High-Performance Communication Primitives and Data Structures on Message-Passing Manycores: Broadcast and Map

THÈSE N^{O} 6328 (2014)

PRÉSENTÉE LE 11 SEPTEMBRE 2014

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE SYSTÈMES RÉPARTIS

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Omid SHAHMIRZADI

acceptée sur proposition du jury:

Prof. B. Falsafi, président du jury
Prof. A. Schiper, directeur de thèse
Prof. P. Felber, rapporteur
Prof. R. Guerraoui, rapporteur
Prof. P. Sens, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2014

To my family ...

Abstract

The constant increase in single core frequency reached a plateau during recent years. This is due to a physical phenomenon, known as *power wall*, where the produced heat inside the chip is so high that cannot be cooled down by existing technologies. An alternative to harvest more computational power per die is to fabricate more number of cores into a single chip. Therefore *manycore* chips with more than thousand cores are expected by the end of the decade. These environments provide a high level of parallel processing power while their energy consumption is considerably lower than their multi-chip counterparts. Although shared-memory programming is the classical paradigm to program these environments, there are numerous claims that taking into account the full life cycle of software, the message-passing programming model have numerous advantages. The direct architectural consequence of applying a message-passing programming model is to support message passing between the processing entities directly in the hardware. Therefore manycore architectures with hardware support for message passing are becoming more and more visible. These platforms can be seen in two ways: (i) as a High Performance Computing (HPC) cluster programmed by highly trained scientists using Message Passing Interface (MPI) libraries; or (ii) as a mainstream computing platform requiring a global operating system to abstract away the architectural complexities from the ordinary programmer. In the first view, performance of communication primitives is an important bottleneck for MPI applications. In the second view, kernel data structures have been shown to be a limiting factor. In this thesis (i) we overview existing state-of-the-art techniques to circumvent the mentioned bottlenecks; and (ii) we study high-performance broadcast communication primitive and map data structure on modern manycore architectures, with message-passing support in hardware, in two different chapters respectively.

In one chapter, we study how to make use of the hardware features to implement an efficient broadcast primitive. We consider the Intel Single-chip Cloud Computer (SCC) as our target platform which offers the ability to move data between on-chip Message Passing Buffers (MPB) using Remote Memory Access (RMA). We propose OC-Bcast (On-Chip Broadcast), a pipelined k-ary tree algorithm tailored to exploit the parallelism provided by on-chip RMA. Experimental results show that OC-Bcast attains considerably better performance in terms of latency and throughput compared to state-of-the-art solutions. This performance improvement highlights the benefits of exploiting hardware features of the target platform: Our broadcast algorithm takes direct advantage of RMA, unlike the other broadcast solutions which are based on a higher-level send/receive interface.

In the other chapter, we study the implementation of high-throughput concurrent maps in

message-passing manycores. Partitioning and replication are the two approaches to achieve high throughput in a message-passing system. This chapter presents and compares different strongly-consistent map algorithms based on partitioning and replication. To assess the performance of these algorithms independently of architecture-specific features, we propose a communication model of message-passing manycores to express the throughput of each algorithm. The model is validated through experiments on a 36-core TILE-Gx8036 processor. Evaluations show that replication outperforms partitioning only in a narrow domain.

Keywords : High Performance, Communication Primitive, Data Structure, Message Passing, Manycore, Broadcast, Map, HPC, Operating System.

Résumé

La fréquence des cœurs de calcul a arrêté d'augmenter depuis quelques années. Ceci est lié au phénomène physique appelé "power wall", qui fait que, avec l'augmentation de la fréquence, la chaleur dissipée par le processeur devient trop importante par rapport aux capacités des systèmes de refroidissement. Pour continuer à améliorer les performances des processeurs, une alternative est alors d'augmenter le nombre de cœurs par processeur. Donc des processeurs *manycore* incluant des centaines de coeurs sont attendus pour la fin de la décennie. Ces environnements offrent un haut niveau de puissance de calcul parallèle alors que leur consommation énergétique est considérablement plus faible que celle des environnements multi-processeurs. Bien que la programmation par mémoire partagée soit le paradigme classique pour programmer ces environnements, plusieurs personnes estiment qu'en tenant compte du cycle de vie complet d'un logiciel, la programmation par échange de messages à de nombreux avantages. La conséquence directe de l'utilisation du modèle de programmation par échange de messages est l'émergence d'architectures supportant la transmission de messages entre les entités de calcul au niveau matériel. Ces plateformes peuvent être utilisées de deux manières : (i) un *cluster* de calcul haute performance programmé par des scientifiques hautement qualifiés utilisant, par exemple, une bibliothèque MPI (*Message Passing Interface*) pour les communications ; (ii) une plateforme ordinaire nécessitant un système d'exploitation permettant au programmeur de s'affranchir de la complexité matérielle. Dans le premier cas, les performances des primitives de communication sont déterminantes pour les applications MPI. Dans le second cas, ce sont les performances des structures de données du noyau qui sont un facteur limitant. Dans cette thèse, (i) nous présentons les solutions de l'état de l'art pour traiter ces deux problèmes, puis (ii) nous consacrons deux chapitres à l'étude de primitives de *broadcast* (diffusion de type *un-vers-tous*) et à l'étude de structures de données de type tableau associatif, sur des architectures *manycore* modernes fournissant un support pour l'échange de messages au niveau matériel.

Dans le premier chapitre, nous étudions comment utiliser les fonctionnalités matérielles pour mettre en œuvre la diffusion un-vers-tous de manière efficace. Nous considérons la puce *Single Chip Cloud* (SCC) d'Intel comme plateforme cible. Cette plateforme offre la possibilité de déplacer des données sur la puce entre les tampons de transmission de messages (*Message-Passing Buffer - MPB*) des différents cœurs par accès mémoire distants (*Remote Memory Acces* - RMA). Nous proposons un algorithme de diffusion en arbre pipeliné d'arité k, appelé OC-Bcast, conçu pour exploiter le parallélisme offert par les accès mémoire distants. Les résultats expérimentaux montrent qu'OC-Bcast atteint des performances nettement supérieures aux

algorithmes de l'état de l'art en termes de latence et de débit. Ces résultats mettent en évidence les avantages d'exploiter les fonctionnalités matérielles de la plateforme cible. Nos algorithmes de diffusion profitent directement des avantages des accès mémoire distants contrairement aux autres algorithmes qui sont fondés sur une interface de type émission/réception de plus haut niveau.
Dans le second chapitre, nous étudions la mise en œuvre de tableaux associatifs efficaces dans les architectures *manycore* à échange de messages. Partitionner et répliquer une structure de donnée sont les deux approches pour en améliorer le débit dans un système à échange de messages. Ce chapitre présente et compare différents algorithmes de tableau associatif avec une cohérence forte, fondés sur le partitionnement et la réplication. Pour évaluer les performances de ces algorithmes indépendamment des caractéristiques propres à une architecture spécifique, nous proposons un modèle des communications par échange de messages au sein des processeurs *manycore* pour exprimer le débit de chaque algorithme. Le modèle a été validé grâce à des expériences réalisées sur un processeur TILE-Gx8036 incluant 36 cœurs. Les résultats montrent que la réplication surpasse le partitionnement uniquement dans un petit sous-ensemble de cas.

Mots clés : Haute performance, primitive de communication, structure de données, échange de messages, *manycore*, diffusion, tableau associatif, calcul haute performance, système d'exploitation.

Acknowledgements

I am extremely grateful of my supervisor Prof. Andre Schiper as well as my colleague Thomas Ropars for their direct guide and support to write my thesis. I am also thankful to Intel and Tilera for providing us with the required infrastructure and technical support. Specifically I express my gratitude for Ted Kubaska at Intel and Chris Croft-White at Tilera for their availability whenever we encountered technical issues. I was honored to have Prof. Pascal Felber, Prof. Rachid Guerraoui and Prof. Pierre Sens as the members of the thesis jury to evaluate my thesis, and Prof. Babak Falsafi for chairing the jury. I am appreciative to all other LSR members, namely Darko Petrovic, Zarko Milosevic and Martin Beily, for their useful feedback during presentations and proofreadings. I also never forget the guides I was given by Sergio Mena and Fatemeh Borran during my first months in a research environment. And the last but not the least, I express my great appreciation for France Faille to support us whenever needed during all these years.

Omid

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Hitting the power wall prevents the semiconductor industry to improve the single core performance according to the Moore law [21]. The alternative to obtain a higher performance is to increase the level of parallelism by putting many ordinary cores on the same chip. With the same power budget, it has been shown that the aggregate performance of a chip with big number of simple cores exceeds that of a chip with small number of complex cores [20]. Therefore manycore processors featuring tens, if not hundreds, of ordinary cores communicating with a highly efficient network-on-chip (NoC) are becoming more and more available.

Taking advantage of the high degree of parallelism provided by such architectures is challenging and raises questions about the programming model to be used [99, 72]. Most existing processors are still based on cache-coherent shared memory. Designing a scalable concurrent algorithm for cache-coherent processors is a difficult task because it requires understanding the subtleties of the underlying cache coherence protocol which is not inherently scalable [27]. On the other hand, though less popular among mainstream programmers, message-passing model looks appealing because it provides the programmer with explicit control of the communication between cores which can lead to significant benefits for the full life-cycle of software [57]. However, compared to the vast literature on concurrent programming in shared-memory systems [49], message-passing programming on manycore processors is not yet a mature research topic. The natural consequence of adopting a message-passing programming model on the architecture of these platforms is to provide message-passing support in hardware, although it can be also emulated on top of a shared address space with considerable performance penalties.

Considering the low latency and high throughput of a NoC, manycore chips are very similar to parallel High Performance Computing (HPC) clusters. In order to provide inter-process communication, HPC applications take advantage of different Message Passing Interface(MPI) libraries. Performance of MPI communication primitives are a major bottleneck in HPC applications, and have been widely studied in different contexts. However, results show that porting an HPC communication library to manycore platforms requires rethinking the design of the communication primitives [83].

Apart from running scientific HPC applications, these chips can be seen as the ordinary computing platform to run general-purpose applications, where a global operating system abstracts away the architectural complexities from the mainstream programmer. Kernel data structures have been shown to be an important performance bottleneck for the operating systems designed for these environments [109, 16, 19]. To increase the performance of data structures in shared memory architectures, several well-known techniques can be used [49]. In message-passing systems, improving the performance of concurrent data structures requires fundamentally different approaches [36]. Existing studies made in distributed message-passing systems are of little help because the high performance of NoCs provides a completely different ratio between computation and communication costs compared to large scale distributed systems.

In the rest of the introduction, we discuss the motivations for applying the message-passing programming model on manycores as well as its architectural implications on these platforms. Afterwards we briefly explain the existing techniques to improve the performance of MPI communication primitives and kernel data structures on manycore environments, with different communication infrastructures. Finally we present an overview of the thesis.

1.1 Message-Passing Programming Model and Manycore Platforms

Message passing and shared memory are the two models to program parallel applications. In a message passing model, access to the shared data is managed by explicit communication with no shared address space while in a shared memory model, shared data is located on a shared address space. A study done in [57] by Intel tries to choose the right programming model to address the specific features of manycore architectures. To do this comparison, they consider full cycle of software development including writing, debugging, validating, optimizing and maintaining of the parallel program. They evaluate different strategies for designing parallel applications against different metrics, covering full life-cycle of a software development. The chosen strategies, derived from [96], cover a broad range of parallel applications: agenda parallelism, result parallelism and specialist parallelism. In agenda parallelism, an application is divided into a particular agenda of tasks and each process is assigned to pick a task from the agenda and do the tasks repeatedly until the job is done. In result parallelism, the ultimate goal of the application is to come up with a data structure as the final result and each process is assigned to produce one piece of the result. In specialist parallelism, an application is based on some logical networks of specialists and each process is assigned to perform one specific kind of work. Each strategy uses different patterns for designing parallel algorithms (See Table 1.1). Each pattern is assessed with respect to a set of concrete metrics inspired from [46], namely generality, expressiveness, viscosity, composition, validation and portability (See Table 1.2 for a brief description of each).

The message-passing programming model is claimed to be a better choice with respect to composition, validation and portability, independently of the chosen strategy for parallelism.

Parallelization Strategies	Design Patterns	Means of Parallelism
agenda parallelism	task parallelism / divide and conquer	tasks
result parallelism	geometric decomposition / data parallelism	data structures
specialist parallelism	producer consumer / event based coordination	events

Table 1.1: Different strategies to implement parallel programs

Metric	Description
composition	ability to modularize parallel programs
validation	possibility of validating program correctness
portability	easy portability of the program to different platforms
generality	ability to express any parallel algorithm
expressiveness	existence of concise concurrency abstractions
viscosity	possibility of implementing incremental changes to a working program

Table 1.2: Metrics to compare message-passing and shared-memory programming models

Parallel programs can be decomposed into different isolated modules, each working on its own private memory, interacting with each other through a well-defined communication interface. Decomposing a program in this way can be naturally expressed in a message-passing programming model. To validate a program we need to guarantee that every legal interleaving of active threads leads to a correct execution. In a message-passing programming model, the programmer only needs to consider combinations of different messaging events. In a shared-memory programming model, validating a program is shown to be an NP-complete problem [55]. Moreover message-passing programming is more portable since it imposes less constraints on the consistency model of different platforms.

Regarding other metrics, namely generality, expressiveness and viscosity, message-passing model seems to be a better option if specialist parallelism is applied. Considering the pipeline algorithms, movement of data between different levels can be naturally expressed in a message-passing model. Moreover it does not suffer from the error-prone synchronization during data movement, a phenomenon in which shared-memory programmer suffers from. Considering event-based algorithms, message-passing programmers should be careful about unpredictable flow of messages between processes. However this issue can be circumvented using a higher-level model in which flows of messages are controlled. The actor model can provide such a coordination for event-based algorithms and it is a natural fit for message-passing programming model. Implementing actor model in shared-memory needs complex synchronization hassles. On the other hand, shared-memory model is argued to be a better choice for agenda and result parallelism. For example, applying a divide and conquer pattern is a better fit for shared-memory programming model since a task is recursively divided into a number of smaller tasks where the data associated with each task must be decomposed. In message-passing programming model, where decomposition of data is explicit, it is difficult to apply when lots of tasks are created dynamically. Shared memory model avoids this problem since all threads have access to the shared data. Moreover considering data parallelism pattern, shared memory model seems more suitable since it does not require complicated

data movements during execution of collective operations.

Whatever the choice of the programming model, it puts requirements on the hardware that supports it. A programming model that requires a shared address space, in order to run efficiently, requires hardware-supported cache coherence. By increasing the number of cores, the overhead of the hardware-based cache coherency limits scalability. This is mainly due to the increase in architectural complexity, coherency traffic on the interconnect and needed storage to track the cached values. These increasing costs mean that, as the cores grow, a cache coherency protocol eventually limits the achievable performance of the parallel cores. On the other end of the spectrum, a programming model that needs message passing, in order to run efficiently, requires an architecture with hardware-based message-passing interface [1]. A message-passing manycore chip, with no support for cache coherency, can be suitable for software implemented using the message-passing model. In this case, there is no coherency barrier and these architectures can scale up to a much larger number of cores.

As you might already conclude, message-passing programming model seems attractive for both programming model and architecture layers. The benefits magnify as the number of cores increase. Following these observations, manycore architectures with programmer-accessible message passing support in hardware are becoming more and more visible. These architectures provide message passing between different cores through shared on-chip buffers. The sender acts by putting its message into the shared buffer and the receiver acts by getting those messages from the shared buffer. In some of these architectures, the programmer is in charge of providing the required synchronization between senders and receivers during their access to the shared buffer. Examples of such architectures include: Intel Polaris [50], Intel SCC [52], Calray MPPA [4] and Adapteva Epiphany [1]. In another set of architectures, this synchronization task is abstracted away from the programmer and is totally managed by the hardware. Examples of such architectures include: Tilera TILE [8], Intellasys SEAfourth [3] and Picochip DSP [7]. To the best of our knowledge, the TILE architecture from Tilera is the only case where all cores have access to a cache-coherent shared address space while they are also provided with a hardware-based message-passing interface.

1.2 High-Performance MPI Communication Primitives

Manycore environments resemble HPC clusters while occupying smaller space and consuming less energy. The performance of MPI communication primitives is a major bottleneck for HPC applications running on manycore environments [86]. We overview some of the existing techniques used to improve the performance of MPI communication primitives in these environments. We start by point-to-point communication primitives followed by collective ones. A summary of these techniques are mentioned in Table 1.3. Note that we only mention those which are specific to manycore environments, avoiding the general techniques which

[1] Ignoring faults, each of the two programming models can be implemented on top of an architecture which suits with the requirements of the other one. However this sacrifices the performance.

Communication	Techniques
point to point	avoid system calls minimize copying (at run time) minimize copying (at compile time) hardware implementation
collective	leverage kernel facilities topology awareness hardware implementation exploit hardware features and properties

Table 1.3: Existing techniques for high-performance MPI primitives on manycores

are applicable independently of the target architecture.

1.2.1 Point-to-point communication

In the absence of hardware-based message-passing interface, implementing message passing on top of shared memory is the only viable solution to benefit from the message passing programming model. This leaves a rather vast space for performance optimizations. Proposed techniques are mainly based on avoiding costly systems calls as well as minimizing the number of memory copies during message transfers. We briefly go over some of these techniques.

Avoiding system calls

The most intuitive way to implement communication between processes, located on different cores, is to use kernel facilities of the operating system. These facilities include Unix domain sockets, TCP and UDP sockets, pipes, IPC and POSIX message queues. Despite some differences, their underlying mechanisms are similar. Using all facilities, a buffer in the kernel space is shared amongst a set of senders and receivers. To send a message, a sender invokes a system call which copies the message from the user space to the kernel space and adds it to the buffer. To receive a message, a receiver invokes another system call that copies the incoming message, if any, from the kernel-space buffer to a user-space buffer.

The main drawback of the above, is the costly system calls which are invoked during each message transfer. Message-passing mechanism in the Barrelfish operating system [2] avoids these costly system calls by functioning totally in user space. For each sender/receiver pair, there are two limited size circular buffers of messages which act as unidirectional communication channels. Each message entry is composed of a header and of a content. The header contains notification flags for the purpose of synchronization, *e.g.* to inform the receiver that the message can be read. Whenever a sender wants to send a message from its private buffer, it first checks if there is enough room in the channel for a new message. If it is the case, it copies the message from its private buffer to the communication buffer. The receiver knows the location of the next entry to be read from the communication channel. To receive a message, it polls the header of the message at that location, waiting for the notification to be written.

As soon as the notification is written by the sender, the receiver copies the message from the communication buffer to its own buffer and sends an acknowledgement back to the sender. A study done in [15] compares the throughput of kernel-based techniques with message-passing mechanism implemented in Barrelfish on a shared-memory manycore architecture. As expected, the achievable throughput of message-passing mechanism in Barrelfish outperforms the kernel-based techniques, especially for the case of small messages.

Minimize copying (at run time)

Previous techniques need two memory copies to transfer a message from the sender to the receiver. Several techniques have been proposed to reduce the number of copies, during point-to-point communication, to only one. Most of these techniques rely on the memory mapping facilities of the operating system kernel. We overview some of these techniques.

A method based on Linux KNEM kenerl module [79], tries to improve the performance of MPI communication on manycores by minimizing the number of memory copies. The main trick roots in the kernel's ability to access the memory of all user-space processes beyond its address space boundary. Therefore inter-core communication via the kernel thread can be done using only one memory copy. Although this method exhibits a better throughput compared to the techniques based on an intermediate shared-memory buffer, its small-message latency suffers from the context switch overhead between user and kernel threads.

ZIMP (Zero-copy Inter-core Message Passing) [15] is an efficient inter-core communication mechanism for manycores functioning totally in user space which, unlike message-passing mechanism in Barrelfish, provides a zero-copy send primitive by allocating messages directly in the shared communication channels. To send a message, a sender first gets the address of the next available entry in the channel. Afterwards it waits for that entry to become free, *i.e.* all receivers have read the previous message at this entry, by polling on a flag. When it is the case, the sender writes the content of the message in that entry followed by updating the synchronization flags. To receive a message, a receiver first gets the index of the next message that can be read. If there is a message to be read, it updates the index of the next message to be read, copies the content of that entry and resets a flag indicating that the message is read. Comparison of the message-passing mechanisms of ZIMP and Barrelfish shows an order of magnitude throughput improvement in favor of ZIMP.

In [40], the authors have implement a Hybrid MPI (HMPI) library, as an abstraction layer on top of existing MPI libraries, to investigate single-copy message-passing techniques on manycores. To implement message passing on top of shared memory, memory used for communication is mapped to the same virtual address on every process. They implement two incoming message queues per receiver. The first queue is globally accessible by all processes. Senders write messages to this global queue which is owned by the destination process. The second queue is private. When a receiver attempts to match incoming messages to its local receives, it empties its global queue and adds incoming messages to its private queue. However processes have to

map the memory region of other processes to their own address space, which means that the total size of the page tables can become large due to excessive mappings.

XPMEM [9] is a Linux kernel module that enables any process to map the memory space of another process into its own virtual address space. After a process maps the memory region of another process to its own address space, it can access that memory region using a single memory copy operation. Similarly to HMPI, this mechanism suffers from a high number of mappings and growing size of the page tables.

SMARTMAP [24] is another strategy based on address space mapping which is implemented in some lightweight kernels. Similar to XPMEM, SMARTMAP enables a process to map the memory of another process into its address space. On SMARTMAP, unlike XPMEM, page table entries that are used for a remote address mapping are shared. A process can access memory of another process by updating the corresponding shared entries. Therefore the total size of the page tables does not grow as in the previous cases. However implementing SMARTMAP in the Linux kernel is hard since the memory management of the Linux does not allow shared page table entries.

PGAS (partitioned global address space) programming model [110] is becoming popular as a suitable programming approach for manycore architectures. In PGAS applications, global arrays are partitioned amongst participating processes. Access to remote part of a global array, located on a remote process, takes place through inter-core communication. Most PGAS languages currently use some shared-memory solutions to implement such communication. However these solutions lead to either two memory copy operations in case of applying a shared buffer or large page tables in case of applying a memory mapping schema. In [93], a new process model named PVAS (partitioned virtual address space), provides a solution that implements single-copy inter-core communication for PGAS languages without suffering from the page table size issue. The main idea is to allow a PGAS process to access the memory region of another process directly, by eliminating the address space boundary between them.

Minimize copying (at compile time)

The study in [25] presents a compiler-based optimization for MPI applications that directly transfer application data structures from senders to receivers without paying the cost of message serialization and deserialization on a shared-memory architecture. It exploits the fact that the code to serialize and deserialize data structures typically does a simple iteration through the communication buffer. This makes it possible to match the serialization writes in the sender's source code with the deserialization reads in the receiver's source code. Therefore a data structure can be transferred between senders and receivers using direct memory access.

Hardware implementation

Implementing message passing on shared-memory architectures requires processor intervention for the data movement and synchronization. By implementing these functions in hardware, cores are freed from having to interfere with message transfers, which allows them to perform some other useful work. Pronto message-passing system [58] provides a DMA-based mechanism for message transfers which performs buffer management and message synchronization directly in hardware. Messages are moved between local memories of each tile through hardware-managed message-passing buffers. This shows an advantage over message-passing architectures with one-way communication primitives, *e.g.* Intel SCC, where message synchronization and buffer management are entirely done by the programmer. Moreover modern manycore chips, *e.g.* Tilera TILE series, provide a mechanism similar in the sense that a high-performance message-passing system between cores is entirely implemented in hardware. Therefore the programmer does not need to deal with message synchronization and buffer management issues of inter-core communication.

