Efficient Communication and Synchronization on Manycore Processors

THÈSE N^{O} 6552 (2015)

PRÉSENTÉE LE 20 MARS 2015

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES RÉPARTIS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Darko PETROVIĆ

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury
Prof. A. Schiper, directeur de thèse
Prof. S. V. Adve, rapporteuse
Prof. P. Fatourou, rapporteuse
Prof. R. Guerraoui, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2015

To Iva and Ivana

Acknowledgements

Before writing these lines, as an obedient citizen willing to conform, I checked out what my fellow graduates usually write under this title. Doing some research on the acknowledgement sections of PhD theses reveals that there is a nice, established pattern to reuse. Following it, I would first and foremost thank my outstanding advisor for giving me all the research freedom in the world. Then I would carry on with a list of my great labmates, specially thanking those who had tons of patience for my stubbornness and imperfect French, followed by a list of my dear friends, where many would stand out for various reasons. A special thanks, of course, would go to the thesis jury members for taking their time to thoroughly review the pages to follow, as well as for providing many useful comments. Last but not least would be expressing my deepest gratitude to my beloved family that has always been there as an eternal source of inspiration, support, and understanding. But despite the urge I feel to elaborate on my thanks, the next few paragraphs will be somewhat different. They will summarize some important lessons I have learned during these years, which is *what* I am most thankful for. As for the *who* part, you people already know who you are and what you mean to me.

Lesson I: Being outsmarted is not a defeat. When this journey started, my belief was that doing a doctorate means being tucked in with your project for years, with no external factors that can influence the outcome of your work. It turns out that, as you are developing an interesting idea, odds are there are some smarter people who are working on the very same idea at the very same time. Seeing their shiny and polished results while your work is still half-baked can get quite frustrating. One of the things that doing a PhD teaches you is that, when this happens, you should not stop. On the contrary, pushing even harder is the way to go, as the reward might be just around the corner.

Lesson II: Criticism is invaluable. Investing much time and energy in rounding up a research task does not mean that the end result is impeccable. As a matter of fact, it never is, and it often has serious shortcomings. Fortunately, there have always been smart people able to spell them out for me, either in early phases, or later, as part of the reviewing process. However, it took me some time to learn how to fight my vanity and realize that the loads of criticism are not there to discourage me and discredit my work, but to help make it better (at least most of the time).

Lesson III: The presentation is as important as the content. I used to naïvely think that only the strength of the end result is important. The PhD has taught me that results rarely speak for themselves: Explaining them in a clear and structured way is equally important. It still strikes me how something that looks obvious to one person can be completely misinterpreted by another, even if they work on the very same topic. It is even more striking to what extent an introduction written differently can change people's perception about a piece of research.

It is not hard to see that all of these lessons pertain not only to research: They are easily extrapolated to life in general, which is one more reason never to forget them. Once again, a big thanks to all of you who have been part of this hard, but rewarding journey. There have been many bumps on the way here, but you were always there to keep me going. It will certainly feel nice being a doctor, but nothing can beat the feeling of being loved as a father, husband, son, brother, uncle, grandson, nephew, cousin, coworker, and friend.

Lausanne, February 23, 2015 Darko Petrović

Preface

This doctoral thesis is based on the following publications:

1. **[PSRS12b]**
   D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper. High-performance RMA-based Broadcast on the Intel SCC. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, Pittsburgh, Pennsylvania, USA, June 2012. URL: http://doi.acm.org/10.1145/2312005.2312029

2. **[PSRS12a]**
   D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper. Asynchronous Broadcast on the Intel SCC using Interrupts. In *The 6th Many-core Applications Research Community (MARC) Symposium*, Toulouse, France, July 2012. URL: https://hal.archives-ouvertes.fr/hal-00719022

3. **[PRS14]**
   D. Petrović, T. Ropars, and A. 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, Orlando, Florida, USA, February 2014. URL: http://doi.acm.org/10.1145/2555243.2555251

4. **[PRS15]**
   D. Petrović, T. Ropars, and A. Schiper. On the performance of delegation over cache-coherent shared memory. In *Proceedings of the 16th International Conference on Distributed Computing and Networking*, ICDCN '15, Goa, India, January 2015. URL: http://dx.doi.org/10.1145/2684464.2684476

In addition, an extended version of [PRS14] has been submitted, as an invited paper, to the ACM Transactions on Parallel Computing (TOPC).

Abstract

The increased number of cores integrated on a chip has brought about a number of challenges. Concerns about the scalability of cache coherence protocols have urged both researchers and practitioners to explore alternative programming models, where cache coherence is not a given. Message passing, traditionally used in distributed systems, has surfaced as an appealing alternative to shared memory, commonly used in multiprocessor systems. In this thesis, we study how basic communication and synchronization primitives on manycore processors can be improved, with an accent on taking advantage of message passing. We do this in two different contexts: (i) message passing is the only means of communication and (ii) it coexists with traditional cache-coherent shared memory.

In the first part of the thesis, we analytically and experimentally study collective communication on a message-passing manycore processor. First, we devise broadcast algorithms for the Intel SCC, an experimental manycore platform without coherent caches. Our ideas are captured by OC-BCAST (on-chip broadcast), a tree-based broadcast algorithm. Two versions of OC-BCAST are presented: One for synchronous communication, suitable for use in high-performance libraries implementing the Message Passing Interface (MPI), and another for asynchronous communication, for use in distributed algorithms and general-purpose software. Both OC-BCAST flavors are based on one-sided communication and significantly outperform (by up to 3x) state-of-the-art two-sided algorithms. Next, we conceive an analytical communication model for the SCC. By expressing the latency and throughput of different broadcast algorithms through this model, we reveal that the advantage of OC-BCAST comes from greatly reducing the number of off-chip memory accesses on the critical path.

