, CA, USA*, 2010.

[25] Robert R Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[26] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*, volume 1. MIT Press, 2 edition, 1998.

[27] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[28] Wei Tang, Narayan Desai, Venkatram Vishwanath, Daniel Buettner, and Zhiling Lan. Job coscheduling on coupled high-end computing systems. In *ICPP Workshops*, pages 317–326, 2011.

[29] 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.

[30] 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, 2007.

[31] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[32] Christopher Zimmer and Frank Mueller. Low contention mapping of real-time tasks onto tilepro 64 core processors. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 131–140. IEEE, 2012.

[33] Christopher Zimmer and Frank Mueller. Nocmsg: Scalable noc-based message passing. In *International Symposium on Cluster, Cloud and Grid Computing*, 2014.