1.2.2 Collective communication

A significant performance overhead of the HPC applications is caused by collective communication, which involves several tasks in one communication. Profiling study in [87] shows that MPI applications spend more than eighty percent of their communication time in collective operations. Improving performance of point-to-point communication indirectly improves the performance of collective operations that are implemented on top of them. However several proposed techniques directly improve the performance of collective operations. We briefly present some of these techniques including: taking advantage of kernel facilities, taking into account topology of on-chip network, direct implementation of collective primitives in hardware and exploiting architectural features and properties during the design phase.

Leverage kernel facilities

Leveraging kernel facilities have shown to be beneficial for the performance of MPI collectives on manycores. In [68], the author implements a high-performance broadcast operation on a shared memory architecture, as an example of MPI collective operations, utilizing the KNEM kernel facility. As mentioned earlier, KNEM is a Linux kernel module that enables inter-core communication with only a single memory copy. Recent versions of this module support multiple processes being able to read or write to the communication buffer simultaneously. Utilizing this module not only decreases the on-chip traffic by reducing the number of memory copies, but also results in higher level of parallelism in designing collective operations. Experiments show a significant performance improvement of kernel-assisted collectives compared to the existing state-of-the-art MPI implementations.

Topology awareness

With the increasing number of computing cores and memory hierarchies integrated into a single chip, the distribution of participating MPI processes inside a chip becomes more critical for the performance of collective operations. As the first try to leverage on-chip topology information to improve performance of collective operations, [69] proposes an automated framework for MPI libraries to detect and take advantage of the process distances during runtime. Based on runtime process distance information, the MPI library constructs an adaptive communication topology for each collective operation. These topology-aware operations provide optimal performance for a given placement of participating processes. This automated optimization approach at the level of the MPI library can complement the clever process placement approaches mentioned in the context of HPC clusters [29, 53].

Hardware implementation

Hardware support can prevent collective communication from becoming a system bottleneck. Authors in [108] propose some hardware features to deal with the communication overheads of on-chip message passing. The main idea is to offload the application from computation-intensive tasks such as collective operations in a transparent way. They propose an interface between processing cores and the on-chip network using a hardware module called Small Network Adapter. Using this interface, they offload the basic mechanism of point-to-point message passing to the hardware, which is leveraged subsequently to implement collective operations. Moreover some recent manycore products, *e.g.* Kalray MPPA [4], Adapteva Epiphany [1] and Picochip DSP [7], provide hardware support for multicast operation with a programmer-accessible interface. Such a support can achieve substantial throughput improvements and power savings, since cores are not involved in the actual transfer of messages. Numerous works investigate implementation of other collective protocols. An an example authors in [67] propose a hardware mechanism for reduction operations. However it targets hardware protocols, *e.g.* cache coherency, and provides no software interface. We are not aware of a manycore chip that provides hardware support for other MPI collectives.

Exploiting hardware features and properties

Implementations of MPI collectives take advantage of shared-memory manycore architectures in the following ways: (i) collectives are built on top of point-to-point message-passing, which uses shared memory as its transport layer; or (ii) collectives are implemented directly on top of the shared memory, where processes can copy a message into the shared memory space so that all communicating processes can have access to it. This feature can reduce the number of memory transfers in collective operations which leads to a better performance [44, 71]. Moreover as the number of cores per chip grows, cores exhibit more and more non-uniform memory access (NUMA) behaviour. A work done in [61], takes the NUMA property into account for better implementation of MPI collectives.

Techniques	Flavours
improve synchronization	increase parallel access (fine-grain locks / RW locks / RCU) reduce memory contention (exponential backoff / queue locks) reduce remote memory referrences (server-based / combining) lock-free synchronization
minimize shared data	address range / shares / kernel cores
avoid shared data	replication / partitioning
relaxing consistency	

Table 1.4: Existing techniques for high-performance kernel data structures on manycores

1.3 High-Performance Kernel Data Structures

Manycore architectures are becoming accessible to main-stream programmers. This introduces important challenges for operating systems designed for these environments in terms of scalability when the number of cores increases [16, 109, 22, 65, 23]. Some studies reveal that poor scalability of some operating system services can dominate application performance [43, 105]. An important source for poor scalability of such services is the use of concurrent kernel data structures, which are accessed by multiple cores at the same time. In this section, we discuss some of the state-of-the-art techniques proposed to improve the performance of data structures in manycore environments. Note that we limit ourselves to general techniques that are applicable to all data structures, and avoid techniques that target a specific data structure. Performance improvement techniques can be classified into the following categories: (i) improving synchronization; (ii) minimizing shared data; (iii) avoiding shared data; and (iv) relaxing consistency. We overview different flavours of each category, which are summarized in Table 1.4, throughout this section.

1.3.1 Improve synchronization

The usual technique to implement high-performance concurrent data structures in shared-memory manycore architectures is to improve the synchronization methods which is used for controlling mutual access to the shared data. This can be done using different approaches: (1) by increasing the parallel access to the data structure; (2) by reducing the contention on the cache lines; (3) by delegating the task of synchronization to another set of cores; and (4) by applying lock free synchronization provided that hardware support is available. Some of these techniques are extensively used in the Linux operating system [5] and its proposed extension to support manycores [22]. We go briefly through each approach.

Increase parallel access

The first version of Linux kernel with multi threading support, applied a single lock to protect critical kernel data structures. Soon it became a major performance bottleneck of the kernel and was subsequently replaced by fine grain locking [100]. Fine grain locking is a mechanism

used to break a single lock into smaller locks where each of them is responsible to protect a single portion of the shared data structure. Operations on the data structure are required to obtain one or more of these locks in order to read or write the corresponding portion of the data structure. Fine-grain locking is able to improve the performance of the operating system due to its ability to let more operations proceed in parallel.

A similar benefit can be provided by having locks that allow multiple concurrent readers, known as reader-writer locks [32]. They allow reader threads to access the shared data concurrently, but exclusively from the writer threads. However depending on the implementation, either readers or writers might face with starvation if other threads keep performing the opposite operation. Therefore variants of these locks with different fairness properties between reader and writer threads are also proposed.

The read-copy-update (RCU) algorithm [75] is a special form of reader-writer locks which is used in Linux kernel. In contrast with conventional locking primitives that provide mutual exclusion among concurrent threads, no matter whether they are readers or writers, or with reader-writer locks that allow concurrent reads but not in the presence of writes, RCU supports a single writer and multiple readers to occur concurrently. This property ensures unconditional progress for read operations.

Reduce memory contention

An important aspect of designing a lock, is to come up with a strategy if trying to acquire a taken lock fails. In a unicore machine, the common solution is to give the core to another thread. However in the case of more than one cores inside a single machine, trying repeatedly to acquire a lock is needed since the lock can be released at any time by a thread which is executing in another core. Spinlocks are made based on this technique. Spinning threads can also be scheduled to get blocked, but this makes sense if the scheduling overhead does not exceed the spinning overhead and cores have something else to do. However spinning on a single synchronization variable can be a severe performance bottleneck, since it can introduce a high memory contention and interconnect traffic.

A solution to deal with pitfalls of spinning is to apply a technique known as exponential backoff [10]. Using this solution, a thread with unsuccessful spinning attempts, waits for a while before trying again. The waiting time grows with the number of failed attempts. This leads to less memory contention and interconnect traffic due to the less unsuccessful attempts.

Exponential backoff can lead to a situation where the lock is free, while all threads trying to acquire it have been delayed and none of them can make progress. A way to avoid such a scenario is to create a logical queue of competing threads so that a lock, upon its release, can be owned by the next waiting thread. Each queue thread can have a flag to inform the next thread to get the lock upon its release. To obtain a lock, a thread adds itself atomically to the tail of the waiting queue. Afterwards it spins on the flag of its predecessor to know whether it

can obtain the lock. Note that each thread spins on its local cacheline in a cache-coherent architecture. Some variants of queue locks implement the queue using an array [12, 45], while other variants implement the queue using a list [33, 70, 76]. Several reader-writer locks are also proposed, applying a similar queue-based technique [77, 56, 91].

Reduce remote memory references

The number of Remote Memory References (RMR) during a synchronization protocol is an important performance bottleneck in a cache-coherent architecture, since accessing memory is an order of magnitude more expensive than accessing local caches. The previous techniques based on queue-locks require a constant number of RMRs to acquire a lock, thanks to the local spinning. However other solutions try to further reduce the number of RMRs. The key idea of these solutions is to delegate execution of the critical section to the core where shared data is located. Two main proposed approaches to achieve this goal include: the server-based approach [66, 31], and the combining approach [48, 38, 39].

In server-based approach, clients send operations to a dedicated server, that contains the shared data structure, to execute them on their behalf. The shared data structure remains in the cache of the server, since it is the only entity that accesses the data structure. Therefore the only possible RMRs during execution of a critical section, are related to the communication between the clients and the server. This simple approach is very efficient when a small number of critical sections are highly contended [66].

An approach based on combining does not require dedicated servers. A so called combiner thread which holds a lock on a shared data structure, executes operations of other threads on the critical section in addition to its own. In order to prevent combiner from being starved, if the number of requests are high, its role changes among different threads over time. Similar to server-based approach, possible RMRs only happen during the communication between the combiner and other threads. Despite complexities involved in synchronizing threads, this approach prevents wasting of CPU cycles in case of no pending critical section requests.

As a further optimization, in [82] authors take advantage of hardware-based message passing to perform the communication between clients and server/combiner. Their message-passing variants of server-based and combining approaches show a considerable gain of performance compared to their shared-memory counterparts, specially in case of small critical sections. Their results clearly shows benefits of using hardware message passing, which leads to decreased number of RMRs during communication between clients and the entity who executes critical section operations.

Lock-free synchronization

Using locks to provide mutual exclusion inherently suffers from several liveness issues including: deadlock, preemption or interruption of the lock holder, priority inversion and convoying.

Moreover there should be a strategy to release the shared resources in case the lock holder fails. An alternative to avoid these issues is to provide synchronization without locks, a method which is known as lock-free synchronization. Concurrent access to a lock-free data structure guarantees that some thread makes progress independently of the behaviour of other threads. However lock-free synchronization requires support of special atomic operations in hardware, such as atomic swap, test-and-set, fetch-and-add, compare-and-swap and load-link/store-conditional. Lock-free synchronization can provide better performance without suffering from liveness issues of locks, although contention and starvation are still a possibility. The difficulty of applying this approach comes from the lack of a general recipe to design a lock-free data structure, hence each data structure should be studied on its own [49]. The authors of [73] propose a multicore operating system kernel which is implemented based on lock-free data structures.

1.3.2 Minimize shared data

To deal with the scalability challenges of traditional operating systems on cache-coherent manycores with respect to kernel data structures, the Corey operating system [23] proposes a new policy: the kernel assumes each data structure is modifiable by only one core, unless applications request a different policy. In this way applications are in charge of controlling sharing of data structures. The application is the entity that has enough information to make sharing decisions. This can include operating system services, application-level libraries and user-level applications. Therefore the operating system pays the sharing costs (*e.g.* cache misses) only when the application logic finds it necessary. To achieve this goal, they introduced three operating system abstractions: address ranges, shares and kernel cores. We introduce each abstraction briefly.

Address ranges allow applications to decide which portions of the address space is private to each core and which are shared amongst all. Accessing private regions does not suffer from contention and invalidations of TLB on other cores. Declaration of shared regions allows sharing of hardware page tables. This reduces the number of page faults, which can happen when a core references pages that are present in physical memory but are not mapped in the hardware page table. Shares are lookup tables for kernel data structures that allow applications to control which data structures are visible to different cores. Finally kernel cores are dedicated cores that are asked by applications to run a specific kernel task. Kernel cores avoid contention on the data that are used by their specified function.

1.3.3 Avoid shared data

Recent operating system prototypes targeting architectures with a very large number of cores, consider a fundamentally different approach to implement kernel data structures. They look at the operating system as a distributed system of functional units, communicating explicitly using message passing. Implementation of kernel data structures avoids sharing to provide

scalability through different approaches: while some consider replication of a shared data structure on client cores, others consider a set of servers to provide the functionality of a shared data structure.

Barrelfish [16] considers the operating system as a distributed system of cores, communicating with each other through explicit message passing. Unlike kernel data structures in traditional operating systems, which are shared and protected by locks, kernel data structures in Barrelfish are replicated across cores. Therefore any potentially shared data structure is considered as if it is a local replica. Consistency amongst replicas is maintained by exchanging messages. Their claim to improve scalability by replication comes from reducing the interconnect traffic, memory contention, synchronization overhead and access latencies.

FOS [109] and Tessellation [65] operating systems consider a set of servers to provide functionality of kernel data structures to applications through message-passing requests. Kernel data structures can be replicated or partitioned amongst servers to further improve the performance. This architecture behaves similarly to the internet servers, which allows them to scale up to a large number of machines.

1.3.4 Relaxing consistency

All previous approaches try to improve the implementation of a kernel data structure to achieve a better performance. However performance can also be improved through relaxing the semantics of a data structure. As the number of cores grows, similarly to the internet services, this relaxation can be more beneficial. Although the concurrent data structures designed for shared memory architectures mostly ensure linearizability [49], recent manycore operating systems might tolerate kernel data structures with weaker consistency criteria. As an example, an implementation of a replicated naming service for the FOS operating system ensures eventual consistency [18]. In [92], authors claim that relaxation of consistency criteria is a necessary step towards providing scalable data structures for future manycores and propose a concurrent quiescent-consistent stack as their proof of concept. However relaxing the consistency semantics of different data structures on manycore architectures is not yet a well studied topic.

1.4 Thesis Overview

In this thesis, we consider a message-passing programming model on top of manycore architectures with programmer-accessible message-passing support in hardware. We study important performance bottlenecks of HPC applications and operating systems designed for these environments, *i.e.* MPI communication primitives and kernel data structures respectively. More specifically, we study high-performance MPI communication primitives, considering the case of broadcast, as well as high-performance kernel data structures, considering the case of a map, in two different chapters. We overview the context and contributions

of each chapter.

1.4.1 High-performance broadcast

High Performance Computing (HPC) is defined as employing aggregating computing power to deliver a much higher performance than one can obtain from a typical workstation in order to solve computationally-intensive problems in science, engineering, and business. Single Program Multiple Data (SPMD) programming, where multiple independent processors simultaneously execute the same program in parallel, is a popular programming technique for implementation of HPC applications. The Message Passing Interface (MPI) [80] is the *de facto* standard for programming SPMD HPC applications. MPI defines a set of primitives for point-to-point as well as collective communication, *i.e.* operations involving more than two parties, between processes. Performance of collective operations have been shown to be an important bottleneck for MPI applications [86].

Architecture of recent manycore chips, *e.g.* the low latency and high throughput of a NoC as well as lack of cache coherency between the cores, makes them very similar to parallel HPC clusters. However they are fabricated inside a much smaller space while consuming much less energy. In order to provide inter-process communication, manycore HPC applications take advantage of different MPI libraries which are ported to these environments. However, porting of MPI communication libraries to a specific manycore platform, without tailoring the design of communication primitives to the underlying architecture, can lead to non-optimal performance.

The Intel Single-Chip Cloud Computer (SCC) is an example of a message-passing manycore chip [52] that resembles an HPC cluster. It integrates 24 2-core tiles on a single chip connected by a high-performance 2D-mesh NoC. Each tile has its own private memory, hence there is no coherency among the caches of different cores. It is provided with on-chip low-latency memory buffers, called *Message Passing Buffers* (MPB), physically distributed across the tiles. *Remote Memory Access* (RMA) to these MPBs allows fast inter-core communication using one-sided *put* and *get* primitives. Several works study the implementation of point-to-point communications on the Intel SCC, but only little attention has been paid to the implementation of collective operations.

In this chapter, we investigate a high-performance implementation for the most useful MPI collective operation, *i.e.* broadcast, on a message-passing manycore chip, *i.e.* the Intel SCC. The broadcast operation allows one process to send a message to all processes in the application. Considering the SPMD paradigm, *e.g.* MPI applications, the broadcast operation is executed by having all processes in the application call the communication function with matching arguments: the sender calls the *broadcast* function with the message to broadcast, while the receiver processes call it to specify the reception buffer. We focus on understanding how to exploit the on-chip RMA-based communication to come up with a high-performance broadcast algorithm.

Contributions of this chapter include:

- Identifying contention sources on the Intel SCC
- Coming up with a new contention-aware broadcast algorithm based on on-chip RMA
- Evaluating the new algorithm against existing solutions and confirm its significant gains

1.4.2 High-performance map

Manycore chips can be seen as ordinary computing platforms to run the general-purpose applications, where a global operating system abstracts away the architectural complexities from the mainstream programmer. Traditional kernel data structures, located on a shared address space, have been shown to be an important performance bottleneck for the operating systems designed for these environments [23]. The main performance penalties come from contention on the locks as well as unnecessary costs of cache coherency. These costs can increase linearly with the grow in number of cores.

In order to avoid the scalability issues of shared kernel data structures, several manycore operating systems prototypes [16, 109, 65] consider operating systems as a distributed system where different entities communicate with each other using explicit message passing, therefore avoiding sharing data in a shared address space. In a message-passing system, partitioning and replication are the two main approaches to improve the throughput of concurrent data structures [36]. Using partitioning, a data structure is partitioned among a set of servers that answer clients requests. Using replication, each client has a local copy of data structure in its private memory and replicas maintain their consistency by exchanging messages amongst themselves. Both strategies have been considered in recent message-passing operating systems for manycores, but performance comparisons are lacking.

Among different data structures, maps are heavily used in many systems including operating system kernels [60]. Their performance is often crucial to the operating systems and have been shown to be an important performance bottleneck [16, 109]. Implementation of a map can benefit from both partitioning and replication: since operations on different keys are independent, maps are easily partitionable [19]; and because a large majority of operations are usually lookup operations [14], replication can help handling a large number of local lookup requests concurrently.

In this chapter, we present a performance comparison of replication and partitioning for the implementation of strongly-consistent concurrent maps in message-passing manycores. Note that existing studies made in distributed message-passing systems are only of little help because the high performance of NoCs provides a completely different ratio between computation and communication costs compared to large scale distributed systems.

Contributions of this chapter include:

- Devising different strongly-consistent concurrent map algorithms to represent the design space of partitioning and replication
- Coming up with a performance model to be able to compare different algorithms independently of their underlying architecture
- Evaluating our algorithms using our model under different assumptions and settings and showing that, under strong consistency, replication can outperform partitioning only in a narrow domain

2 High-Performance Broadcast

Publication : *D. Petrovic, O. Shahmirzadi, T. Ropars, and A. Schiper. High-Performance RMA-Based Broadcast on the Intel SCC. In 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Pittsburg, PA, USA, June 2012.*

Manycore environments could be seen as HPC clusters, providing a high degree of parallelism with much lower energy consumption. Performance of HPC applications is directly affected by efficiency of collective operations, such as broadcast. The Intel Single-Chip Cloud Computer (SCC) is a prototype of a message-passing manycore chip. It offers the ability to move data between on-chip Message Passing Buffers (MPB) using Remote Memory Access (RMA). In this chapter, we study how to make use of the MPBs to implement an efficient broadcast algorithm for the SCC. We propose *OC-Bcast* (*On-Chip Broadcast*), a pipelined k-ary tree algorithm tailored to exploit the parallelism provided by on-chip RMA. Experimental results show that, compared with the state-of-the-art solutions, OC-Bcast attains almost three times better throughput, and improves latency by at least 27%. These performance gains highlight the benefits of exploiting hardware features of the target platform: Our broadcast algorithm take direct advantage of RMA, unlike the other broadcast algorithms based on a higher-level send/receive interface.

2.1 Introduction

The Intel Single-Chip Cloud Computer (SCC) is an example of a message-passing manycore chip [52]. The SCC integrates 24 2-core tiles on a single chip connected by a 2D-mesh NoC. It is provided with on-chip low-latency memory buffers, called *Message Passing Buffers* (MPB), physically distributed across the tiles. *Remote Memory Access* (RMA) to these MPBs allows fast inter-core communication.

The natural choice to program a high-performance message-passing system is to use Single Program Multiple Data (SPMD) algorithms. The Message Passing Interface (MPI) [80] is the *de facto* standard for programming SPMD HPC applications. MPI defines a set of primitives for

point-to-point communication, and also defines a set of collective operations, *i.e.* operations involving a group of processes. Several works study the implementation of point-to-point communications on the Intel SCC [104, 90, 85], but only little attention has been paid to the implementation of collective operations. This chapter studies the implementation of collective operations for the Intel SCC. It focuses on the broadcast primitive (*one-to-all*), with the aim of understanding how to efficiently leverage on-chip RMA-based communication. Note that the need for efficient collective operations for manycore systems, especially the need for efficient broadcast, goes far beyond the scope of MPI applications, and is of general interest in these systems [99].

We are investigating the implementation of efficient broadcast algorithms for a message-passing manycore chip, such as the Intel SCC. The broadcast operation allows one process to send a message to all processes in the application. Considering the SPMD paradigm, *e.g.* MPI applications, the broadcast operation is executed by having all processes in the application call the communication function with matching arguments: the sender calls the *broadcast* function with the message to broadcast, while the receiver processes call it to specify the reception buffer.

To take advantage of on-chip RMA, we propose OC-Bcast (*On-Chip Broadcast*), a pipelined k-ary tree algorithm based on one-sided communication: k processes get the message in parallel from their parent to obtain a high degree of parallelism. The degree of the tree is chosen to avoid contention on the MPBs. To provide efficient synchronization between a process and its children in the tree, we introduce an additional binary notification tree. Double buffering is used to further improve the throughput.

We confirm the gains of our broadcast algorithms through experiments. The comparison of OC-Bcast with the RCCE_comm binomial tree and *scatter-allgather* algorithms based on two-sided communication shows that: (i) our algorithm has at least 27% lower latency than the binomial tree algorithm; (ii) it has almost 3 times higher peak throughput than the *scatter-allgather* algorithm. These results clearly show that collective operations for message-passing manycore chips should be based on one-sided communication in order to fully exploit the hardware resources. The main reason is that OC-Bcast reduces the amount of data moved between the off-chip memory and the MPBs on the critical path.

To sum up, contributions of this chapter include:

- *Identifying contention sources on the Intel SCC*: we identify three possible sources of contention on this platform, which include the NoC mesh, the off-chip memory and the MPBs. Our evaluations show that at the current scale, excessive load on the network links and on the off-chip memory do not degrade the performance. However evaluations show that more than a certain number of cores accessing the same MPB at the same time can create measurable contention. In our algorithms, we take into account this property to limit the number of cores who access the same MPB simultaneously.

- *Coming up with a new contention-aware broadcast algorithm based on on-chip RMA*: to exploit the on-chip RMA, we propose a pipelined k-ary tree algorithm based on one-sided communication where k processes get the message in parallel from their parent's MPB to obtain a high degree of parallelism. The degree of the tree is chosen to avoid contention on the MPBs. To provide efficient synchronization between a process and its children in the tree, we introduce an additional binary notification tree. Double buffering technique is added to further improve the throughput.
- *Evaluating the new algorithm against existing solutions and confirm its significants gains*: to confirm the benefits of our algorithm, we compare its latency and throughput against the best existing solutions through experiments. Our results show that our algorithm has at least 27% lower latency as well as almost 3 times higher peak throughput than the state-of-the-art solutions. These results clearly show that design of collective operations for message-passing manycore chips should take into account the specific hardware features of the target architecture to achieve optimal performance.