The second part of the thesis focuses on lock-based synchronization. We start by introducing the concept of hybrid mutual exclusion algorithms, which rely both on cache-coherent shared memory and message passing. The hybrid algorithms we present, HYBLOCK and HYBCOMB, are shown to significantly outperform (by even 4x) their shared-memory-only counterparts, when used to implement concurrent counters, stacks and queues on a hybrid Tilera TILE-Gx processor. The advantage of our hybrid algorithms comes from the fact that their most critical parts rely on message passing, thereby avoiding the overhead of the cache coherence protocol. Still, we take advantage of shared memory, as shared state makes the implementation of certain mechanisms much more straightforward. Next, we try to profit from these insights even on processors without hardware support for message passing. Taking two classic x86 processors from Intel and AMD, we come up with cache-aware optimizations that improve the performance of executing contended critical sections by as much as 6x.

Keywords: multicore, manycore, concurrency, parallelism, HPC, broadcast, mutual exclusion.

Résumé

L'augmentation du nombre de coeurs intégrés sur une puce a amené de nombreux défis. La difficulté à rendre les protocoles de cohérence de cache scalables a conduit les chercheurs et les praticiens à explorer des modèles alternatifs de programmation, où la cache n'est pas forcément cohérente. L'échange de messages, traditionnellement utilisé dans les systèmes répartis, est apparu comme une alternative attrayante à la mémoire partagée, généralement utilisée dans le contexte des systèmes multiprocesseurs. Dans cette thèse, nous étudions comment la mise en oeuvre des primitives de base pour la communication et la synchronisation sur les processeurs multi-coeurs peuvent être améliorés, avec un accent sur l'utilisation de l'échange de messages. Nous faisons cela dans deux contextes différents : (i) l'échange de messages est le seul moyen de communication et (ii) l'échange de messages coexiste avec la mémoire partagée traditionnelle à caches cohérentes.

Dans la première partie de la thèse, nous étudions analytiquement et expérimentalement la communication collective sur un processeur multi-coeurs à échange de messages. Tout d'abord, nous proposons des algorithmes de diffusion pour Intel SCC, une plate-forme expérimentale multi-coeurs sans caches cohérentes. Nos idées sont exprimées dans OC-BCAST (On-Chip Broadcast), un algorithme de diffusion basé sur un arbre. Deux versions de OC-BCAST sont présentées : l'une pour la communication synchrone, utilisable dans les bibliothèques MPI (Message Passing Interface), et l'autre pour la communication asynchrone, pour utilisation dans les algorithmes répartis et les logiciels à usage général. Les deux versions de OC-BCAST sont basées sur la communication unilatérale (*one-sided communication*) et elles conduisent à des performances considérablement meilleures (jusqu'à trois fois) que celles des algorithmes classiques bilatéraux (*two-sided*). Ensuite, nous développons un modèle analytique des communications pour SCC. En exprimant la latence et le débit des différents algorithmes de diffusion grâce à ce modèle, nous montrons que le gain de performance provient d'une réduction considérable du nombre d'accès à la mémoire hors-puce sur le chemin critique.

La deuxième partie de la thèse est consacrée à la synchronisation à l'aide de verrous. Nous commençons par introduire le concept d'algorithme hybride pour l'exclusion mutuelle, qui utilise à la fois la mémoire partagée à caches cohérentes et l'échange de messages. Les algorithmes hybrides que nous présentons, à savoir HYBLOCK et HYBCOMB, ont des performances bien meilleures (jusqu'à quatre fois) que leurs homologues qui n'utilisent que la mémoire partagée, lorsqu'ils sont utilisés pour mettre en oeuvre, sur le processeur hybride Tilera TILE-Gx, compteurs, piles et queues concurrents. L'avantage de nos algorithmes hybrides provient du fait que les parties les plus critiques utilisent l'échange de messages, ce qui évite les coûts du protocole de cohérence de cache. En même temps nous pouvons profiter de la mémoire partagée, vu que l'existence d'un état partagé rend la mise en oeuvre de certains mécanismes bien plus simple. Nous exploitons également ces idées pour l'échange de messages sur les processeurs sans support matériel. En utilisant deux processeurs x86 classiques d'Intel et AMD, nous présentons des optimisations qui améliorent jusqu'à six fois les performances de l'exécution de sections critiques.

Mots clefs : multi-coeurs, concurrence, parallélisme, HPC, diffusion, exclusion mutuelle.

Contents

Opening Remarks

1 Introduction

1.1 Thesis Context

A logical way to start a thesis that deals with manycore processors is by describing what those are and why we need them. This is easy and hard at the same time. It is easy because there are ample resources on the very same topic, ranging from thoroughly reviewed research articles, to Internet resources such as blog posts, online magazine articles and discussions between technology fans. Much of this abundant information can be filtered and copied into this paragraph verbatim. It is hard for the very same reason: Giving an original perspective, or at least a personal touch, to a description of the necessity for manycore processors is rather challenging. Besides, it is not very rewarding either, as manycore programming is slowly becoming mainstream: Being familiar with its basics is becoming a compulsory part of most undergraduate curricula. With that in mind, this introduction will just briefly touch upon the history of manycore and then turn to an overview of challenges this thesis deals with.

We have been witnessing a big shift in the processor industry during the last decade. Due to physical constraints, it has become infeasible to keep getting more and more efficient processors simply by increasing their operating frequency. Instead, research and industry turned toward integrating multiple cores on a single chip [ONH$^+$96], thus moving a big share of work to software writers, whose responsibility then became to leverage the distributed computational power in the best way. The terms *multicore* and *manycore* were coined. The former denotes the integration of more than one processor on a chip (usually a small number), whereas the latter is used to stress that the number of integrated cores is so big that they become an inexpensive, affordable resource. Soon afterwards, people started talking about the *Manycore Revolution*, the name used to describe numerous challenges related to designing the hardware and software of an efficient, yet easy to exploit, manycore machine.

One of the central questions when talking about manycore processors is the choice of a programming model [McC08]. This question can be discussed at different layers of the software stack, but it is fundamentally dictated by the abstractions that the hardware exposes. Indeed, it is hard to make use of a processor whose cores cannot communicate with each other in some way. The moment we decide to enable inter-core communication, however, we realize that there are a number of decisions to be made.