This chapter is structured as follows. In Section 2.2 we describe the architecture and the communication features of our testbed architecture. Section 2.3 presents assumptions and goal of this chapter. Section 2.4 is devoted to our RMA-based broadcast algorithm. Experimental evaluations on our testbed architecture are presented in Section 2.5. Finally related work are discussed in 2.6.

2.2 The Intel SCC

The SCC is a general-purpose manycore prototype developed by Intel Labs. We consider this platform as the testbed of our studies in this part. In this section we describe the SCC architecture and inter-core communication.

2.2.1 Architecture

The cores and the NoC of the SCC are depicted in Figure 2.1. There are 48 Pentium P54C cores, grouped into 24 tiles (2 cores per tile) and connected through a 2D mesh NoC. Tiles are numbered from (0,0) to (5,3). Each tile is connected to a router. The NoC uses high-throughput, low-latency links and deterministic virtual cut-through X-Y routing [54]. Memory components are divided into (i) message passing buffers (MPB), (ii) L1 and L2 caches, as well as (iii) off-chip private memories. Each tile has a small (16KB) on-chip MPB equally divided between the two cores. The MPBs allow on-chip inter-core communication using RMA: each core is able to read and write in the MPB of all other cores. There is no hardware cache coherence for the L1 and L2 caches. By default, each core has access to a private off-chip memory through one of the four memory controllers, denoted by *MC* in Figure 2.1. The off-chip memory is physically shared, so it is possible to provide portions of shared memory by changing the default configuration. However we view the SCC as a pure message-passing platform. In

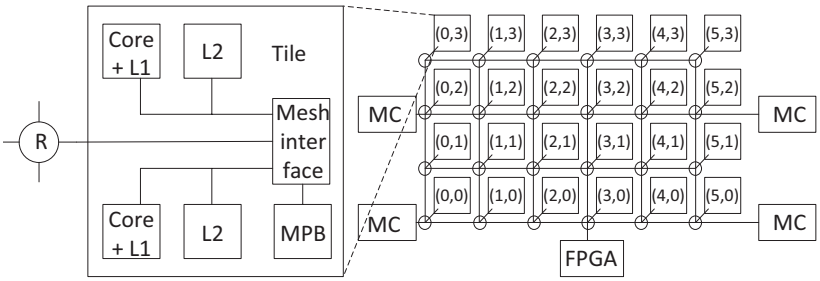


Figure 2.1: SCC architecture

addition, an external programmable off-chip component (FPGA) is provided to add new hardware features to the prototype.

2.2.2 Inter-core communication

To leverage on-chip RMA, cores can transfer data using the one-sided *put* and *get* primitives provided by the RCCE library [102]. Using *put*, a core (a) reads a certain amount of data from its own MPB or its private off-chip memory and (b) writes it to some MPB. Using *get*, a core (a) reads a certain amount of data from some MPB and (b) writes it to its own MPB or its private off-chip memory. The unit of data transmission is the cache line, equal to 32 bytes. If the data is larger than one cache line, it is sequentially transferred in cache-line-sized packets. During a remote read/write operation, each packet traverses all routers on the way from the source to the destination. The local MPB is accessed directly or through the local router[1]. Cores are also able to notify each other using inter-process interrupts (IPI).

2.3 Assumptions and Goal

The study assumes a fault-free manycore architecture where a large set of single-threaded cores are connected through a network on chip. We assume that each core executes a single thread and that threads do not migrate between cores. Cores have their own on-chip private memory and can only communicate through message passing. Communication channels are asynchronous and FIFO. Messages are composed of a set of words and can have various size.

Two one-way communication primitives are available to transfer messages: put and get. Each core is able to have remote memory access (RMA) to on-chip message-passing buffer of other cores using this two operations. Operation *'put src* $\longrightarrow$ *dest'* writes *src* (local memory or local MPB) to the *dest* (remote MPB) and operation *'get dest* $\longleftarrow$ *src'* writes *src* (remote MPB) to the *dest* (local memory or local MPB).

This chapter studies high-performance broadcast primitive in SPMD programs. We consider

[1] Direct access to the local MPB is discouraged because of a bug in the SCC hardware.

latency and throughput as our performance metrics. The source and destinations participate in broadcast by calling the broadcast function *broadcast(msg,root)*, where *root* is the source process and *msg* is the the message buffer containing the original message on the root and the received message on the destinations. Therefore the root is known to all destinations.

2.4 RMA-based Broadcast

To simplify the presentation, we assume first that messages to be broadcast fit in the MPB. This assumption is later removed. The core idea of the algorithm is to take advantage of the parallelism that can be provided by the RMA operations. When a core c wants to send message msg to a set of cores $cSet$, it *puts* msg in its local MPB, so that all the cores in $cSet$ can *get* the data from there. If all *gets* are issued in parallel, this can dramatically reduce the latency of the operation compared to a solution where, for instance, the sender c would *put* msg sequentially in the MPB of each core in $cSet$. However, having all cores in $cSet$ executing *get* in parallel may lead to contention. To avoid contention, we limit the number of parallel *get* operations to some number k, and base our broadcast algorithm on a k-ary tree; the core broadcasting a message is the root of this tree. In the tree, each core is in charge of providing the data to its k children: the k children *get* the data in parallel from the MPB of their parent.

Note that the k children need to be notified that a message is available in their parent's MPB. This is done using a flag in the MPB of each of the k children. The flag, called *notifyFlag*, is set by the parent using *put* once the message is available in the parent's MPB. Setting a flag involves writing a very small amount of data to remote MPBs, but nevertheless sequential notification could impair performance especially if k is large. Thus, instead of having a parent setting the flag of its k children sequentially, we introduce a binary tree for notification to increase the parallelism. This choice is not arbitrary: It can be shown analytically that a binary tree provides the lowest notification latency, when compared to trees of higher output degrees. Figure 2.2 illustrates the k-ary tree used for message propagation, and the binary trees used for notification. C_0 is the root of the message propagation tree; the subtree with root C_1 is shown. Node C_0 notifies its children using the binary notification tree shown at the right of Figure 2.2. Node C_1 notifies its children using the binary notification tree, as depicted at the bottom of Figure 2.2.

Apart from the *notifyFlag* used to inform the children about message availability in their parent's MPB, another flag is needed to notify the parent that the children have got the message (in order to free the MPB). For this we use k flags in the parent MPB, called $doneFlag$, each set by one of the k children.

To summarize, considering the general case of an intermediate core, *i.e.*, the core that is neither the root nor a leaf, a core is performing the following steps. Once it has been notified that a new chunk is available in the MPB of its parent C_s: (i) it notifies its children, if any, in the notification tree of C_s; (ii) it gets the chunk in its own MPB; (iii) it sets its $doneFlag$ in the MPB of C_s; (iv) it notifies its children in its own notification tree, if any; (v) it gets the chunk

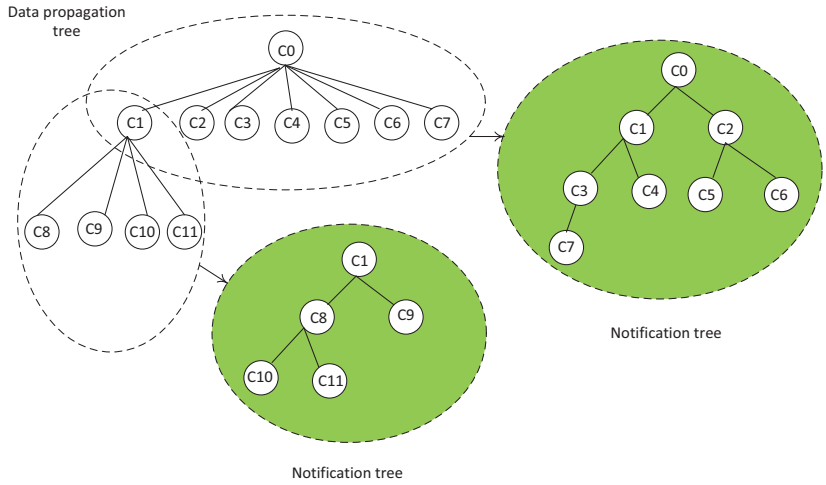


Figure 2.2: k-ary message propagation tree ($k = 7$) and binary notification trees

from its MPB to its off-chip private memory.

Finding an efficient k-ary tree taking into account the topology of the NoC is a complex problem [17] and it is orthogonal to the design of OC-Bcast. It is outside the scope of this chapter since our goal is to show the advantage of using RMA to implement broadcast. In the rest of this chapter, we assume that the tree is built using a simple algorithm based on the core ids: Assuming that s is the id of the root and P the total number of processes, the children of core i are the cores with ids ranging from $(s + ik + 1)\ mod\ P$ to $(s + (i + 1)k)\ mod\ P$. Figure 2.2 shows the tree obtained for $s = 0$, $P = 12$ and $k = 7$.

Broadcasting a message larger than an MPB can easily be handled by decomposing the large message in chunks of MPB size, and broadcasting these chunks one after the other. This can be done using pipelining along the propagation tree, from the root to the leaves.

We can further improve the efficiency of the algorithm (throughput and latency) by using a double-buffering technique, similar to the one used for point-to-point communication in the iRCCE library [30]. Up to now, we have considered messages split into chunks of MPB size,[2] which allows an MPB buffer to store only one message chunk. With double-buffering, messages are split into chunks of half the MPB size, which allows an MPB buffer to store two message chunks. The benefit of double-buffering is easy to understand. Consider message msg split into chunks ck_1 to ck_n being copied from the MPB buffer of core c to the MPB buffer of core c'. Without double buffering, core c copies ck_i to its MPB in a step r; core c' gets ck_i in step $r + 1$; core c copies to its MPB ck_{i+1} in step $r + 2$; etc. If each of these steps takes δ time units, the total time to transfer the message is roughly $2n\delta$. With double buffering, the message chunks are two times smaller and so, message msg is split into chunks ck_1 to ck_{2n}. In a step r, core c can copy ck_{i+1} to the MPB while core c' gets ck_i. If each of these steps takes

[2] Of course, some MPB space needs to be allocated to the flags.

$\delta/2$ time units, the total time is roughly only $n\delta$.

The pseudocode of OC-Bcast for a core c is presented in Algorithm 1. To broadcast a message, all cores invoke the broadcast function (line 20). The input variables are msg, a memory location in the private memory of the core, and $root$, the id of the core broadcasting the message. The broadcast function moves the content of msg on the $root$, to the private memory of all other cores.

The pseudocode assumes that the total number of processes is P and that the degree of the data propagation tree used by OC-Bcast is k. Each core c has a unique data parent $dataParent_c$ in the data propagation tree, and a set of children $dataChildren_c$. The set $notifyChildren_c$ includes all the cores that core c should notify during the algorithm. Note that a core c can be part of several binary trees used for notifications. In the example of Figure 2.2, if we consider core $c1$: $dataParent_{c1} = c0$; $dataChildren_{c1} = \{c8, c9, c10, c11\}$; $notifyChildren_{c1} = \{c3, c4, c8, c9\}$. These sets are computed at the beginning of the broadcast (line 15). MPBs are represented by the global variable MPB where $MPB[c]$ is the MPB of core c. A $notifyFlag$ and k $doneFlag$ (one per child) are allocated in each MPB to manage synchronizations between cores. The rest of the MPB space is divided into two buffers to implement double buffering.

The $broadcast\_chunk$ function is used to broadcast a chunk. Each chunk is uniquely identified using a tuple <$bcastID$,$chunkID$>. Chunk ids are used for notifications. To implement double buffering, the two buffers in the MPB are used alternatively: for the chunk <$bcastID$,$chunkID$>, the buffer '$chunkID \mod 2$' is used. By setting the $notifyFlag$ of a core c to <$bcastID$,$chunkID$>, core c is informed that the chunk <$bcastID$,$chunkID$> is available in the MPB of its $dataParent_c$. Notifications are done in two steps. First, if a core is an intermediate node in a binary notification tree, it forwards the notification in this tree as soon as it receives it (line 28): in Figure 2.2, core $c1$ notifies $c3$ and $c4$ when it gets the notification from core $c0$. Then, after copying the chunk to its own MPB, it can start notifying the nodes that will get the chunk from its MPB (line 32): in Figure 2.2, core $c1$ then notifies $c8$ and $c9$. When a core finishes getting a chunk, it informs its parent using the corresponding $doneFlag$ (line 30). A core can copy a new chunk $chunkID$ in one of its MPB buffers, when all its children in the message propagation tree got the previous chunk ($chunkID - 2$) that was in the same buffer (line 22). Note that the $bcastID$ is needed to be able to differentiate between chunks of two messages that are broadcast consecutively. The broadcast function on core c returns when c has got the last chunk in its private memory (line 34), and it knows that the data in its MPB buffers is not needed by any other core (line 19).

Contention issues

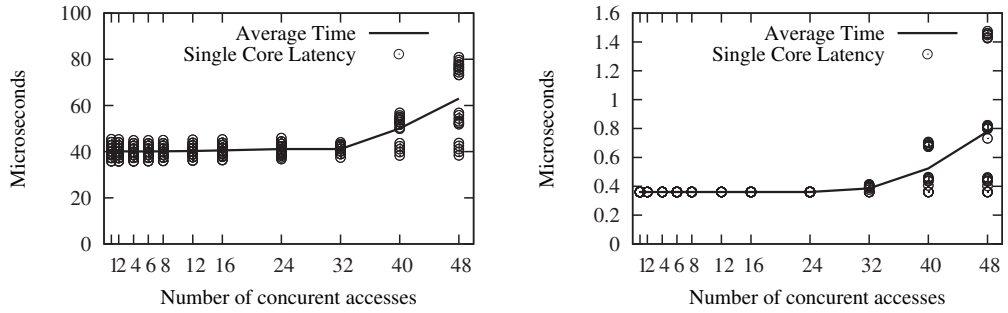
We identify two possible sources of contention related to RMA communication: the NoC mesh and the MPBs. Generally speaking, concurrent accesses to the off-chip private memory could be another source of contention. However, in the configuration without shared memory,

assumed throughout this chapter, each core has one memory rank for itself and there is no measurable performance degradation even when the 48 cores are accessing their private portion of the off-chip memory at the same time [104]. For better understanding of the possible sources of contention, we do the following experiments (Experimental settings are detailed in Section 2.5).

To understand if the mesh could be subject to contention, we have run an experiment that highly loads one link. We selected the link between tile (2,2) and tile (3,2). To put a maximum stress on this link, all cores except the ones located on these two tiles are repeatedly getting 128 cache lines from one core in the third row of the mesh, but on the opposite side of the mesh compared to their own location. For instance, a core located on tile (5,1) gets data from tile (0,2). Because of X-Y routing, all data packets go through the link between tile (2,2) and tile (3,2). The measurement of a MPB-to-MPB *get* latency between tile (2,2) and tile (3,2) with the heavily loaded link did not show any performance drop, compared to the load-free *get* performance. Therefore, at the current scale, the network cannot be a source of contention.

Contention could also arise from multiple cores concurrently accessing the same MPB. To evaluate this, we have run a test where cores are getting data from the MPB of core 0 (on tile (0,0)), and another test where cores are putting data into the MPB of core 0. For these tests, we select two representative scenarios of the access patterns in our broadcast algorithm presented in Section 2.4: parallel *gets* of 128 cache lines and parallel *puts* of 1 cache line. Note that having parallel *puts* of a large number of cache lines is not a realistic scenario since it would result in several cores writing to the same location. Figure 2.3(a) shows the impact on latency when increasing the number of cores executing *get* in parallel. Figure 2.3(b) shows the same results for parallel *put* operations. The x axis represents the number of cores executing *get* or *put* at the same time. The results are the average values over millions of iterations. In addition to the average latency, the performance of each core is displayed to better highlight the impact of contention (small circles in Figure 2.3). When all 48 cores are executing *get* or *put* in parallel, contention can be clearly noticed. In this case, the slowest core is more than two times slower than the fastest one for *get*, and more than four times slower for a *put* operation. Moreover we observed non-deterministic overhead after the contention threshold, by running the same experiment on other cores than core 0. It can be noticed that contention does not equally affect all cores, which makes it hard to model.

These experiments indicate that MPB contention has to be taken into account in the design of algorithms for collective operations. They show that up to 24 cores accessing the same MPB do not create any measurable contention. In our algorithms, this property justifies the necessity of limiting the number of the children, parameter k, at each level of the data propagation tree to avoid contention.



(a) Concurrent MPB *get* completion time (128 CL) (b) Concurrent MPB *put* completion time (1 CL)

Figure 2.3: MPB contention evaluation

2.5 Experimental Evaluation

In this section we evaluate the performance of OC-Bcast on Intel SCC and compare it with the two state-of-the-art broadcast algorithms based on two-sided communication: binomial tree and *scatter-allgather*. We consider their implementations from the RCCE_comm library [28]. RCKMPI library [101] uses the same algorithms, but still keeps their original MPICH2 implementation, not optimized for the SCC. Also, our experiments have confirmed that RCCE_comm currently performs better than RCKMPI. Thus, we have chosen to conduct the experiments using RCCE_comm, as the fastest available implementation of collectives on the SCC, to the best of our knowledge.

Our comparison metrics are latency and throughput of the broadcast primitive. The latency of the broadcast primitive is defined as the time elapsed between the call of the broadcast function by the source, and the time at which the message is available at all cores (including the source), *i.e.*, when the last core returns from the function. The throughput of the broadcast primitive is defined as the number of broadcasts completed, *i.e.* the corresponding message arrived at all destinations, by a single source in one second.

2.5.1 Setup

The experiments have been done using the default settings for the SCC: 533 MHz tile frequency, 800 MHz mesh and DRAM frequency and the standard LUT entries. We use the sccKit version 1.4.1.3, running a custom version of sccLinux, based on Linux 2.6.32.24-generic. We fix the chunk size used by OC-Bcast to 96 cache lines, which leaves enough space for flags (for any choice of k). The presented experiments use core 0 as the source. Selecting another core as the source gives similar results. A message is broadcast from the private memory of core 0 to the private memory of all other cores. The results are the average values over 10'000 broadcasts, discarding the first 1'000 results. For time measurement, we use global counters accessible by all cores on the SCC, which means that the timestamps obtained by different cores are directly

Algorithm 1 OC-Bcast (code for core c)

Global Variables:
1: P {total number of cores}
2: k {data tree output degree}
3: $MPB[P]$ {$MPB[i]$ is the MPB of core i}
4: $notifyFlag$ {MPB address of the flag, of the form <bcastID,chunkID>, used to notify data availability}
5: $doneFlag[k]$ {MPB address of the flags, of the form <bcastID,chunkID>, used to notify broadcast completion of a chunk}
6: $buffer[2]$ {MPB address of the two buffers used for double buffering}
Local Variables:
7: $bcastID \leftarrow 0$ {current broadcast id}
8: $chunkID$ {current chunk ID}
9: $dataParent_c$ {core from which c should get data}
10: $dataChildren_c$ {set of data children of c}
11: $notifyChildren_c$ {set of notify children of c}

12: **broadcast** (msg, $root$)
13: $bcastID \leftarrow bcastID + 1$
14: $chunkID \leftarrow 0$
15: $\{dataParent_c, dataChildren_c, notifyChildren_c\} \leftarrow$ prepareTree($root$, k, P)
16: **for all chunks at offset** i **of** msg **do**
17: $chunkID \leftarrow chunkID + 1$
18: **broadcast_chunk**($msg[i]$, $root$)
19: **wait until** $\forall child \in dataChildren_c$: $MPB[c].doneFlag[child] = (bcastID, chunkID)$

20: **broadcast_chunk** ($chunk$, $root$)
21: **if** $chunkID > 2$ **then**
22: **wait until** $\forall child \in dataChildren_c$: $MPB[c].doneFlag[child] \geq (bcastID, chunkID - 2)$
23: **if** $c = root$ **then**
24: *put* $chunk \longrightarrow MPB[c].buffer[chunkID \bmod 2]$
25: **else**
26: **wait until** $MPB[c].notifyFlag \geq (bcastID_c, chunkID_c)$
27: **for all** $child$ **such that** $child \in notifyChildren_c \setminus dataChildren_c$ **do**
28: *put* $(bcastID, chunkID) \longrightarrow MPB[child].notifyFlag$
29: *get* $MPB[c].buffer[chunkID \bmod 2] \longleftarrow MPB[dataParent_c].buffer[chunkID \bmod 2]$
30: *put* $(bcastID, chunkID) \longrightarrow MPB[dataParent_c].doneFlag[c]$
31: **for all** $child$ **such that** $child \in notifyChildren_c \cap dataChildren_c$ **do**
32: *put* $(bcastID, chunkID) \longrightarrow MPB[child].notifyFlag$
33: **if** $c \neq root$ **then**
34: *get* $chunk \longleftarrow MPB[c].buffer[chunkID \bmod 2]$

comparable. To avoid cache effects in repeated broadcasts, we preallocate a large array and in every broadcast we operate on a different (currently uncached) offset inside the array.

2.5.2 Evaluation of OC-Bcast

We have tested the algorithms with message sizes ranging from 1 cache line (32 bytes) to 32'768 cache lines (1 MiB). We first focus on the latency of short messages, and then analyze the throughput of large messages. Regarding the binomial tree and *scatter-allgather* algorithms, our experiments have confirmed that the former performs better with small messages, whereas the latter is a better fit for large messages. Therefore, we compare OC-Bcast only with the better one for a given message size. We consider three values of k (2, 7, 47) to represent different OC-Bcast tree depths.

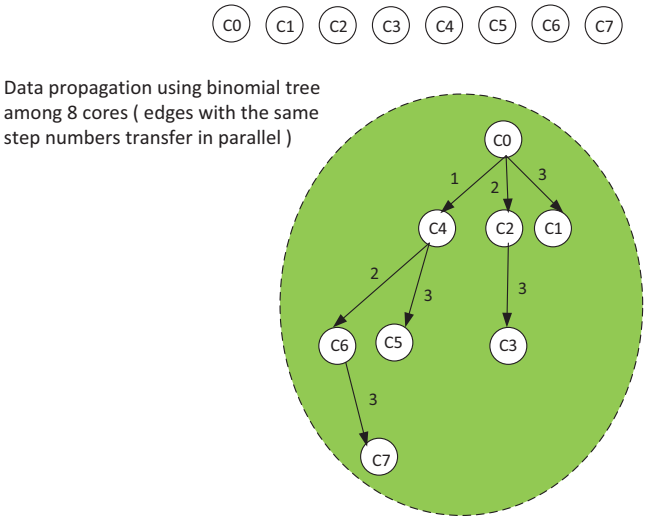


Figure 2.4: Message propagation using binomial tree

Latency of OC-Bcast vs. binomial tree broadcast for small messages

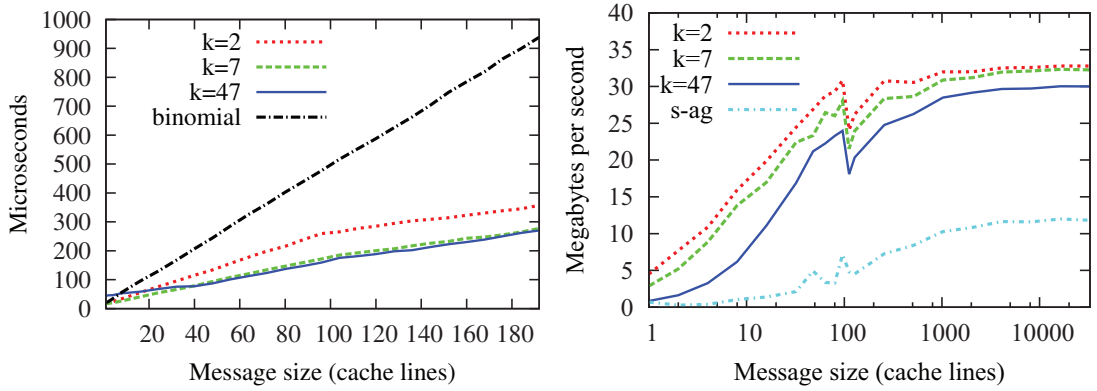
The binomial tree broadcast algorithm is based on a recursive tree. The set of nodes is divided into two subsets of $\lfloor \frac{P}{2} \rfloor$ and $\lceil \frac{P}{2} \rceil$ nodes. The root, belonging to one of the subsets, sends the message to one node from the other subset. Then, broadcast is recursively called on both subsets. Figure 2.4 depicts the mechanism of this algorithm.

Figure 2.5(a) shows the latency of messages of size $m \leq 2M_{oc}$. Even for messages of one cache line, OC-Bcast with $k = 7$ provides 27% improvement compared to the binomial tree ($16.6\mu s$ vs. $21.6\mu s$). As expected, the difference grows with the message size, since a larger message implies more off-chip memory accesses in the RCCE_comm algorithms, but not in OC-Bcast. It can also be noticed that large values of k help improving the latency in OC-Bcast by reducing the depth of the tree. For message size between 96 and 192 cache lines, the latency of OC-Bcast with $k = 7$ is around 25% better than with $k = 2$.

Throughput of OC-Bcast vs. *scatter-allgather* broadcast for large messages

The *scatter-allgather* broadcast algorithm has two phases. During the *scatter* phase, the message is divided into P equal slices[3], where P is total number of cores, of size $m_s = m/P$. Each core then receives one slice of the original message. The second phase of the algorithm is *allgather*, during which a node should obtain the remaining $P-1$ slices of the message. The *allgather* phase implemented in RCCE_comm uses the Bruck algorithm [26]: At each step, core i sends to core $i-1$ the slices it received in the previous step. Figure 2.6 depicts the mechanisms of this algorithm.

[3]For simplicity, we assume that $P|m$.



(a) Broadcast latency

(b) Broadcast throughput

Figure 2.5: Experimental comparison of broadcast algorithms (k=x : OC-Bcast with the corresponding value of k; binomial : RCCE_comm binomial; s-ag : RCCE_comm scatter-allgather)

The results of the throughput evaluation are given in Figure 2.5(b) (note that the x-axis is logarithmic). OC-Bcast gives an almost threefold throughput increase compared to the two-sided *scatter-allgather* algorithm. The OC-Bcast performance drop for a message of 97 cache lines is due to the chunk size. Recall that the size of a chunk in OC-Bcast is 96 cache lines. A message of 97 cache lines is divided into a 96 cache lines chunk and 1 cache line chunk. The second chunk is then limiting the throughput. For large messages, this effect becomes negligible since there is always at most one non-full chunk.

2.6 Related Work

A message-passing manycore chip, such as the SCC, is very similar to many existing HPC systems since it gathers a large number of processing units connected through a high-performance RMA-based network. Broadcast has been extensively studied in these systems. Algorithms based on a k-ary tree have been proposed [17]. In MPI libraries, binomial trees and *scatter-allgather* [94] algorithms are mainly considered [41, 98]. A binomial tree is usually selected to provide better latency for small messages, while the *scatter-allgather* algorithm is used to optimize throughput for large messages. These solutions are implemented on top of send/receive point-to-point functions and do not take topology issues into account. This is not an issue for small to medium scale systems like the SCC. However, it has been shown that for mesh or torus topologies, these solutions are not optimal at large scale: non-overlapping spanning trees can provide better performance [11].

To take advantage of the RMA capabilities of high-performance network interconnects such as InfiniBand [13], *one-sided put* and *get* operations, have been introduced [80]. In one-sided communication, only one party (sender or receiver) is involved in the data transfer and specifies the source and destination buffers. One-sided operations increase the design

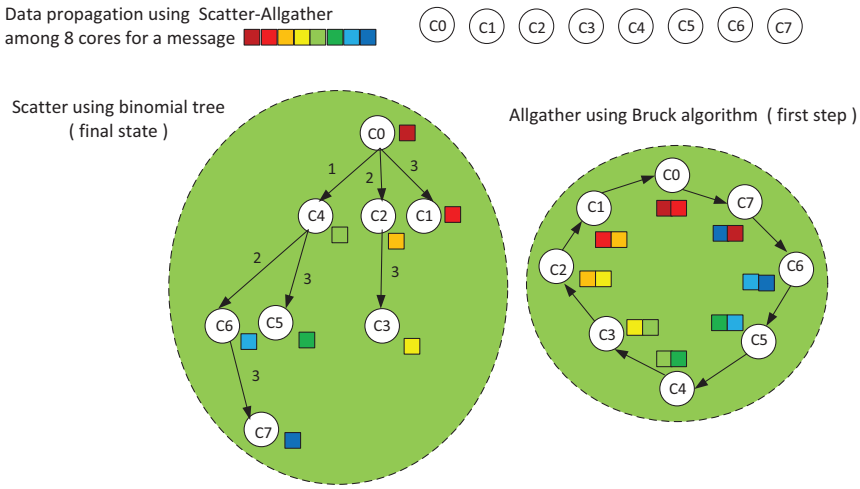


Figure 2.6: Message propagation using *scatter-allgather*

space for communication algorithms, and can provide better performance by overlapping communication and computation. On the SCC, RMA operations on the MPBs allow the implementation of efficient one-sided communication [74].

Most high-performance networks provide *Remote Direct Memory Access* (RDMA) [11, 13], *i.e.*, the RMA operations are offloaded to the network devices. Some works try to directly take advantage of these RDMA capabilities to improve collective operations [47, 51, 63, 97]. However, it is hard to reuse the results presented in these works in the context of the SCC for two main reasons: (i) they leverage hardware specific features not available on the SCC, *i.e.*, hardware multicast [51, 63], and (ii) they make use of large RDMA buffers [47, 97], whereas the on-chip MPBs have a very limited size (8 KB per core). Note also that accesses to the MPBs are not RDMA operations since message copying is performed by the core issuing the operation.

Two-sided communication can be implemented on top of one-sided communication [64]. This way, collective operations based on two-sided communication can benefit from efficient one-sided communication. Currently available SCC communication libraries adopt this solution. The RCCE library [74] provides efficient one-sided *put*/*get* operations and uses them to implement two-sided send/receive communication. The RCCE_comm library implements collective operations on top of two-sided communication [28]: the RCCE_comm broadcast algorithm is based on a binomial tree or on *scatter-allgather* depending on the message size. The same algorithms are used in the RCKMPI library [101].

The assumption of having only one program running at a time, as well as synchronous communication among cores, which holds for HPC applications, is not valid in general-purpose distributed systems. Therefore, using interrupts for asynchronous communication is a natural requirement for porting such systems to the SCC. Examples of SCC software relying upon inter-core interrupts are numerous [59, 62, 81, 102, 106]. However, to the best of our

knowledge, there are very few works that leverages Inter Process Interrupts (IPI) for collective communication. In [84] we took advantage of parallel IPIs on the SCC to implement high-performance asynchronous broadcast based on OC-Bcast. We show that although the use of IPIs for point-to-point communication is not efficient, but they could be useful to implement high-performance asynchronous collectives, *e.g.* broadcast.

3 High-Performance Map

Publication : *O. Shahmirzadi, T. Ropars and A. Schiper. High-Throughput Maps on Message-Passing Manycore Architectures: Partitioning versus Replication, In 20th International European Conference on Parallel Processing (EUROPAR), Porto, Portugal, August 2014.*

The advent of manycore architectures raises new scalability challenges for concurrent applications and operating systems. Implementing scalable data structures is one of them. Several manycore architectures provide hardware message passing as a means to efficiently exchange data between cores. In this chapter, we study the implementation of high-throughput concurrent maps in message-passing manycores. Partitioning and replication are the two approaches to achieve high throughput in a message-passing system. This chapter presents and compares different strongly-consistent map algorithms based on partitioning and replication. To assess the performance of these algorithms independently of architecture-specific features, we propose a communication model of message-passing manycores to express the throughput of each algorithm. The model is validated through experiments on a 36-core TILE-Gx8036 processor. Evaluations show that replication outperforms partitioning only in a narrow domain.

3.1 Introduction

Implementing scalable data structures is one of the basic problems in concurrent programming. To increase the throughput of data structures in shared memory architectures, several well-known techniques can be used including fine-grained locking, optimistic synchronization and lazy synchronization [49]. In message-passing systems, partitioning and replication are the two main approaches to improve the throughput of concurrent data structures [36]. Using partitioning, a data structure is partitioned among a set of servers that answer clients requests. Using replication, each client has a local copy of data structure in its private memory. Both have been considered in recent work on message-passing manycores [16, 109, 19], but performance comparisons are lacking. In this chapter we present a performance comparison of these two approaches for the implementation of high-throughput concurrent objects in

message-passing manycores, considering cases of linearizable and sequentially consistent maps. Note that existing studies made in distributed message-passing systems are only of little help because the high performance of NoCs provides a completely different ratio between computation and communication costs compared to large scale distributed systems.

Maps are used in many systems ranging from operating systems to key-value stores. Their performance is often crucial to the systems using them and have been shown to be an important performance bottleneck [16, 109, 19]. A map is an interesting case study because it is a good candidate to apply both partitioning and replication techniques. Since operations on different keys are independent, maps are easily partitionable [19]. Because a large majority of operations are usually lookup operations [14], replication can help handling a large number of local lookup requests concurrently.

Since message-passing manycore is a new technology, only few algorithms targeting this kind of architectures are available. Thus, to compare partitioning and replication in this context, we devise simple map algorithms that have been chosen to be representative of the design space. To compare our algorithms, we present a model of the communication in message-passing manycores, and express the throughput of our algorithms in this model. Using a performance model allows us to compare the algorithms independently of platform-specific features and to cover a large scope of manycore architectures. We use a 36-core Tilera TILE-Gx8036 processor to validate our model. Evaluations on the TILE-Gx shows an extremely poor performance for replication compare to partitioning. However some limitations of this platform, *i.e.* costly interrupt handling and lack of broadcast service, can be blamed for the poor performance of replication. Our model allows us to come up with a hypothetical platform based on the TILE-Gx, which does not suffer form its limitations. Our evaluations on this *ideal* platform show that even in the best setting in favor of replication, *i.e.* having highly efficient interrupt handling and hardware-based broadcast service, replication can outperform partitioning only when update operations are rare and replicas are located in the cache system of the cores.

To sum up, contributions of this chapter include:

- *Devising different algorithms to represent the design space of partitioning and replication:* since message-passing manycores are a new technology, only few algorithms targeting this kind of architectures are available. To compare partitioning and replication in this context, we propose simple map algorithms that have been chosen to be representative of the design space.

- *Coming up with a communication model to be able to compare different solutions independently of their underlying architecture:* to compare our algorithms, we present a model of the communication in message-passing manycores to express the throughput of our algorithms. Using a performance model let us to compare the algorithms independently of platform-specific features and to cover a large scope of manycore architectures. We use a 36-core Tilera TILE-Gx8036 processor to validate our model.

- *Evaluating our algorithms using our model under different assumptions and settings and showing that, under strong consistency, replication can outperform partitioning only in a narrow domain:* evaluations on the TILE-Gx shows an extremely poor performance for replication compare to partitioning. However some limitations of this platform, *i.e.* costly interrupt handling and lack of broadcast service, can be blamed for the poor performance of replication. Our model allows us to come up with a hypothetical platform based on the TILE-Gx, which does not suffer form its limitations. Our evaluations on this *ideal* platform show that even in the best setting in favor of replication, *i.e.* having highly efficient interrupt handling and hardware-based broadcast service, replication can outperform partitioning only when update operations are rare and replicas are located in the cache system of the cores.

This chapter is structured as follows. Section 3.2 introduces our baseline architecture. Section 3.3 specifies the underlying assumptions and goal of this part. Section 3.4 introduces the performance model and computes the throughput of different algorithms in this model. Section 3.5 provides a model validation on our baseline architecture and presents an extensive study of the performance of the different algorithms. Finally, related work is presented in Section 3.6.

3.2 The Tilera TILE-Gx8036

The TILE-Gx8036 is a general-purpose manycore developed by TILERA Corporation [8]. We use this platform as the baseline architecture for our studies in this part. In this section we describe the high level TILE-Gx8036 architecture and inter-core communication.

3.2.1 Architecture

The cores and the NoC of the TILE-Gx8036 are depicted in Figure 3.1. There are 36 full-fledged 1.2 Ghz, 64-bit processor cores with local cache, connected through a 2D mesh NoC. Each tile is connected to a router. The NoC uses high-throughput, low-latency links as well as deterministic X-Y routing. Cores and mesh operate at the same frequency.

Memory components are divided into (i) L1 data and instruction cache (32 KB each), (ii) 256 KB of L2 cache, and (iii) off-chip global memory. There is full hardware cache coherence among the L1 and L2 caches of different cores. Each core has access to the off-chip global memory through one of the two memory controllers, denoted by *MC* in Figure 3.1. Regions of the global memory can be declared private or shared (a page is a unit of granularity). We see this platform as a pure message-passing manycore, where each thread binds to a specific core and has its own private memory space.

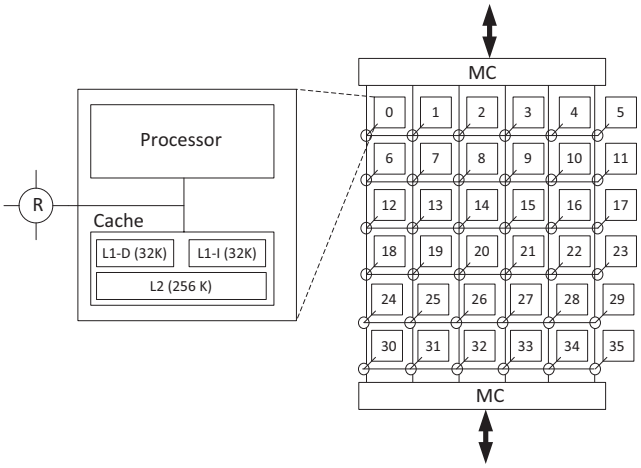


Figure 3.1: TILE-Gx8036 architecture

3.2.2 Inter-core communication

Each core has a dedicated hardware message buffer, capable of storing up to 118 64-bit words. The message buffer of each core is 4-way multiplexed, which means that every per-core buffer can host up to four independent hardware FIFO queues, containing incoming messages. The User Dynamic Network (UDN) allows applications to exchange messages directly through the mesh interconnect, without OS intervention, using special instructions. When a thread wants to exchange messages, it must be pinned to a core and registered to use the UDN (but it can unregister and freely migrate afterwards). When a message is sent from core A to core B, it is stored in the specified hardware queue of core B. The *send* operation is asynchronous and does not block, except in the following case. Since messages are never dropped, if a hardware queue is full, subsequent incoming messages back up into the network and may cause the sender to block. It is the programmer's responsibility to avoid deadlocks that can occur in such situations. When a core executes the *receive* instruction on one of the four local queues, the first message from the queue is returned. If there are no messages, the core blocks. The user can send and receive messages consisting of one or multiple words. Moreover a core, upon receipt of a new message in either of its incoming buffers, has the option of being notified by an inter core interrupt followed by executing an interrupt handler routine.

3.3 Assumptions and Goal

The study assumes a fault-free manycore architecture where a large set of single-threaded cores are connected through a network on chip. We assume that each core executes a single thread and that threads do not migrate between cores. Cores have their own private memory and can only communicate through message passing. Communication channels are asynchronous and FIFO. Messages are composed of a set of words and can have various size.

Three operations are available to send messages: $send$, $broadcast$ and $multicast$. Operation $send(m,i)$ sends message m to thread i. Operation $broadcast(m)$ sends m to all threads. Operation $multicast(m,list)$ sends m to all threads in $list$. Messages can be received using a *synchronous* $receive$ function. Operation $receive(m)$ blocks until message m can be received. Alternatively, threads can be interrupted when a new message is available.

This chapter studies the implementation of a concurrent map with strong consistency criteria, *i.e.* linearizability and sequential consistency. A map is a set of items indexed by unique keys that provides $lookup$, $update$ and $delete$ operations. Operation $update(key,val)$ associates key with the value val. Operation $lookup(key)$ returns the value associated with key (or $null$ if no value is associated with key). We assume that $delete(key)$ is implemented using $update(key,null)$.

3.4 Algorithms and Analytical Modeling

This section describes the algorithms studied in this chapter and presents their performance model. We start by describing our methodology for performance modeling followed by describing and modeling the linearizable and sequential consistent map algorithms. The main reason to use an analytical model is to be able to compare replication and partitioning in a general case so that the final conclusions are not biased towards features of an existing platforms, *e.g.* TILE-Gx. However, as we will see in Section 3.5, analytical modeling also helps us to concretely understand the performance bottlenecks and to be able to assess the algorithms under different architectures, configurations and load distributions. Moreover it can help manycore programmers to decide about their implementation choice on different platforms.

3.4.1 Performance modeling

Manycore processors are usually provided with a highly efficient NoC. Therefore, we assume that the throughput of the algorithms presented in this section is limited by the performance of the cores. This assumption is validated by the experimental results presented in Section 3.5.2. Hence, to obtain the *maximum* throughput of one algorithm executed on a given number of cores, we need to compute T_{lup} and T_{upd}, the total number of CPU cycles[1] required to execute a $lookup$ and an $update$ operation respectively.

All algorithms make the difference between cores that execute as *clients*, *i.e.*, cores executing the user code and issuing operations on the concurrent map, and *servers*, *i.e.*, cores that are earmarked to execute map-related and/or protocol code. Depending on the number c of cores that execute the client code and the number s of cores that execute as server, clients or servers can be the bottleneck for the system throughput. Thus, for each operation op, we actually

[1] Obtaining a duration in seconds from a number of CPU cycles simply introduces a constant factor $1/CPU\_Freq$.

have to compute the number of CPU cycles it takes on the client (T^c_{op}) and on the server (T^s_{op}). Considering a load where the probability of having a lookup operation is p, and assuming that the load is evenly distributed among clients, the maximum throughput $\mathscr{T}^c$ achievable by clients is:

$$\mathscr{T}^c = \frac{c}{p \cdot T^c_{lup} + (1-p) \cdot T^c_{upd}} \tag{3.1}$$

An equivalent formula applies to servers. Hence, the maximum throughput $\mathscr{T}$ of an algorithm is:

$$\mathscr{T} = min(\mathscr{T}^c, \mathscr{T}^s) \tag{3.2}$$

Table 3.1 lists the parameters that we use to describe the performance of our algorithms. To model the operations on the map, we consider a generic map implementation defined by three parameters o_{pre}, o_{lup} and o_{upd}. The underlying data structure used to implement the map is not the focus of the study. Parameter o_{pre} corresponds to the computation that a client has to do before accessing the map, *e.g.*, executing a hash function if a hash table is used to implement the map. Parameters o_{lup} and o_{upd} are the overheads corresponding to accessing the underlying data structure during a $lookup$ and an $update$ operation respectively.

We associate an overhead (*i.e.*, duration) in CPU cycles with each of the communication primitives introduced in Section 3.3. Additionally, we introduce the parameter T_{rtt}, representing round-trip time. More precisely, $T_{rtt}(send\_op, rcv\_op)$ is the round-trip time for messages sent with the $send\_op$ operation (*i.e.*, $send$, $broadcast$ or $multicast$) and received with the rcv_op operation (*i.e.*, rcv or $arcv$)[2]. If the round trip is initiated with $broadcast$ or $multicast$, it finishes when the answer from all destinations have been received.

Finally, in a configuration that uses multiple servers, a client needs to decide which server to contact for a given operation. In all our algorithms, the server selection depends on the key the operation applies to. Typically, it is based on a modulo operation that can have a non-negligible cost. Thus, o_{sel} stands for the server selection overhead. We assume that all other computational costs related to the execution of the algorithms are negligible.

In the following, we describe the different algorithms studied in this chapter, considering linearizable maps and sequential consistent maps respectively. For each algorithm, we provide a figure describing the communication patterns where all CPU overheads appear. We obtain the performance models directly from these figures.

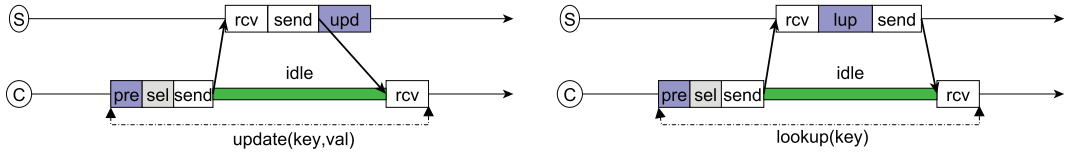
[2]The answer is always sent using $send$ and received using rcv.

parameter	description
c	number of clients
s	number of servers
o_{send}	overhead of $send(m)$
o_{bcast}	overhead of $broadcast(m)$
o_{mcast}	overhead of $multicast(m, list)$
o_{rcv}	overhead of a *synchronous* receive
o_{arcv}	overhead of an *asynchronous* receive
$T_{rtt}(s\_op, r\_op)$	round-trip time with s_op and r_op
o_{pre}	computation done before a map access
o_{lup}	access to the map for a $lookup$
o_{upd}	access to the map for an $update$
o_{sel}	server selection overhead
p	probability of a $lookup$ operation

Table 3.1: Model parameters

3.4.2 Linearizable map

Our goal is to propose linearizable map algorithms that are representative of the design space in a message-passing manycore. Hence, as a basic solution based on partitioning, we consider the approach proposed in [19]: the map is partitioned among a set of servers that clients access for every requests. A typical improvement of such a client/server approach is to introduce caching on client side [103]. We study a second algorithm based on this solution. Regarding replication, the solutions used in distributed systems cannot be directly applied to message-passing manycores: In a distributed system, a server is typically replicated to reduce the latency observed by clients by placing the replicas closer to the clients. In a manycore chip, the NoC provides very low latency between cores. Creating a few replicas of a server hosting a map is not an interesting approach. The only advantage it provides is to allow processing multiple lookup operations in parallel. However, this cannot make replication attractive compared to partitioning since partitioning provides the same advantage without the complexity of ensuring replica consistency during update. Thus, the only way for replication to provide benefits in the context of manycores, is to have a replica of the map on each core, so that clients can lookup the keys locally. We study three replication algorithms based on this idea. The first is based on the traditional approach consisting in using atomic broadcast to implement update operations. With such a solution, lookups require remote synchronization to ensure linearizability. Hence, one can argue that the goal of replication is not achieved. That is why we propose a second algorithm where lookups do not require any remote synchronization. In this case, update operations have to be made more complex to ensure linearizability. However both of the former replication solutions need sequencer servers to provide total order. To come up with a server-less protocol, we bring a variant of two phase commit protocol in which the lookups are purely local without any remote synchronization. However getting rid of the servers, comes at a price: the issuer of the update needs to abort the operation and issue it again in case another conflicting update exists.