One way to enable such communication is *shared memory*, where cores of a multicore processor have access to a region of memory addressable by each and every of them. Shared memory is an abstraction whose implementation involves a number of aspects, including topology (placement of memory with respect to cores), addressability (what cores can access what part of memory), available operations (instructions for accessing memory, their atomicity and granularity), and data consistency (precisely defining the semantics of shared-memory operations), to name a few. The most widespread implementation of shared memory on today's manycore platforms is *cache-coherent shared memory* [SHW11]. Besides a large slow memory that every core can address, each core has one or more levels of small but fast private caches, where recently used data are stored for faster reuse. A consequence is that the most recent copy of shared data is not necessarily in the shared part of memory, but might also be in a core's private cache. In order to keep the cache hierarchy (nearly) invisible to software, and make sure no memory operation manipulates stale data, an additional hardware mechanism is necessary – a *cache coherence protocol* [SHW11]. For every available memory location, the cache coherence protocol knows (or can deduce) where its most recent copy is, and manages simultaneous accesses to it by different cores. Therefore, what the programmer sees is a shared memory with the expected semantics, whereas software can still profit from the cache hierarchy.

Cache-coherent shared memory, however, has its downsides. Namely, sharing data introduces a need for *synchronization*, in order to maintain program correctness even when threads running on different cores try to manipulate shared data simultaneously. On a processor with a cache-coherent shared memory, synchronization is implemented by reading and writing shared variables. However, the cache coherence protocol has a big impact on its performance. Even simple synchronization patterns can result in surprisingly complex message traffic between cores, memory controllers and coherence agents [DGT13]. Optimizing concurrent code thus requires an in-depth understanding of cache coherence protocols and memory consistency models. Worse, vendors tend to hide their details, or provide them in informal or incomplete ways [OSS09], which makes the task of designing efficient synchronization algorithms notoriously hard. On top of that, some recent studies even question the scalability of traditional cache-coherent shared memory [HDH$^+$10], although arguments against such forecasts have also been given [MHS12].

In response to these concerns, message passing has been considered as an alternative. There are experimental [HDH$^+$10] as well as commercial [Kal14, Ada14] processors with message passing as the primary way of inter-core communication and synchronization. Message passing does not suffer from scalability problems and offers explicit control over communication. Indeed, some types of applications, such as various scientific workloads considered by the High Performance Computing (HPC) community, are traditionally written using message passing, albeit usually for large distributed systems. For such applications, message-passing manycores are a very natural fit. But even in the context of traditional software, some studies call for its complete redesign with message passing in mind, notably in the context of operating systems [BBD+09, WA09]. On the other hand, message-passing programming is usually hard, as full control over communication also means that the programmer has to think about data placement and use explicit messages to send data to the right place at the right time.

Both programming models are well established in practice and are supported by a substantial number of available frameworks and tools for general-purpose parallel programming. Some of the most well-known frameworks and libraries for writing parallel programs using shared memory are OpenMP [DM98], Cilk [BJK+96], and TBB [Rei07]. As far as message passing is concerned, the Message Passing Interface (MPI) [SOHL+98] is the de-facto standard. It should be noted that the machine architecture and the high-level programming model do not necessary match – for example, one can do MPI on a shared-memory machine, or shared-memory programming on a message-passing machine, provided there are suitable software abstractions.

Regardless of the concrete framework and machine characteristics, shared memory and message passing rely on some basic communication and synchronization primitives, used as building blocks of user programs, as well as operating systems. In the context of message passing, besides basic *point-to-point* operations, these typically include primitives for data dissemination and gathering, known as *collective operations*. As for shared memory, we most often talk about primitives for *synchronization on shared data*. Such basic primitives are the main topic of this thesis. We study how their performance can be improved and modeled.

1.2 Thesis Overview

The thesis studies communication and synchronization on manycore processors. This includes both message passing and cache-coherent shared memory, as well as taking advantage of the *hybrid* nature of modern hardware, allowing the two programming models to coexist on the same chip. The contributions are organized in two parts.

Part I deals with communication in the context of a message-passing processor. As shared-memory support on such a machine is either limited or non-existing, collective operations are very important, as they enable efficient data exchange that requires the participation of multiple, or even all cores. We study *broadcast*, as one of the basic collective operations, on the experimental Intel SCC message-passing processor. Unlike classic broadcast algorithms, built on top of *send* and *receive* operations, we leverage the existence of lower-level *put* and *get* on-chip communication primitives, which results in a significant performance improvement. The principal contributions of Part I are:

- OC-BCAST, a broadcast algorithm tailored to take advantage of fast on-chip communication. One version of OC-BCAST targets synchronous broadcast, provided by libraries such as those that implement MPI, whereas another version offers fully generic, asynchronous operation.

- A performance evaluation on the Intel SCC, which includes both versions of OC-BCAST and the well-known *binomial tree* and *scatter-allgather* broadcast algorithms. The evaluation reveals that OC-BCAST offers both lower latency and higher throughput than both alternatives.
- A communication model for the Intel SCC, which enables analytical comparison of the different broadcast algorithms, thus explaining the advantage of OC-BCAST.

Part II focuses on classic processors with support for cache-coherent shared memory. In this context, we study *mutual exclusion* on shared data, as one of the basic problems of shared-memory programming. We first show how shared-memory algorithms for mutual exclusion can be significantly improved if they are selectively augmented with message passing, that is, if we use message passing to optimize their most critical parts. This demonstrates the usefulness of *hybrid* processor designs, where shared memory and message passing coexist. Furthermore, these insights enable us to improve the performance of executing contended critical sections even on shared-memory-only hardware. The contributions of Part II include:

- HYBLOCK and HYBCOMB, novel algorithms for mutual exclusion. Their novelty lies in the fact that they use both message passing and cache-coherent shared memory, thus taking the best of both worlds. MP-SERVER is also presented, as a straightforward, but a very efficient way to profit from message-passing hardware in this context.
- Optimizations for shared-memory-only processors, whose goal is to mimic the behavior of MP-SERVER and thus improve performance even without hardware support for exchanging messages.
- Detailed experiments on a hybrid Tilera TILE-Gx processor, as well as as two x86 processors from Intel and AMD, which confirm that our algorithms and optimizations outperform a number of alternatives, including the most efficient known classic locks and combining algorithms.