(a) Update (b) Lookup

Figure 3.2: Simple partitioning

We describe now the five algorithms and model their throughput. We present first the simple partitioning algorithm, then the three replication ones, and finally the one based on partitioning with caching. They are presented is this order to gradually introduce the techniques we use to model their throughput.

I) Partitioned map (PART_SIMPLE)

In this approach called PART_SIMPLE, each server handles a subset of the keys. In this algorithm each client contacts a corresponding server to perform lookup and update on a key. Both operations block until the client receives a response from the server, which trivially ensures linearizability. The pseudocode of this algorithm is given in Figure 3.3[3]. The communication pattern is described in Figure 3.2. It is the same for a $lookup$ and an $update$ operation. The only difference is that applying the update can be removed from the critical path of the client (see Figure 3.2(a)). Computing T^s_{op} (where op is upd or lup), T^c_{lup} and T^c_{upd} based on Figure 3.2 is trivial:

$$T^s_{op} = o_{rcv} + o_{op} + o_{send} \tag{3.3}$$

$$T^c_{lup} = o_{pre} + o_{sel} + T_{rtt}(send, rcv) + o_{lup} \tag{3.4}$$

$$T^c_{upd} = o_{pre} + o_{sel} + T_{rtt}(send, rcv) \tag{3.5}$$

II) Replicated map – Lookups with remote synchronization (REP_REMOTE)

In replication approaches, lookups should be synchronized with updates to avoid violating linearizability as illustrated by Figure 3.4, where lookups return locally with no synchronization. Moreover all updates should be applied in total order in all replicas. The first two replication solutions provide total order among updates using atomic broadcast while the third solution ensures it using a variant of two phase commit protocol. In the first replication algorithm, called REP_REMOTE, lookups are totally ordered with respect to update operations.

Before detailing the algorithm, we need to discuss the atomic broadcast (*abcast*) implementa-

[3]For simplicity, we present the algorithms only for a single given key.

Algorithm 2 PART_SIMPLE (code for client *c*)

Global Variables:
1: S {total number of server cores}

2: **lookup** (key)
3: $myServer \leftarrow key\%S$
4: $send(LUP, key)$ *to* $myServer$
5: ***wait until*** val *is received from* $myServer$
6: $return(val)$

7: **update** (key, val)
8: $myServer \leftarrow key\%S$
9: $send(UPD, key, val)$ *to* $myServer$
10: ***wait until*** ACK *is received from* $myServer$

Algorithm 3 PART_SIMPLE (code for server *s*)

Local Variables:
1: map {map partition }

2: **upon rcv** ($command, key, val$) **from client** c
3: **if** $command = UPD$ **then**
4: $map.update(key, val)$
5: $send(ACK)$ *to* c
6: **else**
7: $val = map.lookup(key)$
8: $send(val)$ *to* c

Figure 3.3: Linearizable partitioned map without caching

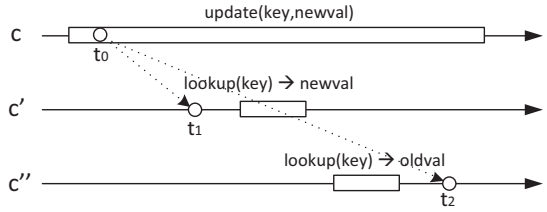


Figure 3.4: Non-linearizable execution with a replicated map

tion. To choose among the five classes of atomic broadcast algorithms presented in [35], we use three criteria. First, the number of messages exchanged during *abcast* should be minimized to limit the CPU cycles used for communication. This implies that solutions relying on multiple calls to broadcast should be avoided. Second, the solution should allow to increase the throughput by instantiating multiple instances of the *abcast* algorithm. Indeed, to obtain a linearizable map, only the operations on the same key have to be ordered. Thus, if *abcast* is the bottleneck, being able to use multiple instances of *abcast*, each associated with a subset of the keys, can increase the system throughput. Finally, the performance of the algorithm should not be impacted if some processes do not have messages to broadcast. Clearly, if multiple instances of *abcast* are used, we cannot assume that all processes will always have requests to *abcast* for each subset of keys. Only fixed-sequencer-based algorithms meet all the criteria.

In a fixed sequencer atomic broadcast algorithm, one process (called server in the following) is in charge of assigning sequence numbers to messages. After contacting the sequencer, the thread calling *abcast* can broadcast the message and the sequence number. The communica-

tion pattern of REP_REMOTE for an update issued by client c is shown in Figure 3.5(a). For each lookup, the client has to contact the server in charge of the key to know the sequence number sn of the last update ordered by this server (see Figure 3.5(b)). Then, the lookup terminates once the client has delivered the update with sequence number sn. Pseudecode of this algorithm is given in Figure 3.6 and its correctness trivially follows. Note that in this algorithms delegating the task of broadcast to the server could lead to violation of linearizability: if an update on a key finishes on the issuing client before the corresponding value on the server is updated, a later lookup could still return the old value.

In this algorithm, interrupts are used to notify a client that it has a new update message to deliver. An alternative to avoid interrupts would be to buffer updates until the client tries to execute an operation on the map. At this time the client would deliver all pending updates before executing its own operation. However, such a solution would potentially require large hardware buffers to store pending updates. Relying on interrupts avoids this issue. Moreover receiving a batch of messages instead of one, upon raising an interrupt, could be translated into lower cost for asynchronous receives.

Computing the throughput of clients in this algorithm is complex because clients can be interrupted to deliver updates. But handling the interrupts is not always on the critical path of the clients. Indeed, one can notice that clients are idle during an operation while waiting for an answer from the server. An interrupt handled during this period would not be on the critical path. We define $\mathscr{O}^c$ as the maximum amount of time spent in interrupts handling that can be removed from the critical path of clients execution and update formula 3.1 in the following way:

$$\mathscr{T}^c = \frac{c}{p \cdot T^c_{lup} + (1-p) \cdot T^c_{upd} - \mathscr{O}^c} \tag{3.6}$$

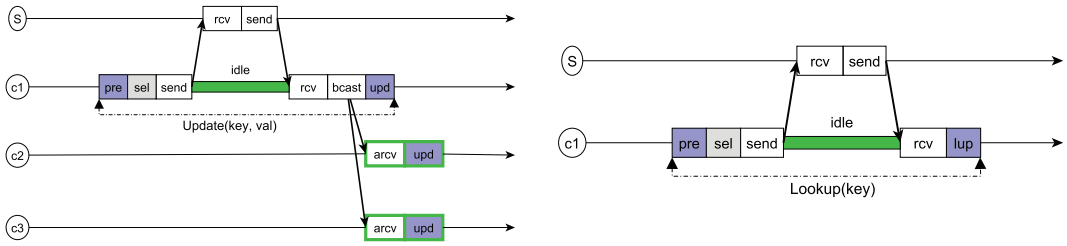
We deduce the cost of an update and a lookup operation from Figure 3.5.

$$T^s_{op} = o_{rcv} + o_{send} \tag{3.7}$$

$$T^c_{lup} = o_{pre} + o_{sel} + o_{lup} + T_{rtt}(send, rcv) \tag{3.8}$$

$$T^c_{upd} = o_{pre} + o_{sel} + o_{upd} + T_{rtt}(send, rcv) + o_{bcast} + (c-1) \cdot (o_{arcv} + o_{upd}) \tag{3.9}$$

$\mathscr{O}^c$ depends on T_{idle}, the idle time on a client during one operation, n_{idle}, the average number of idle periods per operation, T_{int}, the time required to handle an interrupt (green-border



(a) Update (b) Lookup

Figure 3.5: Replication with remote synchronization for lookups

boxes in Figure 3.5(a)), and n_{int}, the average number of asynchronous requests per operation:

$$T_{idle} = T_{rtt}(send, rcv) - o_{send} - o_{rcv} \tag{3.10}$$

$$n_{idle} = 1 \tag{3.11}$$

$$T_{int} = o_{arcv} + o_{upd} \tag{3.12}$$

$$n_{int} = (c-1) \cdot (1-p) \tag{3.13}$$

We compute $\mathscr{O}^c$ in three steps. We first compute the number of asynchronous requests that can be fully handled during one idle period (k), then the number of interrupts that can be fully overlapped with idle time on one client (n_{full}), and finally, the number of interrupts that can be partially overlapped with idle time on one client ($n_{partial}$).

$$k = \lfloor \frac{T_{idle}}{T_{int}} \rfloor \tag{3.14}$$

$$n_{full} = min(k \cdot n_{idle}, n_{int}) \tag{3.15}$$

$$n_{partial} = min(n_{int} - n_{full}, n_{idle}) \tag{3.16}$$

$$\mathscr{O}^c = n_{full} \cdot T_{int} + n_{partial} \cdot (T_{idle} - k \cdot T_{int}) \tag{3.17}$$

III) Replicated map – Lookups without remote synchronization (REP_LOCAL)

In REP_LOCAL, lookups do not require any remote synchronization (Figure 3.7(b)) but updates are more complex than in REP_REMOTE (Figure 3.7(a)). To provide linearizability, this algorithm ensures that during an update, no lookup can return the new value if a lookup by another client can return an older value. To do so, the update operation includes two phases of communication as shown in Figure 3.7(a). When client c runs an update, it first asks for a sequence number from the server, before atomically broadcasting the update message to all clients. Then it waits until all clients acknowledge the reception of this message. Finally, it broadcasts a second message to validate the update. If a client tries to lookup the key after

Algorithm 4 REP_REMOTE (code for replica c)

```
Global Variables:
 1: S {total number of servers}
Local Variables:
 2: map {map replica }
 3: maxsn {keeps the sequence number of the latest update
    for the key }

 4: lookup (key)
 5:     myServer ← key%S
 6:     send(SNREQ, key) to myServer
 7:     wait until sn is received from myServer and maxsn ≥
        sn
 8:     val ← map.lookup(key)
 9:     return(val)

10: update (key, val)
11:     myServer ← key%S
12:     send(INC, key) to myServer
13:     wait until sn is received from myServer
14:     bcast(UPD, key, val, sn)

15: upon adel(UPD, key, val, sn) from some replica c'
16:     map.update(key, val) {asynchronous total order de-
        livery}
17:     maxsn ← maxsn + 1
```

Algorithm 5 REP_REMOTE (code for server s)

```
Local Variables:
 1: abCtr {counter to assign total order sequence numbers}

 2: upon rcv (command, key) from replica c
 3:     if command = SNREQ then
 4:         send(abCtr) to c
 5:     else
 6:         abCtr ← abCtr + 1
 7:         send(abCtr) to c
```

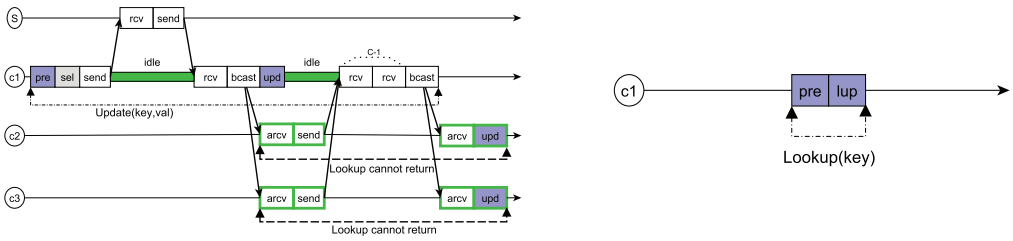
Figure 3.6: Linearizable replicated map with local lookups with remote synchronization

it has received the update message, the lookup cannot return until the validation has been received. This way a lookup that returns the new value always finishes after all clients have received the update message, which is enough to ensure linearizability [4]. The pseudocode of this algorithm is given in Figure 3.8. Theorem 3.4.1 proves the correctness of this algorithm.

Theorem 3.4.1 *Algorithms in Fig. 3.8 ensure linearizability with respect to the map operations.*

Proof If we prove that a map with only a single key is linearizable, due to the composability of linearizability, the whole map, which is composed of a set of independent key entries, is linearizable too. Considering only one key, the total order of updates is trivially ensured. Moreover if two updates on a key are executed with no timing overlap, in the global history of updates the second update is placed after the first one, since the first update is assigned with a smaller sequence number. Considering lookups, we show that the two following scenarios are not possible: (1) having two non-overlapping lookups, where the former one returns the new value and the latter one returns the old value, as it is shown in Figure 3.9(a); and (2)

[4] Update messages can be also received synchronously. In this case the time between sending the ACK back to the issuer and receiving the update from the issuer cannot be used to perform some other useful task, while on the positive side it avoids the cost of asynchronous receive. Our evaluations show that this trade-off is not in favor of the algorithm throughput, especially at scale. The main reason is that the length of waiting periods increases linearly with the increase in the number of replicas.



(a) Update (b) Lookup

Figure 3.7: Replication with no remote synchronization for lookups

having a non-overlapping update and lookup, where the update happens before the lookup and lookup returns the old value, as it is shown in Figure 3.9(b). Apart from these two cases, all other scenarios, with respect to the relative position of two operations, are safe with respect to linearizability, *i.e.* a linearizable history can be made.

Case (1): suppose this scenario happens according to Figure 3.9(a). In this case, we assume replica c_1 updates the new value. Assume t_1 and t_2 are the beginning and the end of the lookup operation on c_2 and t_3 and t_4 are the beginning and the end of the lookup operation on c_3 and $t_1 > t_4$. Replica c_3 should receive the $ACKALL$ for this update at some point before t_4, called A (execution of line 18). However replica c_1 should have sent this update to the replica c_2 at some point after t_1, called B (execution of line 14). Note that B is not necessarily before t_2. This means that $B \rightarrow A$ since replica c_2 should have sent the ACK message to the replica c_1 (execution of line 23), before replica c_1 could send the $ACKALL$ message to replica c_3 (execution of line 16). Therefore $t_1 \rightarrow B$, $B \rightarrow A$, $A \rightarrow t_4$, and so $t_1 \rightarrow t_4$. This means that $t_1 \leq t_4$, a contradiction.

Case (2): Suppose this scenario happens according to Figure 3.9(b). Assume t_1 and t_2 are the beginning and the end of the lookup operation on c_2 and t_3 and t_4 are the beginning and the end of the update operation on c_1 and $t_1 > t_4$. It means that replica c_2 atomically delivers the update from replica c_1 at some point after t_1, called A (execution of line 20). Moreover it means that replica c_1 receives the ACK for this update from replica c_2 at some point before t_4, called B (execution of line 15). Therefore we have $t_1 \rightarrow A$, $A \rightarrow B$, $B \rightarrow t_4$, and so $t_1 \rightarrow t_4$. This means $t_1 \leq t_4$, a contradiction. □

Since this algorithm introduces idle time on clients and uses interrupts, computing the throughput of clients is based on Formula 3.6. Notice that update operations introduce

Algorithm 6 REP_LOCAL (code for replica c)

Global Variables:
1: C {total number of replicas}
2: S {total number of sequencer servers}
Local Variables:
3: map {map replica }
4: $maxsn$ {keeps sequence number of the latest update for the key}
5: $flag[C]$ {set of C local flags}

6: **lookup** (key)
7: **wait until** $\nexists i \mid flag[i] = key$
8: $val \leftarrow map.lookup(key)$
9: $return(val)$

10: **update** (key, val)
11: $myServer \leftarrow key\%S$
12: $send(INC, key)$ *to* $myServer$
13: **wait until** sn *is received from myServer*
14: $bcast(UPD, key, val, sn)$
15: **wait until** $rcv(ACK, key)$ *from all*
16: $bcast(ACKALL, key)$

17: **upon arcv**($ACKALL, key$) from some replica c'
18: $flag[c'] \leftarrow nil$
19: $map.update(key, val)$ {asynchronous total order delivery}

20: **upon adel**(UPD, key, val, sn) from some replica c'
21: $maxsn \leftarrow maxsn + 1$
22: $flag[c'] \leftarrow key$
23: $send(ACK, key)$ *to* c'

Algorithm 7 REP_LOCAL (code for server s)

Local Variables:
1: $abCtr$ {counter to assign total order sequence numbers}
2: **upon rcv** (INC, key) **from some replica** c
3: $abCtr \leftarrow abCtr + 1$
4: $send(abCtr)$ *to* c

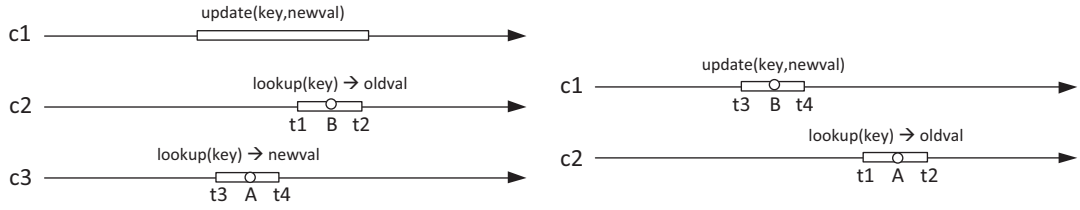
Figure 3.8: Linearizable replicated map with local lookups with no remote synchronization

two idle periods with different durations as well as two different costs for handling interrupts:

$$T_{idle\_1} = T_{rtt}(send, rcv) - o_{send} - o_{rcv} \tag{3.18}$$
$$T_{idle\_2} = max\big(T_{rtt}(bcast, arcv) - o_{bcast} - (c-1) \cdot o_{rcv} - o_{upd}, 0\big) \tag{3.19}$$
$$n_{idle\_1} = n_{idle\_2} = 1 - p \tag{3.20}$$
$$T_{int\_1} = o_{arcv} + o_{send} \tag{3.21}$$
$$T_{int\_2} = o_{arcv} + o_{upd} \tag{3.22}$$
$$n_{int\_1} = n_{int\_2} = (c-1) \cdot (1-p) \tag{3.23}$$

Here int_1 and int_2 correspond to the delivery of the first and second broadcast message respectively. Note that T_{idle_2} needs to consider the maximum between the actual computation and 0 to account for the fact that if o_{upd} is large, there might not be any idle time.

Computing the exact value of $\mathscr{O}^c$ in this case is a complex problem. Instead, we approximate this value using Formulas 3.14-3.17 with weighted averages for T_{idle} and T_{int}. For instance,



(a) Case 1 (b) Case 2

Figure 3.9: Scenarios used to prove the Theorem 3.4.1

here are the values we use for n_{idle} and T_{idle}:

$$n_{idle} = n_{idle\_1} + n_{idle\_2} \tag{3.24}$$

$$T_{idle} = \frac{T_{idle\_1} \cdot n_{idle\_1} + T_{idle\_2} \cdot n_{idle\_2}}{n_{idle}} \tag{3.25}$$

Finally, we deduce the cost of lookups and updates from Figure 3.7:

$$T^{s}_{upd} = o_{rcv} + o_{send} \tag{3.26}$$

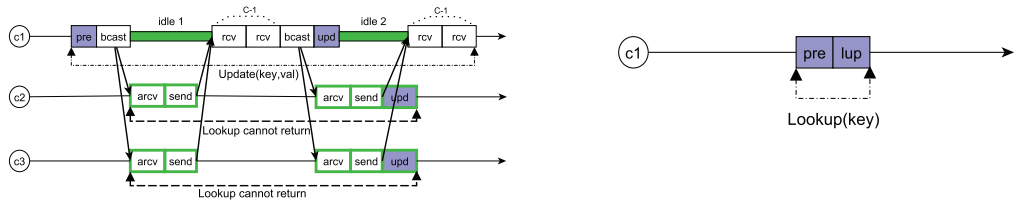
$$T^{c}_{upd} = o_{pre} + o_{sel} + T_{rtt}(send, rcv) + max\big(T_{rtt}(bcast, arcv), o_{bcast} + o_{upd} + (c-1) \cdot o_{rcv}\big) + o_{bcast} + (c-1) \cdot (2 \cdot o_{arcv} + o_{send} + o_{upd}) \tag{3.27}$$

$$T^{c}_{lup} = o_{pre} + o_{lup} \tag{3.28}$$

IV) Replicated map – Based on two phase commit (REP_2PC)

Previous replicated solutions rely on some dedicated servers to assign the sequence numbers to the messages, in order to ensure total order delivery of updates. However one might save these dedicated servers, by applying some variants of atomic commit protocols. In this way the solution does not rely on any sequencer, but upon detecting another update on the same key the current update should be aborted. Therefore in REP_2PC, lookups do not require any remote synchronization (Figure 3.10(b)), but updates are more complex compared to REP_LOCAL (Figure 3.10(a)).

A variant of two phase commit protocol provides total order of updates since as long as an update on a key is executing, other conflicting updates on that key will abort. To be more precise, upon issuing an update, a $VREQ$ message is broadcast to all the replicas and the issuer is blocked until it receives a vote from all. A YES vote from replica c means that another update on that key is executing on replica c. In this case, the issuer broadcasts an $ABORT$ message to all replicas to abort the current update and returns unsuccessfully. Otherwise it sends a commit message to all other replicas, meaning that it is safe for them to apply the update on that key. Each replica after applying the update sends an ACK back. Upon receiving



(a) Update with no conflicts

(b) Lookup

Figure 3.10: Replication using two phase commit

Algorithm 8 REP_2PC (code for replica c)

Global Variables:
1: C {total number of clients}
Local Variables:
2: map {map replica }
3: $flag$ {local flags to synchronize lookups on key to ensure linearizability}

4: **lookup** (key)
5: **wait until** $flag = -1$
6: $val \leftarrow map.lookup(key)$
7: $return(val)$

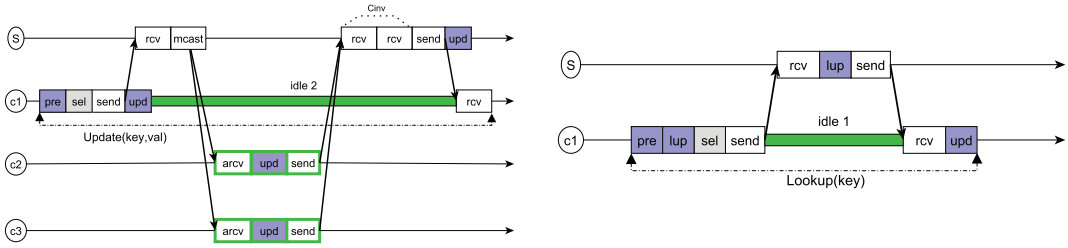
8: **update** (key, val)
9: $bcast(VREQ, key)$
10: **wait until** $vote$ *is received from all*
11: **if** all votes are NO **then**
12: $bcast(COMMIT, key, val)$
13: **wait until** ACK *is received from all*
14: $return(0)$ {no conflict}
15: **else**
16: $bcast(ABORT, key)$
17: $return(1)$ {conflict}

18: **upon arcv**($command, key, val$) from some replica c'
19: **if** $command = VREQ$ **then**
20: **if** $flag < c'$ **then**
21: $send(NO)$ *to* c'
22: $flag \leftarrow c'$
23: **else**
24: $send(YES)$ *to* c'
25: **if** $command = COMMIT$ **then**
26: $flag \leftarrow -1$
27: $send(ACK)$ *to* c'
28: $map.update(key, val)$
29: **if** $command = ABORT$ **then**
30: **if** $flag = c'$ **then**
31: $flag = -1$

Figure 3.11: Linearizable replicated map with local lookups using two phase commit

the ACK from all, the issuing replica terminates the update successfully. However lookups still need to use a similar synchronization technique which is used in REP_LOCAL to ensure linearizability: as far as $flag$ is not equal to -1, meaning that an update is pending on a key, the lookup is not allowed to return the value of that key. The pseudocode of this algorithm is given in Figure 3.11. Correctness of this algorithm can be proved similarly to the proof of Theorem 3.4.1. Just note that to ensure liveness, we use replica ids to break the ties when multiple replicas issue update on the same key at the same time (line 20).

The parameters needed to obtain the maximum throughput of this algorithm are computed using Figure 3.10. Note that in this algorithm there is no notion of server, and so server



(a) Update with invalidation of local caches

(b) Lookup with cache miss

Figure 3.12: Partitioning with local caches

selection cost. The calculation methods are similar to those of REP_LOCAL:

$$T_{idle\_1} = max\big(T_{rtt}(bcast, arcv) - o_{bcast} - (c-1)\cdot o_{rcv}, 0\big) \tag{3.29}$$

$$T_{idle\_2} = max\big(T_{rtt}(bcast, arcv) - o_{bcast} - (c-1)\cdot o_{rcv} - o_{upd}, 0\big) \tag{3.30}$$

$$n_{idle\_1} = n_{idle\_2} = 1 - p \tag{3.31}$$

$$T_{int\_1} = o_{arcv} + o_{send} \tag{3.32}$$

$$T_{int\_2} = o_{arcv} + o_{send} + o_{upd} \tag{3.33}$$

$$n_{int\_1} = n_{int\_2} = (c-1)\cdot(1-p) \tag{3.34}$$

$$T^{c}_{upd} = o_{pre} + T_{rtt}(bcast, arcv) + max\big(T_{rtt}(bcast, arcv), o_{bcast} + o_{upd} + (c-1)\cdot o_{rcv}\big) + (c-1)\cdot(2\cdot o_{arcv} + 2\cdot o_{send} + 2\cdot o_{rcv} + o_{upd}) \tag{3.35}$$

$$T^{c}_{lup} = o_{pre} + o_{lup} \tag{3.36}$$

V) Partitioned map - With local caches (PART_CACHING)

The PART_CACHING algorithm extends PART_SIMPLE to introduce caching on client side. If a lookup hits the cache, the pattern is the same as the one in Figure 3.7(b). Otherwise, the communication pattern is shown in Figure 3.12(b). It includes a first local lookup that fails and an update of the local cache once the value has been retrieved from the server.

When a key is updated, local copies of the associated value need to be invalidated. As shown in Figure 3.12(a), the server invalidates local copies using multicast. Once the server has received an acknowledgment from all clients involved, the operation can terminate. This algorithm could also be viewed as a hybrid solution between partitioning and replication, since the local caches are *replicated* on different clients and need to remain consistent among each other using invalidations. The pseudocode of this algorithm is given in Figure 3.13. Theorem 3.4.2 proves the correctness of this algorithm.

Theorem 3.4.2 *Algorithms in Fig. 3.13 ensure linearizability with respect to the map operations.*

Algorithm 9 PART_CACHING (code for client c)

```
Global Variables:
 1: S {total number of servers}
Local Variables:
 2: map {map local cache }

 3: lookup (key)
 4:     val ← map.lookup(key)
 5:     if key is in the local cache then
 6:         return(val)
 7:     else
 8:         myServer ← key%S
 9:         send(LUP,key) to myServer
10:         wait until val is received from myServer
11:         map.update(key,val)
12:         return(val)

13: update (key,val)
14:     myServer ← key%S
15:     send(UPD,key) to myServer
16:     wait until (ACK) is received from myServer

17: upon rcv (INV,key,c′) from some server s′
18:     map.update(key,nil)
19:     send(ACKINV) to s′
```

Algorithm 10 PART_CACHING (code for server s)

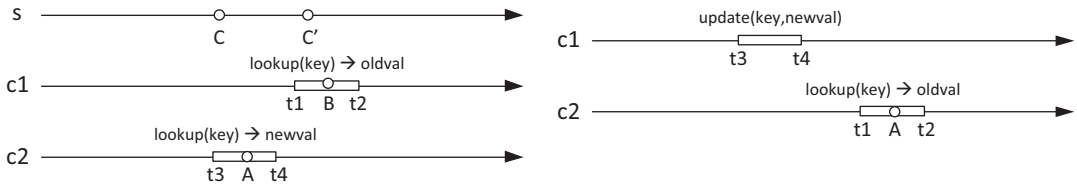
```
Local Variables:
 1: map {map partition}

 2: upon rcv (command,key,val) from client c
 3:     if command = UPD then
 4:         bcast(INV,key,c) to invalidation set of key
 5:         wait until rcv(ACKINV) from all clients in invali-
            dation set of key
 6:         map.update(key,val)
 7:         send(ACK) to c
 8:     else
 9:         val = map.lookup(key)
10:         add c to invalidation set of key
11:         send(val) to c
```

Figure 3.13: Linearizable partitioned map with caching

Proof Similarly to the proof of Theorem 3.4.1, we show that the two cases in Figures 3.14(a) and 3.14(b) cannot happen:

Case (1): Suppose that the scenario of Figure 3.14(a) happens. Assume t_1 and t_2 are the beginning and the end of lookup operation on c_1 and t_3 and t_4 are the beginning and the end of lookup operation on c_2 and $t_1 > t_4$. There are four different subcases considering these two lookup operations. (i) Both lookups are remote: in this case the mentioned scenario in Figure 3.14(a) is not possible clearly since the second lookup returns a value which is not older than the $newval$. (ii) The first lookup is remote and the second lookup is local: this means that client c_1 receives the invalidation at some point after t_1, called B (not necessarily before t_2). Moreover assume that update of the new value at server s finished at point C (execution of line 6 of the server code). We will have $C \rightarrow t_4$, $B \rightarrow C$ and $t_1 \rightarrow B$, which means $t_1 \rightarrow t_4$, a contradiction. (iii) The first lookup is local and the second lookup is remote: this means that client c_2 updates its local value to the $newval$ at some point before t_4 which is called A (execution of line 11 of the client code). Assume server s sends the new value to client c_2 at point C (execution of line 11 of the server code) and B is the point when the client c_1 executes line 9 of the client code. Therefore we have $t_1 \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$ and $A \rightarrow t_4$ which implies $t_1 \rightarrow t_4$, a contradiction. (iv) Both lookups are local: assume C and C' are the times on the server when it sends to the clients c_1 and c_2 the old and the new values respectively



(a) Case 1 (b) Case 2

Figure 3.14: Scenarios used to prove the Theorem 3.4.2

by execution of line 11 of the server code. Clearly C should be before C'. Therefore we have $C' \rightarrow t_4$ and $t_1 \rightarrow C$, and so $t_1 \rightarrow t_4$, a contradiction.

Case (2): Suppose this scenario happens according to Figure 3.14(b), where client c_1 issues an update with new value and client c_2 issues a lookup which returns the old value. Assume t_1 and t_2 are the beginning and the end of lookup operation on c_2 and t_3 and t_4 are the beginning and the end of the update operation on c_1 and $t_1 > t_4$. Assume that client c_2 returns the lookup value at time A. In this case the invalidation will be received after point A on client c_2. Therefore $t_1 \rightarrow A$ and $A \rightarrow t_4$, and so $t_1 \rightarrow t_4$, a contradiction. □

To model the performance of this algorithm, we need to introduce two additional parameters: p_{ll} is the probability that a lookup hits the local cache; n_{inv} is the average number of copies that needs to be invalidated when a key is updated. Note that the two parameters are correlated:

$$n_{inv} = \frac{p}{1-p} \cdot (1 - p_{ll}) \tag{3.37}$$

Indeed, the number of lookups on a key that requires an access to the server correspond to the number of copies that will have to be invalidated during the next update of that key. Thus, n_{inv} is equal to the average number of lookups between two updates ($\frac{p}{1-p}$) multiplied by the probability for lookups to require accessing the server.

The probability that a lookup hits the cache depends on the distribution of the accesses to one key among the clients: If some clients access a key much more often than others, the number of cache hits will be high. For a given probability distribution, we can use its probability mass function $pmf_{key}(c)$ to compute p_{ll}. For a cache hit to occur, a client should lookup a key two times and the key should not be updated in the meantime. Thus, we compute the probability that a lookup on key by client k is preceded by a sequence of i consecutive lookups made by other clients and by one lookup made by k, that is $p \cdot pmf_{key}(k) \cdot (p \cdot (1 - pmf_{key}(k)))^i$. To obtain p_{ll}, we need then to consider all possible values of i and to compute a weighted

average among all clients:

$$P_{ll} = \sum_{k=0}^{c-1} pmf_{key}(k) \cdot \sum_{i=0}^{\infty} p \cdot pmf_{key}(k) \cdot \Big(p \cdot \big(1 - pmf_{key}(k)\big)\Big)^i \tag{3.38}$$

For a uniform distribution of the key accesses, *i.e.* all the clients have the same probability of accessing a given key ($pmf_{key}(k) = \frac{1}{C}$), the general formula simplifies as follows:

$$P_{ll} = \sum_{i=0}^{\infty} \frac{p}{C} \cdot \big(\frac{p \cdot (C-1)}{C}\big)^i = \frac{p}{p + C \cdot (1-p)} \tag{3.39}$$

Since the communication patterns for this algorithm includes two idle periods of different duration, we apply Formulas 3.24-3.25 to compute $\mathscr{O}^c$ with:

$$T_{idle\_1} = o_{lup} + T_{rtt}(send, rcv) - o_{send} - o_{rcv} \tag{3.40}$$
$$n_{idle\_1} = p \cdot (1 - p_{ll}) \tag{3.41}$$
$$T_{idle\_2} = max\big(0, T_{rtt}(send, rcv) - o_{send} - o_{rcv} + T_{rtt}(mcast, arcv) - o_{upd}\big) \tag{3.42}$$
$$n_{idle\_2} = 1 - p \tag{3.43}$$
$$T_{int} = o_{arcv} + o_{send} + o_{upd} \tag{3.44}$$
$$n_{int} = (1-p) \cdot n_{inv} \tag{3.45}$$

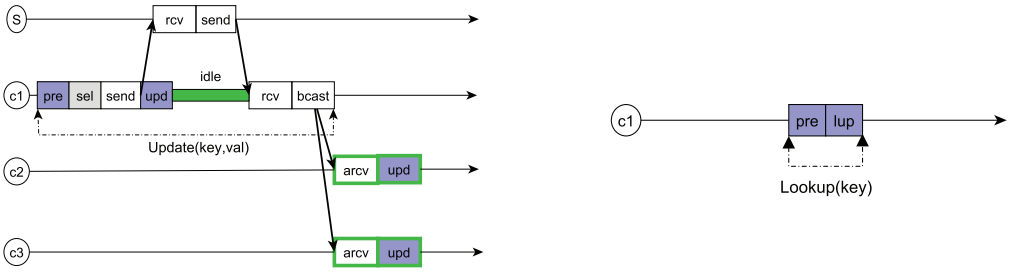
Note that the formula for T_{idle_2} assumes that the cost of $T_{rtt}(mcast, arcv)$ depends on n_{inv}. If $n_{inv} = 0$, then $T_{rtt}(mcast, arcv) = 0$.

The cost of a lookup depends whether there is a cache hit or a cache miss. The cost of a cache hit is the same as a lookup with REP_LOCAL (Formula 3.56). Otherwise, the cost is given by Figure 3.12(b). Together we get:

$$T^c_{lup} = p_{ll} \cdot (o_{pre} + o_{lup}) + (1 - p_{ll}) \cdot \big(o_{pre} + 2 \cdot o_{lup} + o_{sel} + T_{rtt}(send, rcv) + o_{upd}\big) \tag{3.46}$$
$$T^s_{lup} = (1 - p_{ll}) \cdot (o_{rcv} + o_{lup} + o_{send}) \tag{3.47}$$

The cost of updates is computed based on Figure 3.12(a). To compute the cost on the server, we do not consider the time it waits for acks of the invalidation messages as idle time. We assume that the server always have requests from other clients to handle during this time:

$$T^c_{upd} = o_{pre} + o_{sel} + max\big(o_{upd} + o_{send} + o_{rcv}, T_{rtt}(send, rcv) + T_{rtt}(mcast, arcv)\big) + n_{inv} \cdot (o_{arcv} + o_{send} + o_{upd}) \tag{3.48}$$
$$T^s_{upd} = (n_{inv} + 1) \cdot o_{rcv} + o_{mcast} + o_{send} + o_{upd} \tag{3.49}$$



(a) Update (b) Lookup

Figure 3.15: Sequential consistent replication

3.4.3 Sequential consistent map

To be able to assess the affect of consistency criteria on the relative performance of different algorithms, we consider a weaker consistency criteria. Sequential consistency is weaker than linearizability since providing a global history of operations as well as keeping the local order of operations are enough to provide sequential consistency. Weaker consistency criteria such as fifo consistency and eventual consistency could also be useful, however they come up with a much broader design space for the algorithms, which is out of the scope of this chapter. In this subsection, we try to exploit sequential consistency in favor of our algorithms.

I) Replicated map

Considering replication, providing a total order of updates is enough to satisfy sequential consistency. Lookups can return immediately with no synchronization, and they can be freely placed in the global history of update operations to create a global history. Therefore replication algorithms that use a fixed sequencer to create a total order of updates, *i.e.* REP_REMOTE and REP_LOCAL, can be weakened to the algorithm depicted in the Figure 3.16 (We call this algorithm REP_SC for short). In this algorithm, updates are propagated using fixed-sequencer atomic broadcast and lookups return local values immediately. The communication pattern of this algorithm is shown in Figure 3.15 and its parameters are calculated as follows:

$$T_{idle} = max\big(T_{rtt}(send,rcv) - o_{send} - o_{rcv} - o_{upd}, 0\big) \tag{3.50}$$

$$n_{idle} = 1 - p \tag{3.51}$$

$$T_{int} = o_{arcv} + o_{upd} \tag{3.52}$$

$$n_{int} = (c-1)\cdot(1-p) \tag{3.53}$$

$$T^{s}_{upd} = o_{rcv} + o_{send} \tag{3.54}$$

$$T^{c}_{upd} = o_{pre} + o_{sel} + max\big(T_{rtt}(send,rcv), o_{send} + o_{rcv} + o_{upd}\big) + o_{bcast} + (c-1)\cdot(o_{arcv} + o_{upd}) \tag{3.55}$$

$$T^{c}_{lup} = o_{pre} + o_{lup} \tag{3.56}$$

Algorithm 11 REP_SC (code for replica c)

Global Variables:
1: S {total number of sequencer servers}
Local Variables:
2: map {map replica }
3: $maxsn$ {keeps sequence number of the latest update for the key}

4: **lookup** (key)
5: $val \leftarrow map.lookup(key)$
6: $return(val)$

7: **update** (key, val)
8: $myServer \leftarrow key\%S$
9: $send(SNREQ, key)$ *to* $myServer$
10: **wait until** sn *is received from myServer*
11: $bcast(UPD, key, val, sn)$

12: **upon adel**(UPD, key, val, sn) from some replica c'
13: $map.update(key, val)$ {asynchronous total order delivery}
14: $maxsn \leftarrow maxsn + 1$

Algorithm 12 REP_SC (code for server s)

Local Variables:
1: $abCtr$ {counter to assign total order sequence numbers}

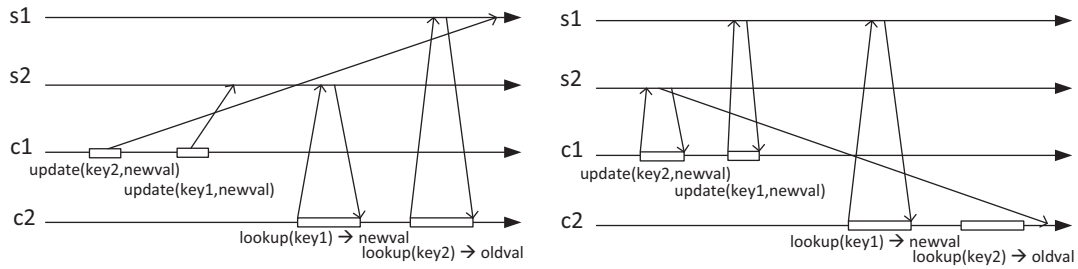
2: **upon rcv** $(SNREQ, key)$ **from some replica** c
3: $abCtr \leftarrow abCtr + 1$
4: $send(abCtr)$ *to* c

Figure 3.16: Sequential consistent replicated map

The replication algorithm based on two phase commit cannot exploit the sequential consistency for update operations: still a two phase commit protocol is needed to avoid conflicts and to provide a total order among updates. However lookups can return immediately. Since in our analysis, we are interested in the maximum throughput of each algorithm, the variant of replication based on two phase commit cannot provide better throughput compared to the linearizable one. Therefore we ignore the sequentially consistent variant of this protocol.

II) Partitioned map

To exploit sequential consistency for partitioning solutions, one can think of two optimizations: (1) To make the clients return immediately after sending the update message to the server (which applies to both PART_SIMPLE and PART_CACHING), and (2) to make the server to return immediately after broadcasting invalidation messages to the clients who hold a cached value of a key (which only applies to PART_CACHING). In case of having only one server, both optimizations can be applied and resulting algorithms are sequentially consistent. However since sequential consistency is not compositional, having more than one server can break sequential consistency in both cases as they are shown in Figures 3.17(a) and 3.17(b). In the first case, consider the PART_SIMPLE or PART_CACHING in a scenario mentioned in Figure 3.17(a). The issued update by client c_1 on key_2 arrives to the corresponding server after a long delay. After returning from the first update, it issues another update on key_1 to server s_2. Afterwards



(a) Case 1: PART_SIMPLE (b) Case 2: PART_CACHING

Figure 3.17: Impossibility of exploiting sequential consistency for partitioning algorithms

client c_2 issues a lookup on key_1, which arrives at s_2 after updating the local value of key_1 to $newVal$, as well as a lookup on key_2 which arrives at s_1 before updating the local value of key_2 to $newVal$. Since the first lookup on key_1 returns $newval$ and the second lookup on key_2 returns $oldval$, it is not possible to create a global history of operations complying with the returned values. In the second case, consider PART_CACHING in a scenario mentioned in Figure 3.17(b). Assume server s_2 needs to invalidate client c_2 upon receiving an update message on key_2 from c_1. Suppose it takes a long time for this invalidation message to arrive at c_2. Client c_1 issues another update after the first one, which updates the value of key_1 on server s_1 to $newval$. Later client c_2 issues a lookup on key_1 to server s_1, which returns $newval$, while the second lookup on key_2 is done from the local cache, since c_2 has not yet received the invalidation message from server s_2. In this case also it is not possible to create a valid global history of these operations.

To apply those optimizations to the partitioning algorithms with more than one server, one might come up with solutions which need extra communication, the case we want to avoid. Therefore these two optimizations can be applied only in the case of having one server. Even in the case of having only one server, these optimizations in practice require some flow control mechanisms to avoid buffers to overflow when updates and invalidations are sent repeatedly to the servers and the clients. Implementing a flow control mechanism to avoid buffer overflow can decrease the anticipated performance. We conclude that there is no way to exploit sequential consistency for partitioning solutions to obtain a better maximum throughput compared to their linearizable counterparts.

3.5 Evaluation

In this section, we first model the communication performance of a Tilera TILE-Gx processor. Then we validate the model of our map algorithms on this platform. Finally, using this model, we conduct a detailed study of the performance of the partitioning and replication algorithms in a message-passing manycore. Throughout this section, we consider a map implemented using a hash table. This is representative of most map implementations [19, 60].

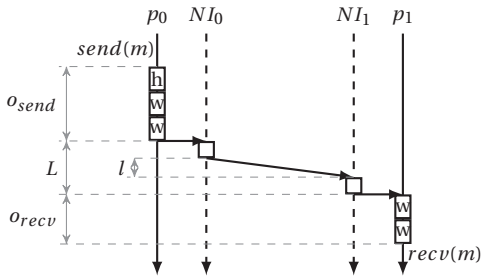


Figure 3.18: Point-to-point communication on the TILE-Gx for a 2-word message m (NI: network interface)

3.5.1 Modeling TILE-Gx8036

We run experiments on a Tilera TILE-Gx8036 processor. We use it as a representative of current message-passing manycore architectures [8]. Experiments are run with version 2.6.40.38-MDE-4.1.0.148119 of Tilera's custom Linux kernel. Applications are compiled using GCC 4.4.6 with O3 flag. To implement our algorithms, we use the User Dynamic Network (UDN). In our experiments, we dedicate one queue to *asynchronous* messages: An interrupt in generated each time a new message is available in this queue. Note that the TILE-Gx8036 processor does not provide support for collective operations. Hence, we implement $broadcast$ and $multicast$ as a set of $send$ operations. Such an implementation will be later replaced by a hardware-based broadcast service.

Figure 3.18 describes how we model a point-to-point communication on the TILE-Gx processor. The figure illustrates the case of a 2-word message transmission using $send$ and $recv$. This model is solely based on our evaluations of the communication performance and is only valid for small-sized messages. We do not claim that Figure 3.18 describes the way communication are actually implemented in the processor.

We obtain value of the TILE-Gx model parameters using some microbenchmarks. The overhead o_{send} of a message of n words includes a fix cost of 8 cycles associated with issuing a header packet, plus a variable cost of 1 cycles per word. The overhead o_{recv} is equal to 2 cycles per word. The header packet is not received at the application level. The transmission delay L between the sender and the receiver includes some fix overhead at the network engines on both the sender and the receiver, plus the latency l associated with network traversal. The fix overhead is 10 cycles in total. The latency l depends on the number of routers on the path from the source to the destination: 1 cycle per router. However, on a 36-core mesh the distance between processes has little impact on the performance. Thus, to simplify the study we assume that l is constant and is equal to the average distance between cores, *i.e.*, $l = 6$. Note that there is no *gap* between two consecutive messages sent by the same core. Moreover our measurements show that the cost of invoking an interrupt handler and restoring the previous context account for 138 cycles. As previously mentioned, we implement $broadcast$

Parameter \Platform	TILE-Gx	*Intermediate*	*Ideal*
o_{send}	$8+\|m\|$	-	-
o_{rcv}	$2\cdot\|m\|$	-	-
o_{arcv}	$138+o_{rcv}$	$4+o_{rcv}$	$4+o_{rcv}$
o_{bcast}	$c\cdot o_{send}$	-	o_{send}
o_{mcast}	$\|list\|\cdot o_{send}$	-	o_{send}
$T_{rtt}(send,rcv)$	$2\cdot(o_{send}+o_{rcv}+L)$	-	-
$T_{rtt}(send,arcv)$	$2\cdot(o_{send}+L)+o_{arcv}+o_{rcv}$	-	-
$T_{rtt}(bcast,arcv)$	$o_{bcast}+o_{arcv}+$ $o_{send}+o_{rcv}+2\cdot L$	-	-
$T_{rtt}(mcast,arcv)$	$o_{mcast}+o_{arcv}+$ $o_{send}+o_{rcv}+2\cdot L$	-	-
o_{sel}	17 if $s=2^x$, 90 otherwise	-	-
L	16	-	-

Table 3.2: Parameters value in cycles (A "-" means that the value is the same as on TILE-Gx)

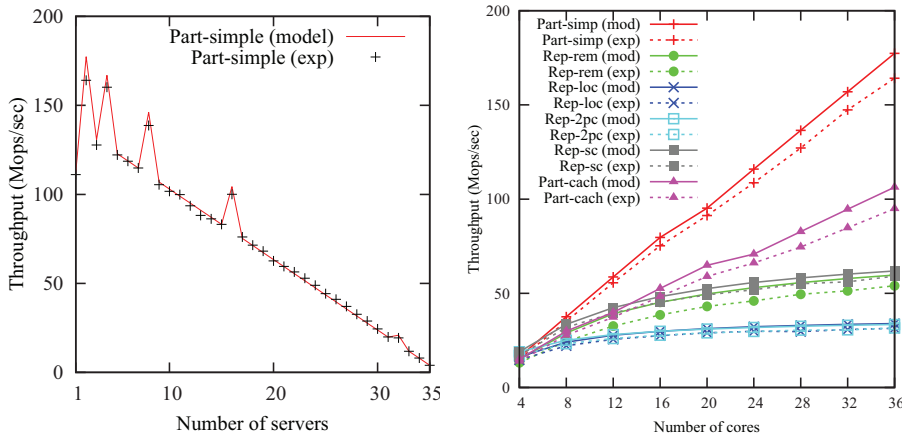
and $multicast$ operations as a sequence of $send$ operations. When the round-trip time is initiated with a collective operation, its duration corresponds to the time required to send all messages plus the time to receive the answer to the last message sent. Finally, we implement the server selection operation using the $modulo$ operation. Its cost o_{sel} varies depending whether the number of server is 2^x (in this case $modulo$ is implemented with a bit-wise AND) or not. The second column of Table 3.2 summarizes the value of the model parameters for the TILE-Gx processor.