Broadcast on a Message-Passing Processor: Algorithms and a Model

I

2 Background and Preliminaries

2.1 Motivation

As we saw in Chapter 1, one of the manycore research directions is studying loosely-coupled configurations, where cores cannot communicate in a traditional way, over a cache-coherent shared memory. Instead, the principal means of communication is direct exchange of messages. In the context of traditional computer systems, this shift represents a serious challenge, since most existing software, ranging from operating systems and device drivers, up to high-level application code, is written with coherent shared memory in mind. Two solutions to this problem have been proposed: Keeping the well-established shared-memory abstraction and implementing it in software on top of message passing [Tor09] and re-thinking the whole software stack with message passing in mind [MVdWF08].

There are, however, applications where the existence of shared memory is not required. Such is the case of High-Performance Computing (HPC) applications. They are typically written to run on distributed systems comprising of thousands of machines, connected via fast networks. Message passing is commonly used for work coordination between different machines, most often in form of the Message Passing Interface (MPI) [SOHL$^+$98]. Therefore, it comes at no surprise that such applications lend themselves nicely to message-passing manycores. What is more, moving from traditional clusters to manycore processors is advantageous in terms of resource consolidation and energy efficiency [K$^+$08, Tor09].

Although the transition of HPC applications to message-passing manycore systems is much more straightforward than it is the case with traditional shared-memory software, challenges still exist. One of them, tackled in this part of the thesis, is providing communication primitives optimized for the underlying hardware platform. Namely, for a given hardware stack (machines and interconnect), a set of communication primitives is provided to the programmer, typically as part of the MPI interface. These primitives include functions that enable point-to-point and collective communication between computational units. They abstract away hardware details: The programmer uses them as a black box when writing her application code. Internally, however, the implementation of collective communication primitives is very dependent on the hardware at hand, since the goal is to put the available resources to optimal use. When transitioning from machine clusters to manycore processors, a question that arises is whether we can use the existing techniques to optimize communication, or we have to come up with new ones. In the remainder of this part, contributions are given to answering this question: We demonstrate how collective communication can be optimized to leverage the resources of a manycore processor.

The concrete communication problem that we study is broadcast, the primitive that a computational unit (sender) uses when it needs to disseminate data to all other computational units (receivers). This primitive is important in HPC applications [Nis09], but the need for efficient broadcast is ubiquitous and of general interest [Tor09]. Indeed, when message passing is the only means of communication, one way to provide the data sharing abstraction is through replicating data across cores [BBD$^+$09]. Consistency is then managed using different protocols [BBD$^+$09, DGY14], in which broadcast typically plays an important role. Although our work focuses on the broadcast primitive, our insights and results are expected to be useful in studies that cover other collective operations.

Obviously, the most direct way to study manycore communication is by experimenting with existing products and prototypes. With this respect, our platform of choice is the Intel SCC [HDH$^+$10], a manycore prototype developed by Intel Labs. Although not an end-user product, this processor comprises many features existing in chips commercially available at the time of writing this report. Indeed, the Parallella processor [Ada14] is very similar to the SCC in many aspects, including the NoC interconnect and small on-chip buffers for *one-sided communication* between cores.

In the rest of this chapter, we briefly describe the concept of one-sided communication, which is key to the improvements we propose (Section 2.2). Then we present the Intel SCC platform (Section 2.3), before detailing our contributions (Section 2.4) and related work (Section 2.5).

2.2 One-sided vs. Two-sided Communication

In classic distributed systems, parties usually communicate using *send* and *receive* primitives. The *send* primitive takes data from a buffer specified by the sender and transfers it to the receiver. The *receive* primitive places the data into a buffer specified by the receiver. Note that these primitives completely abstract away the internal mechanism used to implement communication: Any kind of network, communication protocol and data representation can be used behind the scenes. While this is convenient from the perspective of portability and ease of use, a shortcoming of this approach is the reduced ability to leverage the characteristics of the underlying hardware. Communication that relies on send and receive is often referred to as *two-sided*. The reason is that any complete data transfer from one party to another necessarily involves actions (i.e. consumes CPU cycles) from both the sender and the receiver.

To remedy this, a paradigm often seen in high-performance environments is *one-sided commu-*

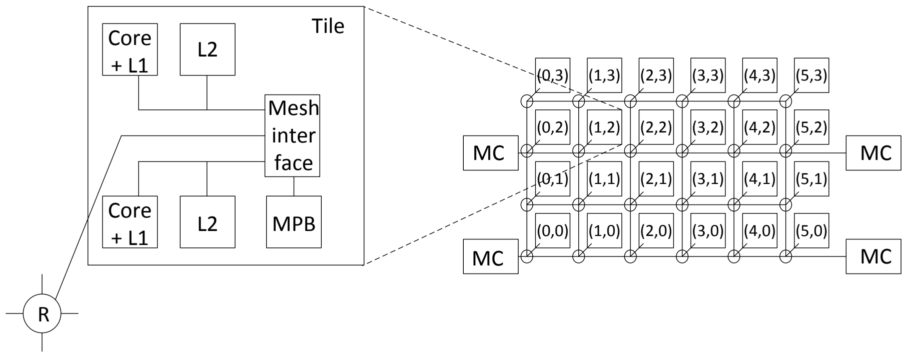


Figure 2.1 – SCC Architecture (MC – memory controller, MPB – message-passing buffer, R - router)

nication, consisting of *put* and *get*, and optionally some additional synchronization operations. In this paradigm, one communication party can *directly* access a portion of memory physically belonging to another, remote party. Both *put* and *get* take two buffers as arguments, one local and one remote. A *put* transfers data from a local buffer to a remote one, whereas a *get* does the opposite. Consequently, data can be transferred between communication parties with only one of them performing the actual work, since it can directly access both the source and the destination buffer. Next we describe the experimental Intel SCC processor, which offers an actual on-chip implementation of one-sided communication.