3.5.2 Model validation

To validate our model, we run our algorithms on the TILE-Gx processor and compare the achieved throughput to the one predicted by the model. The experiment considers a hash table with keys of 36 bytes and values of 8 bytes. The DJB hash function, which generates 4 bytes long hash-keys, is used: o_{pre} = 156 cycles. The processes manipulate 100 keys, and so, we assume that the hash table fits into the L1 cache of the cores. Also, in all experiments we assume a collision free scenario. Thus, assuming that an access to the L1 cache is negligible, we have $o_{lup} = o_{upd} = 0$.

Threads are pinned to cores in ascending order: thread t_i is pinned to core i. Note that the size of the messages depends on the algorithms specification. For instance, in PART_SIMPLE, update requests sent to a server include 4 words: the id of the sender, the operation id, the hash-key, and the value. The answer is a one-word message containing simply the acknowledgment. The messages size is taken into account for the modeling.

The results presented in Figure 3.19 assume a load of 90% of lookups ($p = 0.9$). Each point is the average throughput of 6 runs, where in each run every core issues 10000 operations repeatedly on the map. Client threads randomly choose the next operation to execute with a uniform distribution. Keys are distributed among the servers uniformly. Similarly, clients randomly select the key for the next operation with a uniform distribution, *i.e.*, $pmf_{key}(k) = 1/c$. Figure 3.19(a) shows the variation of the throughput with PART_SIMPLE when the total



(a) Part-simple on 36 cores

(b) Algorithms performance

Figure 3.19: Model validation on Tilera TILE-Gx processor (90% of lookup operations)

number of threads is 36 and when the number of server threads varies from 1 to 35. It compares the performance obtained through experiments (dots) and predicted by the model (line). It first shows that the model manages to precisely estimate the performance of the algorithm. The *hiccups* that can be observed are due to the cost of the $modulo$ function used for server selection, and correspond to cases where the number of servers is 2^x. Both the experiments and the model show that the optimal configuration in this case is with 2 servers.

Figure 3.19(b) presents the maximum throughput of the different algorithms when varying the total number of threads. To obtain this graph, for each case we run the same test as described by Figure 3.19(a), and we take the best configuration. This figure shows that we manage to correctly model the performance trends of the algorithms executing on the TILE-Gx processor. Also, it shows that the throughput obtained with the model is always higher than the experimental one. This is expected since the model ignores some computational costs (*e.g.*, operations on private variables) related to the implementation of the algorithms. Additionally, the model considers the maximum *overlapping* $\mathscr{O}^c$ between idle periods and interrupts handling, which is most probably less during experiments. Hence, the model provides an upper bound on the performance of the algorithms, which is at the same time not far from the actual performance. PART_CACHING is the algorithm for which the difference between the model and the experiments is the highest. But even in this case, the difference is at most 12%. Finally, note that in this experiment PART_SIMPLE always outperforms the other solutions. This might be due to the high cost of interrupt handling as well as non-efficient broadcast service, which penalizes the other algorithms. Hence these results could not be generalized.

3.5.3 Analysis of the map algorithms

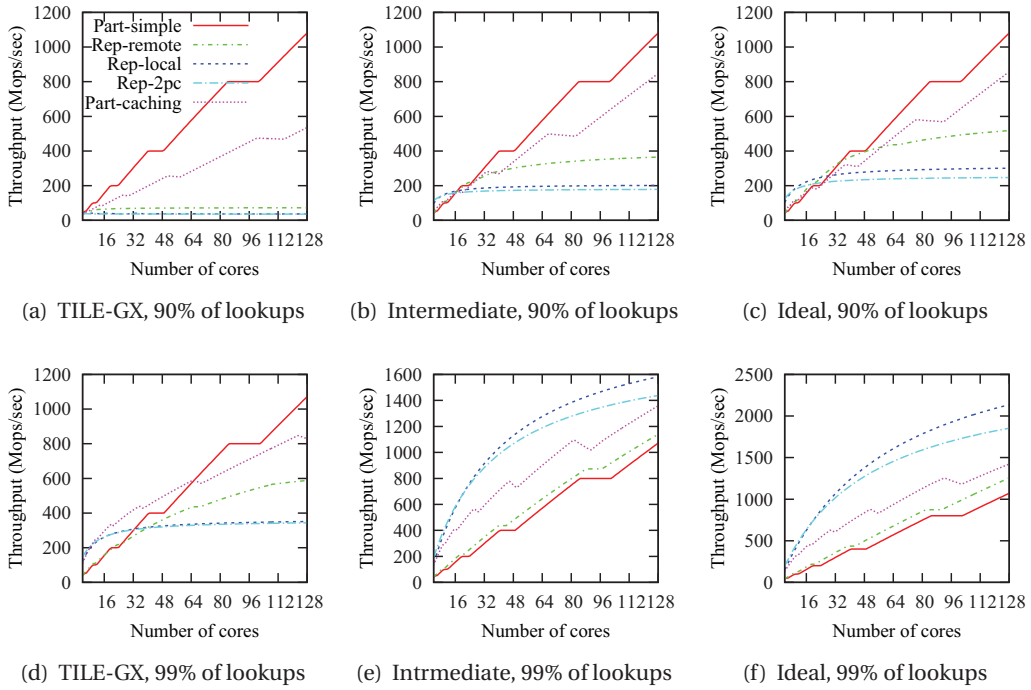
Analytical modeling helps us to do the comparative study of different algorithms under different settings and loads, *e.g.* where the target platform has different architectural features or the load distributions are not uniform. Moreover it helps us to concretely understand the performance bottlenecks of different algorithms. Using our model, we analyze the performance of partitioning and replication algorithms under different settings. To assess the performance on current and future platforms, we consider two features, not provided by the TILE-Gx processor, that can be blamed for the poor performance of applying the replication paradigm.

The first feature is non-efficient broadcast service on Tile-Gx. Due to the lack of a hardware-based broadcast service on this platform, broadcasting to n participants consumes cpu time of n sends and n receives. Note that even the most efficient software implementation of broadcast on top of send and receive primitives, leads to the consumption of the mentioned amount of cpu time [5]. Some recent architectures implement the broadcast service in hardware, *e.g.* Kalray MPPA [4], Adapteva Epiphany [1] and Picochip DSP [7]. To model this feature on these platforms, we assume that the overhead of $broadcast$ and $multicast$ is the same as the overhead of a $send$, which would be the ideal case. Second, even if interrupt handling on the TILE-Gx is rather efficient, its overhead remains high compared to other cpu costs. Solutions have been proposed to save and restore an execution context very efficiently using different architectural and compilation techniques [88, 111, 37, 95]: More specifically in [88], a solution with a constant 4 cycles cost is presented. Hence the second feature we consider is efficient interrupt handling with a cost of only 4 cycles.

In order to assess the affect of the mentioned features on the comparative performance of different algorithms, we incrementally define two platforms which do not suffer from them. First we define an *intermediate* platform that has the same characteristics as the TILE-Gx processor but provides efficient asynchronous receives (see Table 3.2, *intermediate* platform). Second we define an *ideal* platform that has the same characteristics as the *intermediate* platform but also provides hardware-based broadcast service (see Table 3.2, *ideal* platform).

Considering a hash table implementation of a map, we compare the algorithms on the mentioned three platforms for different ratio of lookup operations. We assume a collision free scenario in order to not to deal with other orthogonal issues. First under the same consistency criteria, *i.e.* linearizability, we compare the performance of different algorithms on the three platforms for a given use case, *i.e.*, we fix the cost of the hash function and the cost of accessing the hash table. Second, we study how the cost of the hash function and of the hash table accesses impact the performance. Third, we focus on the PART_CACHING algorithm and analyze how the probability distribution of client access to the keys affects its performance. Fourth, we study how weakening the consistency criteria to sequential consistency could be in favor of replication. Fifth, we assess the effects of colocating clients and servers on the same core on

[5] When broadcast is implemented using asynchronous communication, the throughput of the system is independent from the broadcast algorithm [84].



(a) TILE-GX, 90% of lookups (b) Intermediate, 90% of lookups (c) Ideal, 90% of lookups

(d) TILE-GX, 99% of lookups (e) Intrmediate, 99% of lookups (f) Ideal, 99% of lookups

Figure 3.20: Performance on the three platforms ($o_{pre} = 12$, $o_{op} = 11$)

the performance of the algorithms. Finally we calculate how non-uniform load distribution on the servers can impair the maximum obtainable throughput.

Comparison of the three platforms

Figure 3.20 shows the performance of the different linearizable algorithms as a function of the total number of cores when the percentage of lookups is 90% and 99%, representative loads of many map use-cases [49, 14]. The assumptions made in this evaluation are: i) keys are integers and a simple shift-add hash function is used, *i.e.*, $o_{pre} = 12$; ii) the hash table is small enough to fit into the L2 cache of one core, *i.e.*, we assume that accesses to the hash table cost one L2 access ($o_{op} = 11$)[6]; iii) clients randomly select the key for the next operation with a uniform distribution, *i.e.*, $pmf_{key}(k) = 1/c$. Note that the uniform distribution can be considered as a worst case for PART_CACHING since it implies that the probability that one core issues many lookups on the same key is low. Later we see that a non-uniform key access distribution can improve the performance of PART_CACHING. The two first assumptions are representative of the use of maps in an operating system [60].

Three conclusions that can be drawn from Figure 3.20. First, if the ratio of lookups is not very high, then partitioning approaches outperforms replication at scale on all platforms (see Figures 3.20(a) to 3.20(c)). On the *ideal* platform, REP_LOCAL provides the best performance

[6]We prefer assuming L2 rather than L1, due to its bigger size.

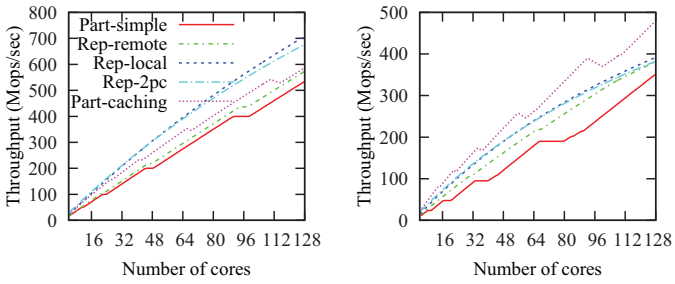
for 128 cores with 99% of lookups, but the minimum ratio of lookups for REP_LOCAL to be the most efficient in this case is actually 98%. However its throughput reaches a plateau if the total number of cores increases indefinitely. Second, on the TILE-Gx processor, partitioning outperforms replication even if the ratio of lookups is very high (see Figures 3.20(a) and 3.20(d)). Replication can outperform partitioning on TILE-Gx only if the lookups are less than 0.1% of the total number of operations. Third, the effect of having broadcast in hardware is much less than providing efficient asynchronous receives.

We explain now the shape of the curves with the partitioning algorithms. One can see plateau in the throughput of PART_SIMPLE. This is due to the variable cost of the modulo function used to select a server. At the beginning of a plateau, the optimal configuration requires 2^x servers. Then servers become the system bottleneck, and so, the number of servers should be increased. However, adding one server dramatically increases the cost of the modulo function and makes clients again the bottleneck. Hence, the maximum throughput remains constant despite the increase of the number of cores because the number of servers remains 2^x as long as there are not enough clients to afford having a more costly modulo function. The same phenomenon exists with PART_CACHING, but in this case it is even worse because adding more clients increase the cost of updates on the server (more invalidation messages are needed on average), leading to a performance decrease.

Impact of the computational costs

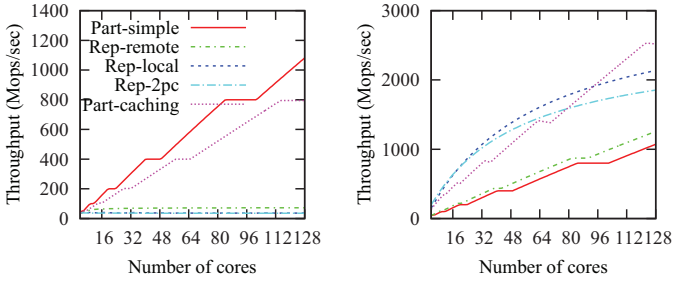
One might wonder if the results displayed in Figure 3.20 depend on the assumptions made on the map. Figure 3.21 shows the performance of the linearizable algorithms for other values of o_{pre} and o_{op}. To better assess the impact of these changes, we consider the ideal platform because the relative cost of these parameters is then higher compared to the communication costs. Additionally, we assume a load with 99% of lookups.

Figure 3.21(a) presents the performance when the hash function cost is 156 cycles, which is a typical cost for a hash function operating on strings. A comparison with Figure 3.20(f) shows that the maximum throughput of all algorithms decreases but that their relative performance does not change. Figure 3.21(b) presents the performance when the cost of the operations on the hash table is also increased to 88 cycles. It corresponds to the cost of an access to the main memory. This setting is representative of an in-memory key-value store [6]. In this case, the algorithms based on replication are mainly impacted because the cost of updating the hash table is higher. As a result, compared to Figure 3.20(f) where REP_LOCAL was providing the best results, PART_CACHING is now the most efficient algorithm. This is due to the fact that replicated maps are not able to leverage the locality if map replicas are not cached. Note that we do not present results for a configuration with a low hash function cost and a high operation cost because we could not find any corresponding use case.



(a) $o_{pre} = 156$, $o_{op} = 11$ (b) $o_{pre} = 156$, $o_{op} = 88$

Figure 3.21: Impact of the computational costs (*ideal* platform, 99% of lookups)



(a) Tilera, 90% of lookups (b) Ideal, 99% of lookups

Figure 3.22: Impact of the access pattern ($o_{pre} = 12$, $o_{op} = 11$)

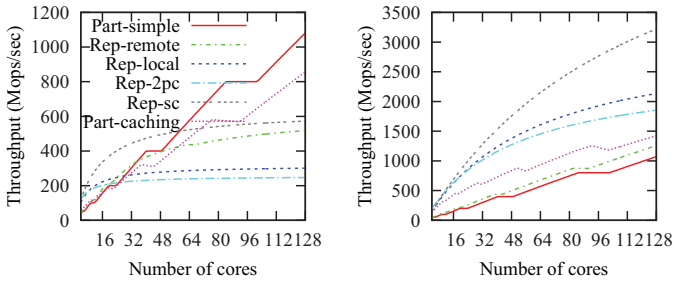
Performance of PART_CACHING with non-uniform client key access

All evaluations until now assume a uniform distribution of the probability for clients to access one key. This distribution has a negative impact on PART_CACHING since all clients may access a key, which minimizes the probability of local lookups. Moreover, it is not representative of many use cases where only on small number of clients issue most operations on a given key. To evaluate the performance of PART_CACHING in such a scenario, we define another distribution function where a fix number of clients c_{key} issue r% of the operations on a key.

Figure 3.22 shows the performance with $c_{key} = 4$ and $r = 80$. It considers TILE-Gx with 90% of lookups and the *ideal* platform with 99% of lookups. In both cases, the performance of PART_CACHING is greatly improved. In Figure 3.22(b), PART_CACHING even outperforms REP_LOCAL.

Impact of weakening consistency criteria to sequential consistency

As we discussed earlier, unlike partitioning solutions, replication solutions are able to exploit sequential consistency. As we saw earlier in comparing linearizable solutions, partitioning is the best approach unless three conditions are met: (i) the percentage of lookups are extremely



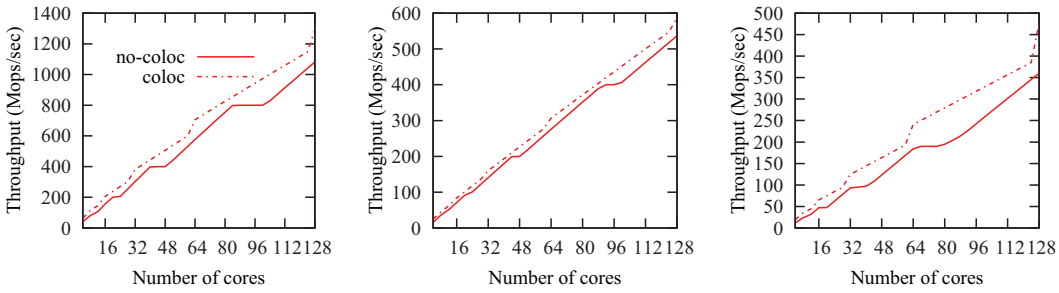
(a) ideal, 90% of lookups (b) Ideal, 99% of lookups

Figure 3.23: Impact of weakening consistency criteria ($o_{pre} = 12$, $o_{op} = 11$)

high; (ii) the cost of asynchronous receives are extremely low; and (iii) the map is located in the cache system of the cores. Provided that these conditions are met, replication can outperform partitioning. In order to understand up to which extent a weaker consistency criteria could be in favor of replication, we compare the performance of REP_SC with other linearizable solutions on the *ideal* platform, where $o_{pre} = 12$, $o_{op} = 11$, for both 90 and 99 percent of lookup workload. As you see in Figure 3.23(a), with 90 percent of lookups operations REP_SC still cannot beat partitioning solutions at scale, although it outperforms other replication solutions as expected. However as you see in 3.23(b), with 99 percent of lookups REP_SC outperforms all other solutions significantly. The threshold for percentage of lookup operations in which after that REP_SC outperforms all other algorithm at all scales, is around 95%. This threshold for REP_LOCAL is around 98%, which was mentioned earlier too. Therefore weakening consistency criteria although improves the performance of replication, but still the three conditions are necessary for replication to outperform partitioning, even though partitioning solutions are not able to exploit sequential consistency in their favor.

Colocating clients and servers on the same core

Our evaluations are based on the assumption that clients and servers are located on different cores. One can argue that placing clients and servers on the same core might lead to a better maximum throughput. In this case a core, while playing the role of a server, can receive the requests asynchronously. This strategy does not make sense on the TILE-Gx architecture since the relative high cost of asynchronous receive is added to the critical path of all operations. However considering the ideal platform, where the cost of asynchronous and synchronous receives are in the same order, it is not clear how this strategy can affect the maximum throughput. Therefore we use our model to obtain the maximum throughput in this case. For the sake of simplicity, we consider the simple partitioning algorithm, PART_SIMPLE. To obtain the maximum throughput of this algorithm, one can consider a total number of C cores partitioned into two sets: the first set S_1, with the size of $C - S$, are those who are purely clients and the second set S_2, with the size of S, are those who colocate clients and servers. To compute the maximum throughput, we obtain the maximum throughput of each set and sum



(a) $o_{pre} = 12, o_{op} = 11$ (b) $o_{pre} = 156, o_{op} = 11$ (c) $o_{pre} = 156, o_{op} = 88$

Figure 3.24: Impact of colocating clients and servers on ideal platform (PART_SIMPLE)

them up. The maximum throughput of S_1 is calculated in the same way as before:

$$\mathscr{T}^{S_1} = min\Big(\frac{C-S}{p \cdot T^c_{lup} + (1-p) \cdot T^c_{upd}}, \frac{S}{p \cdot T^s_{lup} + (1-p) \cdot T^s_{upd}}\Big) \tag{3.57}$$

Assuming that there is no request from the S_1 to the S_2, the obtainable throughput from S_2 is equal to:

$$\mathscr{T}^{S_2*} = \frac{S}{p \cdot (T^c_{lup} + T^s_{lup}) + (1-p) \cdot (T^c_{upd} + T^s_{upd}) - \mathscr{O}^c} \tag{3.58}$$

However this throughput cannot be obtained from S_2, since a portion of each cpu time during one second is devoted to serve the requests which were received from the cores in S_1[7]. This means that $\mathscr{T}^{S_2} = (1-\mathscr{L}) \cdot \mathscr{T}^{S_2*}$, where $\mathscr{L}$ is the portion of cpu time of each core in S_2, is devoted to serve the requests received from the cores in S_1. $\mathscr{L}$ can be calculated from $\mathscr{T}^{S_1}$ as follows:

$$\mathscr{L} = \frac{\mathscr{T}^{S_1} \cdot \big(p \cdot T^s_{lup} + (1-p) \cdot T^s_{upd}\big)}{S} \tag{3.59}$$

Considering the above formula, we obtained the maximum achievable throughput of PART_SIMPLE with colocating clients and server in Figure 3.24. Considering all three use cases, the performance improvement is at most 20 percent. Analysis of other algorithms show that their performance improvement by colocating clients and servers does not exceed 20 percent.

[7] For simplicity, this calculation assumes the idle time during each request issued by the clients in S_2, cannot be used to serve the requests issued from the clients in S_1. The exact formula will be much more complex.

Non-uniform load distribution on the servers

In calculating throughput of all algorithms, we assumed that the clients uniformly access the servers. However non-uniform distribution of the keys among servers can affect the maximum obtainable throughput. This non-uniform distribution can be due to different reasons depending on the implementation of the map. For example if the map is implemented using a hash table, a non-uniform hash function can create non-uniform load on different servers. Another example is a name service to track different services in a factored operating system, implemented using a table. If some services are accessed more often than the others, it can also create a non-uniform load among the servers. We calculate the maximum obtainable throughput of our algorithms for a non-uniform load on the servers, given an arbitrary load distribution among them.

If we consider an arbitrary load among s servers, it can clearly affect the throughput of the system when the servers are the bottleneck. However in case that the clients are the bottleneck, the throughput of the system remains as before. Consider an arbitrary load where server s_i is accessed with the probability of p_i, where $\sum_{i=1}^{s} p_i = 1$. Now assume that the server(s) with maximum load is(are) accessed with the probability of p_{max}. Therefore the load on any other server is a fraction of p_{max} such that $p_i = p_{max} \cdot k_i$ where $0 \leq k_i \leq 1$. Since the server with the maximum load would be the bottleneck for the throughput of the servers, the total throughput of the servers is equal to:

$$\mathscr{T}^s = \sum_{i=1}^{s} k_i \cdot \frac{1}{p \cdot T^s_{lup} + (1-p) \cdot T^s_{upd}} \tag{3.60}$$

Clearly the uniform distribution leads to the highest server throughput ($k_i = 1$). The negative effects of non-uniform distribution threatens partitioning solutions more than the replication ones, since replication algorithms are less sensitive, if not non-sensitive, to the changes in the distribution of the load on the servers.

3.5.4 Discussion

Results show that the only situation where replication could be used to implement a high throughput linearizable map on a message-passing processor is when the percentage of lookups is extremely high, the processor provides features such as highly efficient interrupt handling and the map is located in the cache system of the cores. In this case, REP_LOCAL could be efficient but the REP_REMOTE approach is not interesting because of the high cost of its lookup operation.

Althought the map algorithms designed for shared memory architectures mostly ensure linearizability [49], to assess the effects of weakening the consistency criteria, we also study the case of sequential consistency. Replicated maps are able to exploit sequential consistency

by removing the synchronization between lookups and updates. On the contrary partitioned maps are not able to exploit sequential consistency, mainly because sequential consistency is not compositional. Evaluations show that replication still needs the same conditions as with the case of linearizability to outperform partitioning. Study of even weaker consistency criteria [107], using a similar methodology, can complement this study.

Clients and servers can be colocated on the same core. This configuration avoids dedicating resources to play the server role. On the TILE-Gx, this is not a desirable choice since a costly asynchronous receive will be involved in every request sent to the servers. Evaluations on the *ideal* platform show that, despite efficient asynchronous receives, this colocation only leads to a negligible performance gain. The main reason is that in the best configurations, the number of servers that can be colocated with the clients is small.

Client can access the servers non-uniformly, *e.g.* when the map is implemented using a hash table with a non-uniform hash function. This non-uniformity decreases the throughput of the servers, and consequently of the overall map (except for REP_2PC). Moreover a non-uniform access of the clients to different keys increases the throughput of the PART_CACHING algorithm, by increasing the probability of local lookups and decreasing the number of invalidations. For a given distribution of the client accesses among servers and the key accesses among clients, throughput of the maps can be quantified using our model. Evaluations considering realistic load distributions based on real case scenarios can be an interesting extension of this work.

We considered the TILE-Gx, a general purpose message-passing manycore, as the baseline for our evaluations. We believe that our conclusions remain valid on similar architectures since: (i) TILE-Gx provides efficient inter-core communication; (ii) using our model we could consider cases where broadcast operations and asynchronous receives are very efficient. Still, using our model, one can directly do a comparison on other architectures. One exception is the architectures with one-sided communication primitives, *e.g.* Intel SCC [52]. The main reason is that inter-core communication in these architectures involves some synchronization costs [83] which are not included in our model.

3.6 Related Work

This chapter uses performance modeling to compare different algorithms. A few recent studies have proposed performance models for other manycore architectures [83, 89]. Our approach is similar to the one used in these papers. They all cover the same communication scenarios as the LogP model [34] (or its extensions) that is commonly used in message-passing systems. The main difference is that the underlying communication system considered in these studies are different from the one of this chapter: [83] models RMA-based communication and targets the Intel SCC processor; [89] models point-to-point communication on top of cache-coherent shared memory and targets the Intel Xeon Phi processor.

The implementation of scalable data structure in message-passing manycore is an impor-

tant research topic for message-passing-based operating systems[16, 109, 42]. The Barrelfish operating system [16] considers the operating system as a distributed system of cores, communicating using message passing, where the state is replicated instead of shared. Hence any potentially shared data structure is considered as if it is a local replica. Consistency among the replicas is maintained by exchanging explicit messages. The authors claim to improve scalability by applying replication is based on reducing the traffic on the interconnect, memory contention, synchronization overhead and access latencies. On the other hand, use of a client-server approach on chip level, is on the rise. The Fos [109] operating system applies a model, where the operating system is factored into function specific services, where each service is provided by a set of cores, so called fleets. Cores communicate with fleets using only messages. Fleets behave similar to Internet servers, which allowed them to scale up to millions of machines, but instead of web pages they provide traditional kernel operations and data structures. Fleets can internally apply different techniques, *e.g.* partitioning, to improve their performance. As an interesting use-case, the implementation of a naming service for the FOS operating system has been studied in [18]. The naming service is based on a hash map which is made scalable using replication. The replication algorithm used is this study is similar to REP_2PC but is not compared to other approaches. Partitioning and replication were both originally proposed as a mean to scale the operating system in the Tornado project [42]. The Tornado project targets NUMA machines where remote memory accesses are an order of magnitude more costly than local accesses. Since Tornado was designed for shared-memory processors, message-passing was emulated in software with a high cost for software-based multicast operations. We compared partitioning and replication in the context of modern message-passing manycore chips, which provide completely different trade-offs regarding communication performance compared to [42].

Optimization of in-memory key-value stores for manycore is an area where our results could be applied [19, 78]. The authors of [19] and [78] both propose a partitioning approach similar to the PART_SIMPLE algorithm. The solution proposed in [78] is based on message-passing emulated on top of shared memory whereas [19] takes advantage of hardware message-passing provided by Tilera. This chapter complements these studies by providing a comparison between partitioning and replication.

4 Conclusion

Manycore architectures, in which a large number of general-purpose processing cores are fabricated into a single chip, provide a high level of parallel processing power while their energy consumption is considerably lower than their multi-chip counterparts. Although shared-memory programming is the classical paradigm to program manycore environments, there are several claims that taking into account the full life cycle of software, the message-passing programming model has numerous advantages. This already led to modern manycore chips with message-passing support in hardware. These platforms can be seen in two ways: (i) as a HPC cluster programmed by highly trained scientists using MPI libraries; or (ii) as a mainstream computing platform requiring a global operating system to abstract away the architectural complexities from the ordinary programmer. Each approach faces with the performance bottlenecks caused by MPI communication primitives and kernel data structures respectively. This thesis studies the mentioned bottlenecks in the context of high-performance broadcast communication primitive and map data structure on modern message-passing manycores in two different chapters.

In one chapter, we proposed OC-Bcast as a pipelined k-ary tree broadcast algorithm based on one-sided communication. It is designed to leverage the inherent parallelism of on-chip RMA in manycores. Experiments on the SCC show that OC-Bcast outperforms the state-of-the-art broadcast algorithms on this platform. OC-Bcast provides around 3 times better peak throughput and improves latency by at least 27%. These performance gains are mainly due to a limited number of off-chip data movements on the critical path of the operation: one-sided operations allow to take full advantage of the on-chip MPBs.

In the other chapter, we studied the implementation of strongly-consistent maps in message-passing manycores. Using a communication model we compare the performance of partitioned and replicated maps under different settings. A Tilera TILE-Gx8036 processor is used to validate the model and serves as a baseline for the evaluations. The results show that replication can outperform partitioning only if handling interrupts is highly efficient, update operations are rare and map replicas are located in the cache system of the cores.

This thesis clearly shows that hardware-specific features should be taken into account to design efficient algorithms for manycore architectures. Moreover, it highlights the importance of analytical modeling for concrete understanding of complex phenomenons as well as predicting behaviour of algorithms beyond existing platforms. Immediate extensions of this thesis include the followings:

- *Studying other MPI collectives in similar architectures with on-chip RMA:* in this thesis, we took advantage of RMA access to the on-chip message-passing buffers of the cores to implement a high-performance broadcast. However other collective operations involving one-to-many messaging components can benefit from similar techniques. Examples of such collectives include: barriers, scatter and all-to-all operations. Providing an RMA-aware library to run communication-heavy macrobenchmarks on similar architectures can highlight the benefits of our approach at a higher level.
- *Taking into account topology information:* in our analysis, we ignored the hop latencies among cores assuming the same distance between each pair. This simplifying assumption is rather realistic: in platforms which are considered in this thesis, the difference between the latency of sending a message to the furthest core compared to the neighboring core is at most 30%. However by increasing the number of cores, topology information and actual hop distances introduce interesting optimization problems regarding creation of broadcast trees as well as placement of the servers with respect to the clients.
- *Considering realistic distribution of key accesses and server loads:* to compare replication and partitioning, we assumed a uniform distribution of client accesses to the servers and to the different keys. However in real world scenarios, uniform accesses might not be the case. One such scenario is when a map is implemented using a hash table with a non-uniform hash function. Another scenario can happen when certain keys are accessed only by a certain number of clients, *e.g.* in the context of a name service in which some services are accessed only by certain cores. Such distributions can affect the map performance in both positive and negative ways. Although for a given distribution, throughput of the map can be quantified using our model but evaluations considering realistic scenarios, *e.g.* access patterns inside manycore operating systems, can lead to more concrete results.
- *Studying map implementations under weak consistency criteria:* to compare partitioning and replication, we considered strong consistency criteria, *e.g.* linearizability and sequential consistency. However such a criteria might be heavier than what is actually needed, a case already true for many internet-scale services. On a chip scale, such a relaxation is already applied in implementation of some proposed name services for manycore operating systems. Weakening consistency can be in favor of both partitioning and replication, provided that they are fundamentally able to exploit such a relaxation. Systematic study of weak consistency criteria, similarly to the approach of this thesis for strong consistency, can complement this work.

- *Studying abstract data types which are not naturally partitionable:* in this work, we compared replication and partitioning in the context of a map data structure. However advantages of partitioning can diminish in the case of an abstract data type in which its individual items are not independent from each other, *e.g.* trees. In the case of a tree, parent-child relationships must be kept among different partitions using additional protocols. For such data types, provided that read operations are frequent enough, replication might outperform partitioning in a broader domain.

And at the end, comparing communication primitives and data structures based on the message-passing programming model with their counterparts based on the shared-memory programming model seems to be an interesting long-term research direction. These comparisons can be facilitated using architectures in which direct hardware support for both programming models is provided (*e.g. Tilera TILE*) and might lead to interesting hybrid solutions which take advantage of both programming models.

Bibliography

[1] Adapteva. http://www.adapteva.com/.

[2] Barrelfish operating system. http://www.barrelfish.org.

[3] Intellasys. http://www.intellasys.net/.

[4] Kalray. www.kalray.eu.

[5] Linux symposium. http://www.linuxsymposium.org/.

[6] Memcached. www.memcached.org.

[7] Picochip. http://www.picochip.com/.

[8] Tilera. www.tilera.com.

[9] Xpmem cross-process memory mapping. https://code.google.com/p/xpmem/.

[10] Anant Agarwal and Mathews Cherian. *Adaptive backoff synchronization techniques*, volume 17. ACM, 1989.

[11] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 253–262, 2005.

[12] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, 1990.

[13] InfiniBand Trade Association. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[15] P Aublin, S Mokhtar, Gilles Muller, and Vivien Quéma. Zimp: Efficient intercore communications on manycore machines. Technical report, Available: http://proton. inrialpes. fr/ aublin/zimp/zimp TR. pdf.

[16] 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*, SOSP '09, pages 29–44, 2009.

[17] O. Beaumont, L. Marchal, and Y. Robert. Broadcast Trees for Heterogeneous Platforms. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '05, pages 80–92, 2005.

[18] Nathan Beckmann. Distributed naming in a factored operating system. Master's thesis, Massachusetts Institute of Technology, 2010.

[19] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, pages 1–8, 2011.

[20] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, 2007.

[21] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[22] S. Boyd-Wickizer, A.T. Clements, Y. Mao, A. Pesterev, M.F. Kaashoek, R.T. Morris, N. Zeldovich, et al. An analysis of Linux scalability to many cores. 2010.

[23] 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.

[24] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. Smartmap: Operating system support for efficient data sharing among processes on a multi-core processor. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[25] Greg Bronevetsky, Andrew Friedley, Torsten Hoefler, Andrew Lumsdaine, and Dan Quinlan. Compiling mpi for many-core systems. Technical report, LLNL-TR-638557, Lawrence Livermore National Laboratory, 2013.

[26] Jehoshua Bruck, Ching-Tien Ho, Eli Upfal, Shlomo Kipnis, and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8:1143–1156, November 1997.

[27] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

[28] E. Chan. RCCE comm: A Collective Communication Library for the Intel Single-chip Cloud Computer. http://communities.intel.com/docs/DOC-5663, 2010.

[29] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and Harold Kuhn. Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multi-clusters. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 353–360. ACM, 2006.

[30] C. Clauss, S. Lankes, J. Galowicz, and T. Bemmerl. iRCCE: a non-blocking communication extension to the RCCE communication library for the Intel Single-chip Cloud Computer. http://communities. intel. com/docs/DOC-6003, 2011.

[31] Jimmy Cleary, Owen Callanan, Mark Purcell, and David Gregg. Fast asymmetric thread synchronization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):27, 2013.

[32] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[33] Travis Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report 93-02-02, University of Washington, Seattle, Washington, 1994.

[34] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, 1993.

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

[36] Bill Devlin, Jim Gray, Bill Laing, and George Spix. Scalability terminology: Farms, clones, partitions, and packs: Racs and raps. Technical report, MS-TR-99-85, Microsoft Research, Redmond, CA, 1999.

[37] Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):36, 2013.

[38] Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 325–334. ACM, 2011.

[39] Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266. ACM, 2012.

[40] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. Hybrid mpi: efficient message passing for multi-core systems. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 18. ACM, 2013.

[41] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[42] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 87–100, 1999.

[43] Corey Gough, Suresh Siddha, and Ken Chen. Kernel scalability—expanding the horizon beyond fine grain locks. In *Proceedings of the Linux Symposium*, pages 153–165, 2007.

[44] R. Graham and G. Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140, 2008.

[45] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, 1990.

[46] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[47] Rinku Gupta, Pavan Balaji, Dhabaleswar K. Panda, and Jarek Nieplocha. Efficient Collective Operations Using Remote Memory Operations on VIA-Based Clusters. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 46–62, 2003.

[48] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[49] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[50] Rick C. Hodgin. Inside intel's tera-scale project. http://www.tgdaily.com/hardware-features/33657-background-inside-intels-tera-scale-project, 2007.

[51] T. Hoefler, C. Siebert, and W. Rehm. A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium*, IPDPS '07, page 232, 2007.

[52] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, and et al. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *2010 IEEE International SolidState Circuits Conference*, pages 108–109. IEEE, 2010.

[53] Emmanuel Jeannot and Guillaume Mercier. Near-optimal placement of mpi processes on hierarchical numa architectures. In *Euro-Par 2010-Parallel Processing*, pages 199–210. Springer, 2010.

[54] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3(4):267–286, 1979.

[55] Philip N Klein, Robert HB Netzer, and Hsueh-I Lu. Detecting race conditions in parallel programs that use semaphores. *Algorithmica*, 35(4):321–345, 2003.

[56] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable rea, der-writer lock. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 201–204. IEEE, 1993.

[57] Rakesh Kumar, Timothy G Mattson, Gilles Pokam, and Rob Van Der Wijngaart. The case for message passing on many-core chips. In *Multiprocessor System-on-Chip*, pages 115–123. Springer, 2011.

[58] Sumeet S Kumar, Mitzi Tjin A Djie, and Rene Van Leuken. Low overhead message passing for high performance many-core processors. In *Computing and Networking (CANDAR), 2013 First International Symposium on*, pages 345–351. IEEE, 2013.

[59] S. Lankes, P. Reble, C. Clauss, and O. Sinnen. The Path to MetalSVM: Shared Virtual Memory for the SCC. In *The 4th symposium of the Many-core Applications Research Community (MARC)*, page 7, 2011.

[60] Chuck Lever and Sun-Netscape Alliance. Linux kernel hash table behavior: analysis and improvements. *Ann Arbor*, 1001:48103–4943, 2000.

[61] Shigang Li, Torsten Hoefler, and Marc Snir. Numa-aware shared-memory collective communication for mpi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96. ACM, 2013.

[62] N. Linnenbank, F. Reader, A.S. Tanenbaum, and D. Vogt. Implementing MINIX on the Single Chip Cloud Computer. www.nieklinnenbank.nl/download/scc.pdf, 2011.

[63] Jiuxing Liu, Amith R. Mamidala, and Dhabaleswar K. Panda. Fast and Scalable MPI-Level Broadcast Using InfiniBand's Hardware Multicast Supportsch. In *Proceedings of the 18th*

International Symposium on Parallel and Distributed Processing, IPDPS '04, page 10, 2004.

[64] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 295–304, 2003.

[65] 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.

[66] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, Gilles Muller, et al. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 Usenix Annual Technical Conference*, pages 65–76, 2012.

[67] Sheng Ma, Natalie Enright Jerger, and Zhiying Wang. Supporting efficient collective communication in nocs. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[68] Teng Ma. Kernel-assisted mpi collective communication among many-core clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 741–745. IEEE, 2012.

[69] Teng Ma, Thomas Herault, George Bosilca, and Jack J Dongarra. Process distance-aware adaptive mpi collective communications. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 196–204. IEEE, 2011.

[70] Peter Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171. IEEE, 1994.

[71] A.R. Mamidala, R. Kumar, D. De, and DK Panda. Mpi collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 130–137. IEEE, 2008.

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

[73] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *ACM SIGOPS Operating Systems Review*, 26(2):108, 1992.

[74] T. Mattson and R. Van Der Wijngaart. RCCE: a Small Library for Many-Core Communication. http://techresearch.intel.com, 2010.

[75] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[76] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[77] John M Mellor-Crummey and Michael L Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *ACM SIGPLAN Notices*, volume 26, pages 106–113. ACM, 1991.

[78] Zviad Metreveli, Nickolai Zeldovich, and M Frans Kaashoek. Cphash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 319–320, 2012.

[79] Stéphanie Moreaud, Brice Goglin, Raymond Namyst, and David Goodell. Optimizing mpi communication within large multicore nodes with kernel assistance. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–7. IEEE, 2010.

[80] MPI Forum. MPI2: Extensions to the Message-Passing Interface. www.mpi-forum.org, 1997.

[81] S. Peter, A. Schpbach, D. Menzi, and T. Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*, 2011.

[82] Darko Petrović, Thomas Ropars, and André Schiper. Leveraging hardware message passing for efficient thread synchronization. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 143–154, New York, NY, USA, 2014. ACM.

[83] Darko Petrović, Omid Shahmirzadi, Thomas Ropars, and André Schiper. High-performance rma-based broadcast on the intel scc. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 121–130, 2012.

[84] Darko Petrović, Omid Shahmirzadi, Thomas Ropars, André Schiper, et al. Asynchronous broadcast on the intel scc using interrupts. In *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, pages 24–29, 2012.

[85] Thomas Preud'homme, Julien Sopena, Gael Thomas, and Bertil Folliot. BatchQueue: Fast and Memory-Thrifty Core to Core Communication. In *Proceedings of the 2010 22nd International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '10, pages 215–222, 2010.

[86] James Psota and Anant Agarwal. rmpi: Message passing on multicore processors with on-chip interconnect. In *High Performance Embedded Architectures and Compilers*, pages 22–37. Springer, 2008.

[87] Rolf Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the message passing interface developer's and user's conference*, volume 1999, pages 77–85, 1999.

[88] NI Rafla and Deepak Gauba. Hardware implementation of context switching for hard real-time operating systems. In *54th IEEE International Midwest Symposium on Circuits and Systems*, 2011.

[89] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent smp systems: a case-study with xeon phi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 97–108, 2013.

[90] R. Rotta. On efficient message passing on the intel scc. In *Proceedings of the 3rd MARC Symposium*, pages 53–58, 2011.

[91] Michael L Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 31–40. ACM, 2002.

[92] Nir Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, 2011.

[93] Akio Shimada, Balazs Gerofi, Atsushi Hori, and Yutaka Ishikawa. Pgas intra-node communication towards many-core architecture. In *Proceedings of PGAS: International Conference on Partitioned Global Address Space Programming Models*, Santa Barbara, California, USA, 2012.

[94] M. Shroff and R.A. Van De Geijn. CollMark: MPI collective communication benchmark. In *International Conference on Supercomputing 2000*, page 10, 1999.

[95] Jeffrey S Snyder, David B Whalley, and Theodore P Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, 1995.

[96] Matthew J Sottile, Timothy G Mattson, and Craig E Rasmussen. *Introduction to concurrency in programming languages*. CRC Press, 2011.

[97] S. Sur, U. K. R. Bondhugula, A. Mamidala, H. W. Jin, and D. K. Panda. High performance RDMA based all-to-all broadcast for infiniband clusters. In *Proceedings of the 12th international conference on High Performance Computing*, HiPC'05, pages 148–157, 2005.

[98] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[99] Josep Torrellas. Architectures for Extreme-Scale Computing. *Computer*, 42(11):28–35, November 2009.

[100] Dean M Tullsen, Jack L Lo, Susan J Eggers, and Henry M Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 54–58. IEEE, 1999.

[101] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. RCKMPI - lightweight MPI implementation for intel's single-chip cloud computer (SCC). In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 208–217, 2011.

[102] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on Intel's single-chip cloud computer processor. *ACM SIGOPS Operating Systems Review*, 45(1):73–83, February 2011.

[103] Maarten van Steen and Guillaume Pierre. Replicating for performance: Case studies. In *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 73–89. 2010.

[104] Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck, and Chris R. Jesshope. Efficient Memory Copy Operations on the 48-core Intel SCC Processor. In *Proceedings of the 3rd MARC Symposium*, pages 13–18, 2011.

[105] Bryan Veal and Annie Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66. ACM, 2007.

[106] M. Verstraaten, C. Grelck, M.W. van Tol, R. Bakker, and C.R. Jesshope. Mapping distributed S-Net on to the 48-core Intel SCC processor. In *3rd MARC Symposium, Fraunhofer IOSB, Ettlingen, Germany*, 2011.

[107] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[108] S. Wallentowitz, M. Meyer, T. Wild, and A. Herkersdorf. Accelerating collective communication in message passing on manycore System-on-Chip. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 9–16. IEEE, 2011.

[109] David Wentzlaff and Anant Agarwal. Factored Operating Systems (FOS): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.

[110] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.

[111] Xiangrong Zhou and Peter Petrov. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proceedings of the 43rd annual Design Automation Conference*, pages 352–357. ACM, 2006.

Curriculum Vitae

EDUCATION

EPFL, Lausanne, Switzerland **Jul. 2014**
- PhD in Computer Science (Distributed Systems)

KTH, Stockholm, Sweden **Mar. 2008**
- M.Sc in Information Technology (Distributed Systems)

Amirkabir University of Technology, Tehran, Iran **Sep. 2006**
- B.Sc in Computer Engineering (Software)

PUBLICATIONS

- O. Shahmirzadi, T. Ropars and A. Schiper. High-Throughput Maps on Message-Passing Many-core Architectures: Partitioning versus Replication, In 20th International European Conference on Parallel Processing (EUROPAR), Porto, Portugal, August 2014.
- D. Petrovic, O. Shahmirzadi, T. Ropars, and A. Schiper. Asynchronous Broadcast on the Intel SCC using Interrupts. 5th Manycore Applications Research Community (MARC) Symposium, Toulouse, France, July 2012.
- D. Petrovic, O. Shahmirzadi, T. Ropars, and A. Schiper. High-Performance RMA-Based Broadcast on the Intel SCC. In 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Pittsburg, PA, USA, June 2012.
- A. Schiper, Z. Milosevic and O. Shahmirzadi. Student mini-kernel project based on an FPGA board, in ACM SIGOPS Operating Systems Review, vol. 45, num. 2, p. 54, 2011.
- O. Shahmirzadi, S. Mena and A. Schiper. Relaxed Atomic Broadcast: State-Machine Replication Using Bounded Memory. 28th IEEE International Symposium on Reliable Distributed Systems (SRDS), Niagara Falls, NY, USA, September 2009.