2.3 The Intel SCC

The Intel SCC is a general-purpose manycore prototype developed by Intel Labs. The cores and the network-on-a-chip (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 [KK79]. 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: 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 non-coherent shared memory by changing the default configuration.

Cores can transfer data using the one-sided *put* and *get* primitives provided by the RCCE library [vdWMH11]. Using *put*, a core (a) reads certain 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 a 32-byte cache line. If the data are larger than one cache line, they are sequentially transferred in cache-line-sized packets. During a remote read/write operation, each packet traverses routers on the way from the source to the destination. The local MPB is accessed directly or through the local router.[1]

Using the basic inter-processor interrupt (IPI) mechanism on the SCC, a core can send an interrupt to another core by writing a special value to the configuration register of that core. This generates a packet which is sent through the on-chip network to the destination core. Although this mechanism is simple and straightforward, it lacks some essential features. For example, the identity of the notifier is unknown and it is possible to send only one interrupt at a time. Fortunately, the SCC has an off-chip FPGA, which allows for adding new hardware features. The Global Interrupt Controller (GIC) is an extension to the basic IPI mechanism, provided by Intel. The GIC comprises a set of registers for managing IPI (request, status, reset and mask). As a consequence, a core can send an interrupt to up to 32 other cores in just one instruction, by writing an appropriate bit mask to its request register.[2] The work of generating interrupt packets is completely delegated to the GIC.

2.4 Contributions

Our principal goal is investigating the efficient implementation of the broadcast primitive on the Intel SCC. We distinguish between two types of broadcast. The first is executed by having all processes in the application call the broadcast function with matching arguments: the sender calls it with the message to broadcast, while the receiving processes call it to specify the reception buffer. Such an interface is traditionally used in writing parallel HPC applications, where all of the processes typically run the same program, just on different data sets. We will refer to broadcast with such an interface as *synchronous broadcast*.

The second type does not require explicit participation of the receiving processes: If there is an incoming message, the receiving process automatically handles it by executing a predefined routine. Such a primitive can be used when a manycore processor is viewed as a general-purpose distributed system, with different processes running different programs. For example, it can be used for ensuring replicated data consistency in a message-passing operating system [BBD$^+$09]. This version will be referred to as *asynchronous broadcast*.

The principal contributions of this part of the thesis are as follows:

- **OC-BCAST, a novel synchronous broadcast algorithm.** To make the best use of on-chip resources, we devise OC-BCAST (*On-Chip Broadcast*), a k-ary tree algorithm based on one-sided communication (*put* and *get*). The basic idea of OC-BCAST is simple: Instead of disseminating data by sequentially copying them to individual message buffers of each and every core, the broadcast sender only puts them in its own message

[1]Direct access to the local MPB is discouraged because of a bug in the SCC hardware.
[2]The upper limit of 32 is merely a consequence of the 32-bit memory word on the P54C

passing buffer (MPB): Multiple receivers then fetch the data in parallel, thus improving performance. The main difficulty lies in the fact that this parallelism is not infinite: Too many cores simultaneously accessing the same message buffer in parallel cause contention on the buffer and the interconnect, possibly canceling out the performance gain, or worse, resulting in a performance loss with respect to the "sequential" version, where the sender sends the data to the receivers one by one. For that reason, OC-BCAST forms a k-ary tree, where k can be adapted to the available platform and its characteristics. As we will show, even on a single chip (the Intel SCC in this context), different values of k can be optimal, depending on the exact configuration and metric. Besides the k-ary tree for data dissemination, OC-BCAST relies on separate *notification* binary trees. Additionally, we improve the performance of OC-BCAST using *pipelining* and *double buffering*.

- **Extending OC-BCAST to support asynchrony.** Next, we target asynchronous broadcast. Using the SCC's hardware support for inter-core interrupts, we conceive a modified, asynchronous version of OC-BCAST. From the conceptual perspective, the main challenge is in avoiding deadlocks, which exist if OC-BCAST is straightforwardly modified to operate with interrupts. We solve this problem by queuing messages that could cause a deadlock in the off-chip memory, and then sending them later when appropriate. From the implementation perspective, there is a need for enabling fast asynchronous communication between userspace programs running on different cores: Inter-core interrupts are treated by the kernel and are not propagated to applications. Our solution consists of a kernel module that converts received inter-core interrupts to UNIX signals, which can be propagated to the appropriate application.

- **Experimental comparision of OC-BCAST and state-of-the art broadcast algorithms.** Our performance evaluation shows that OC-BCAST is much more efficient than existing approaches to broadcast on the Intel SCC. The comparison of OC-BCAST with the *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. Recall that the binomial tree and scatter-allgather are conceived to provide low latency and high throughput, respectively. 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.

- **Analytical model of on-chip communication.** Finally, to better understand the obtained performance increase, we present a LogP-based [CKP+93] communication model for the Intel SCC. We use the model to analytically evaluate OC-BCAST, as well as the broadcast algorithms based on scatter-allgather and a binomial tree. The result shows that the main advantage of OC-BCAST is in fewer off-chip memory accesses on the critical path, which translates to lower latency and higher throughput. Furthermore, the performance predicted by the model is, for the most part, within 10% of that measured in the experiments. We attribute this precision to the way the Intel SCC is architected: The use of simple cores and interfacing logic, as well as direct communication using message passing, results in highly predictable and modelable performance. This is a much more challenging task in systems with more complex cores or cache coherence [RH13].

2.5 Related Work

2.5.1 Broadcast in high-performance systems

A message-passing manycore chip, such as the SCC, is similar to many existing HPC systems since it includes a large number of processing units connected through a high-performance network. Broadcast has been extensively studied in such systems. In MPI libraries, binomial trees and *scatter-allgather* [SVDG00] algorithms are mainly considered [GFB+04, TRG05]. 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. For large mesh or torus topologies, non-overlapping spanning trees can provide better performance [AHA+05]. Our algorithms will be directly compared with binomial tree and scatter-allgather in Section 3.3.

One-sided operations, as described in Section 2.2, have been introduced to take advantage of the capabilities of high-performance network interconnects such as InfiniBand [TA00]. On the SCC, operations on the MPBs allow the implementation of efficient one-sided communication [MVDW10]. The implementation of collective operations on top of one-sided communication has been extensively studied. Most high-performance networks provide *Remote Direct Memory Access* (RDMA) [AHA+05, TA00], *i.e.*, the one-sided operations are offloaded to the network devices. Some work tries to directly take advantage of these RDMA capabilities to improve collective operations [GBPN03, HSR07, LMP04, SBM+05]. However, it is hard to reuse these results in the context of the SCC for two main reasons: (i) they leverage hardware-specific features not available on the SCC, *e.g.*, hardware multicast [HSR07, LMP04], and (ii) they make use of large RDMA buffers [GBPN03, SBM+05], 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 [LWK+03]. This way, the well-established algorithms for collective operations based on two-sided communication can be directly used. The SCC communication libraries available at the time of carrying out our study adopted this solution. The RCCE library [MVDW10] 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 [Cha10]: 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 [CURK11]. Instead, our work aims to directly leverage one-sided communication available on the SCC.

Some concurrent and subsequent studies have also addressed optimizing collective communication on the SCC. Kohler et al. [KRGF12] improve RCCE_comm by applying some general optimizations. Although they discuss the importance of MPB-aware strategies, they mainly improve performance by introducing nonblocking communication (provided by the iRCCE library [CLRB11]) and minimizing software overheads. Matienzo et al. [MJ13] propose the MPB-aware *ModMPB* broadcast algorithm, which shares some basic ideas with OC-BCAST. Unfortunately, there is no direct performance comparison.

2.5.2 Asynchronous communication on the Intel SCC

On the Intel SCC, the only means of asynchronous communication are inter-processor interrupts (IPIs).[3] Communication using interrupts is often expensive because of various hardware and software overheads. The SCC is no exception, as confirmed by several studies. The SCC port of Barrelfish [PSMR11] uses IPIs to notify cores about message arrivals. The round-trip message latency reported by the authors was found too high for point-to-point communication in such a system, despite running it on bare metal with the minimum needed software overhead. IPIs have also been used in the SCC port of distributed S-NET [VGvT+11], a declarative coordination language. The port is based on an asynchronous message-passing library: Interrupts are trapped by the Linux kernel and then forwarded to the registered userspace process in the form of a UNIX signal, which is the idea reused in our study. Using a similar round-trip experiment as in [PSMR11], the authors confirm the high latency of inter-processor interrupts. Moreover, the latency they observe is even higher than in [PSMR11], mainly because of a necessary context switch before delivering a signal to the registered userspace process. A direct comparison with RCCE, the native SCC message-passing library based on polling [MVDW10], has shown that IPIs are far less efficient in terms of latency for point to point communication.

We are unaware of other studies targeting asynchronous collective operations on the SCC, which is likely due to the aforementioned high cost. Nevertheless, our work demonstrates that IPIs can be used in this context with acceptable performance, since the high cost can be compensated by sending parallel interrupts.

2.5.3 Modeling on-chip communication

There are ample resources on modeling computation and communication in different contexts. In this study, the well-known LogP model [CKP+93] is used to analytically evaluate broadcast

[3]Strictly speaking, it is possible to communicate asynchronously using a dedicated polling thread on every core, but this solution is not feasible in practice, as it wastes CPU cycles and energy.

algorithms on an experimental platform, the Intel SCC. In the context of the SCC, LogP has also been used to model some basic communication patterns [Rot11]. Ramos and Hoefler [RH13] model the Xeon Phi processor, the commercial successor of the SCC. This proves to be a hard task, mostly because the Xeon Phi is a cache-coherent machine.

Many models extend LogP in different ways. Some notable examples are LogGP [AISS95], in which special consideration is given to big messages, and LogPC [MF98], which takes network contention into account. LogP and similar models have been used to explore the design space of tree-based broadcast algorithms and to prove their efficiency under different assumptions [KSSS93, BMR05, SST09].

2.6 Outline

The rest of this part of the thesis is organized as follows. Chapter 3 presents the two versions of OC-BCAST, as well as experiments that compare it with broadcast algorithms well-established in the literature. In Chapter 4, we derive a performance model of the SCC, which enables us to analytically express the advantage of OC-BCAST over commonly used broadcast algorithms implemented using two-sided communication.

3 Broadcast on the Intel SCC: Algorithms and Evaluation

In this section, we first describe the synchronous version of OC-BCAST, our broadcast algorithm for the SCC built on top of one-sided communication primitives (Section 3.1). Then we discuss how OC-BCAST can be extended to support asynchrony (Section 3.2). Finally, we experimentally compare our solution with broadcast algorithms implemented using two-sided communication (Section 3.3). Section 3.4 discusses the results.

3.1 OC-BCAST: a Synchronous Broadcast Algorithm

3.1.1 High-level description

To simplify the presentation, we first assume 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 one-sided communication 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 on the MPBs and on-chip network (experiments in Section 4.1.3 will confirm this). 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 in 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 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 3.1 illustrates the k-ary tree used for message propagation, and the binary trees used for notification. Core c_0 is the root of the message propagation tree; the subtree with root c_1 is shown. Core c_0 notifies its children using the binary notification tree shown at the right of Figure 3.1. Node c_1 notifies its children using the binary notification tree, as depicted at the bottom of Figure 3.1.

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 c, *i.e.*, the core that is neither the root nor a leaf, c performs the following steps. Once it has been notified that a new chunk is available in the MPB of its parent c_p, core c: (i) notifies its children, if any, in the notification tree of c_p; (ii) copies the chunk to its own MPB (using a *get*); (iii) sets its $doneFlag$ in the MPB of c_p; (iv) notifies its children in its own notification tree, if any; (v) gets the chunk from its MPB to its off-chip private memory; (vi) waits until its children's $doneFlag$-s have been set.

Finding an efficient k-ary tree that takes into account the topology of the NoC is a complex problem [BMR05] and it is orthogonal to the design of OC-BCAST. On top of that, as Chapter 4 will show, the SCC itself is not an ideal testbed for addressing topology issues: The difference between the lowest (two adjacent cores) and the highest (two farthermost cores) point-to-point communication latency is only about 1.3x, whereas the choice of the broadcast algorithm has a much higher performance impact. With that in mind, henceforth 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 3.1 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. Instead of waiting for each chunk to be completely disseminated before sending the next one, pipelining can be used along the propagation tree, from the root to the leaves: As soon as children of a tree node acknowledge chunk reception, a new chunk can be immediately transferred.

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 [CLRB11]. Until now, we have considered messages split into chunks of MPB size,[1] which allows an MPB buffer to store only one message chunk. With double-buffering,

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

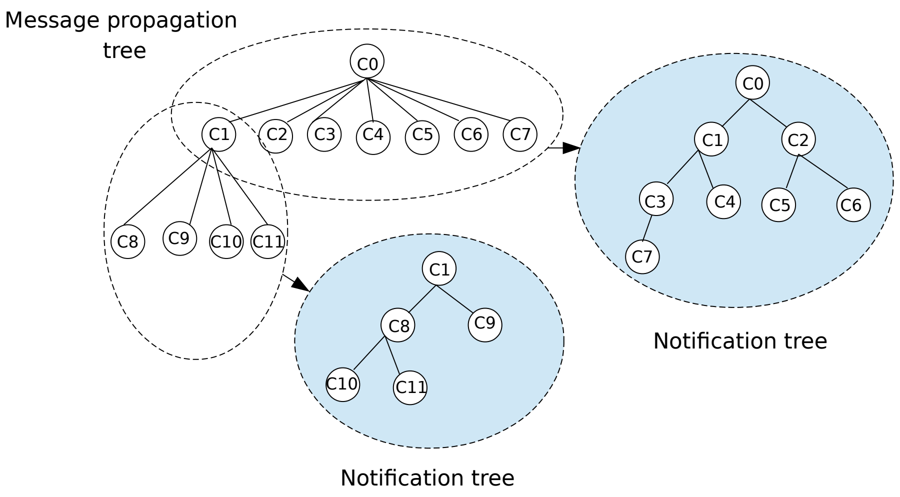


Figure 3.1 – k-ary message propagation tree ($k = 7$) and binary notification trees.

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

3.1.2 Detailed description

The pseudocode for a process running OC-BCAST on core c is presented in Algorithm 1. To broadcast a message, all cores invoke the broadcast function (line 20). The input variables are msg, containing a pointer to the message to broadcast (at the root), or pointing where the received message should be stored (at any other node), and $root$, the id of the core broadcasting the message. The message size is an implicit argument (it is needed at line 16 to determine the number of chunks).

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. We introduce the following notation for put and get operations: *'put src $\longrightarrow$ dest'* and *'get dest $\longleftarrow$ src'*. Recall that this version of OC-BCAST does not handle concurrency – since all of the participating nodes have to call the *broadcast* function with matching arguments (as typically seen in HPC applications), it is assumed that multiple broadcast operations cannot take place in parallel. This limitation will be removed in Section 3.2.

Each core c has a unique data parent $dataParent_c$ in the data propagation tree, and a

Algorithm 1 OC-BCAST, synchronous version (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 $chunk \longleftarrow MPB[c].buffer[chunkID \bmod 2]$
```

set of children $dataChildren_c$. The set $notifyChildren_c$ includes all of the cores that core c should notify during one execution of the algorithm. Note that a core c can be part of several binary trees used for notification. In the example of Figure 3.1, 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, $prepareTree$ function). As noted before, a simple algorithm based on ids will be used throughout the study, but we do not put any strict requirements on the tree structure. We only assume that a predefined deterministic algorithm is used to compute the broadcast trees.

MPBs are represented by the global variable MPB where $MPB[c]$ is the MPB of core c. A $notifyFlag$ and k instances of $doneFlag$ (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 notification. To implement double buffering, the two buffers in the MPB are used alternatively: for the chunk <$bcastID$,$chunkID$>, 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 3.1, 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 3.1, 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).

3.2 Asynchronous Version of OC-BCAST

Now we describe how OC-BCAST can be extended to support *asynchronous* broadcast, where only the sender calls the broadcast function and broadcast operations from multiple sources can take place in parallel. In this context there are two problems to address: replacing polling with interrupts (Section 3.2.1) and managing multiple broadcast operations taking place in parallel (Section 3.2.2). After describing our solutions, the modified version of OC-BCAST is given (Section 3.2.3), followed by implementation details (Section 3.2.4).

3.2.1 Enabling asynchronous communication

As we have seen, OC-BCAST uses MPB polling for notification. Clearly, polling is not an option if we want message reception to be asynchronous, so the SCC interrupt hardware presented in Section 2.3 has to be used. This means that the notification mechanism has to be modified. Recall from Section 2.3 that the SCC's Global Interrupt Controller (GIC) contains a register which enables a core to request interrupts to be sent to up to 32 cores using a single instruction. [2] Instead of using a binary notification tree like in Algorithm 1, a parent can directly inform all its children that a message is available, by sending them parallel interrupts. On the other hand, using interrupts to acknowledge message reception to the parent is not necessary, as there is no asynchrony: The parent knows it should wait for the acknowledgement, and it is safe to discard the data in the local MPB only after the acknowledgement has arrived. Thus, the modified algorithm can be summarized as follows:

1. The broadcast sender puts the message from its private memory to its MPB and sends a parallel interrupt to all of its children. Then it waits until all the children have received the message.
2. Upon receiving the interrupt, a core c copies the data from the parent's MPB to its own MPB and acknowledges the reception of the message to the parent by setting the corresponding flag in the parent's MPB.
3. Core c then sends a parallel interrupt to notify its own children (if any) and then copies the message from the MPB to its private memory. Then it waits until all its children have copied the message to their buffers.
4. When all of c's children have acknowledged reception, c can make its MPB available for other actions (possibly a new message).

3.2.2 Managing concurrent broadcast

Recall that OC-BCAST is designed in the context of HPC applications, where a core explicitly calls the broadcast function to participate in the collective operation. As a consequence, a core is involved in only one collective operation at a time. However, using interrupts in OC-BCAST allows us to move to a more general model where broadcast operations can arbitrarily interleave at one core. Now we discuss how to efficiently manage this aspect.

The above algorithm has to be modified to allow asynchronous broadcast operations issued by different cores. Indeed, without modifications the algorithm would be prone to deadlocks. A simple scenario can be used to illustrate a deadlock situation. Consider two cores c and c' that try to broadcast a message concurrently, with c' being a child of c in the tree where c is

[2] Note that the mere fact that interrupts can be *requested* in parallel does not necessarily mean that they are *sent* in parallel. However, the GIC implementation does actually ensure this kind of parallelism, as our evaluation in Section 3.3 will confirm.

the root and the opposite in the tree where c' is the root. Core c' cannot copy the message that c is trying to broadcast in its MPB because it is busy with its own message. Core c' will be able to free its MPB when it knows that all its children have copied the message. However c cannot get the message from c' either, because it is in exactly the same situation as c'. There is a deadlock.

To deal with this problem, a simple solution would be to use a global shared lock to prevent multiple broadcast operations from being executed concurrently. In general, in a system based on message passing, the lock can be implemented using an algorithm for distributed mutual exclusion [NTA96]. In the specific case of the SCC, there is an alternative solution that uses on-chip test-and-set registers. Whatever lock implementation we choose, no further modifications to OC-BCAST are necessary. However, this would limit the level of parallelism and prevent us from fully using the on-chip resources.

To avoid deadlocks without limiting the parallelism, we adopt the following solution: If the MPB of some core c is occupied when a notification about a new message arrives, c copies the message directly to its off-chip private memory, instead of copying it to the MPB. Additionally, if c has to forward the message, it is added to a local queue of messages that c has to forward (a queue is used in order to eliminate the possibility of starvation). Eventually, when the MPB is available again, c removes messages from the queue and forwards them to the children.

3.2.3 Modified OC-BCAST

Algorithm 2 presents the pseudocode for a core c. During initialization, each core is able to compute the tree that will be used by each source (line 7). As before, an arbitrary algorithm can be used for tree computation. As with original OC-BCAST, if a message is larger than the available MPB, it is divided into multiple chunks.

For the sake of simplicity, the pseudocode is not fully detailed. It only illustrates the important modifications that are made to synchronous OC-BCAST avoid deadlocks. As with synchronous OC-BCAST, the $broadcast$ function is called by the sender. However, the reception part is asynchronous: The function $deliver\_chunk$ is registered to be triggered when an interrupt is received from another core.

We define three functions that implement basic data movement and notification procedures. $OCBcast\_send\_chunk(chunk, Tree)$ initiates the broadcast of the chunk $chunk$ in the tree $Tree$. It puts $chunk$ in the caller's MPB, notifies the caller's children in $Tree$ by sending an interrupt to each of them, and then polls the local MPB until each of the children has fetched $chunk$. $OCBcast\_forward\_chunk(chunk, Tree)$ is similar, except for the assumption that the chunk is already in the MPB of the caller (so it only notifies the children and waits). $OCBcast\_receive\_chunk(chunk, buf, src)$ gets $chunk$ from the MPB of core src into buf (buf being either the MPB of the caller or a memory region in its off-chip private memory), and then notifies src using an MPB flag. It is worth noting that a chunk includes not only payload,

Algorithm 2 OC-BCAST, asynchronous version (code for core c)

```
Local Variables:
 1: $MPB_c$ {MPB of core $c$}
 2: $MPBStatus_c$ ← available {Status of the MPB}
 3: $chunkQueue_c \leftarrow \emptyset$ {Queue of chunks to forward}
 4: set of trees $Tree_1, Tree_2, ..., Tree_n$ {$Tree_c$ is the tree with $c$ as root}

 5: initialization:
 6:     define deliver_chunk() as the IPI handler
 7:     for $coreID \in 0...n$ do compute $Tree_{coreID}$

 8: broadcast($msg$)
 9:     for all chunk of $msg$ do
10:         broadcast_chunk($chunk$)

11: broadcast_chunk($chunk$)
12:     $MPBStatus_c$ ← busy
13:     OCBcast_send_chunk($chunk$, $Tree_c$)
14:     $MPBStatus_c$ ← available
15:     flush_queue()

16: deliver_chunk($chunk$, $source$)
17:     if $chunkQueue_c$ is empty ∧ $MPBStatus_c$ = available then
18:         $MPBStatus_c$ ← busy
19:         OCBcast_receive_chunk($chunk$, $MPB_c$, $source$)
20:         if $c$ has children in $Tree_{chunk.root}$ then
21:             OCBcast_forward_chunk($chunk$, $Tree_{chunk.root}$)
22:         $MPBStatus_c$ ← available
23:         flush_queue()
24:     else
25:         let $item$ be the memory allocated to receive the chunk
26:         OCBcast_receive_chunk($chunk$, $item$, $source$)
27:         if $c$ has children in $Tree_{chunk.root}$ then
28:             enqueue $item$ in $chunkQueue_c$
29:     if $msg$ corresponding to $chunk.msgID$ is complete then
30:         deliver $msg$ to the application

31: flush_queue()
32:     while $chunkQueue_c$ is not empty do
33:         dequeue $chunk$ from $chunkQueue_c$
34:         $MPBStatus_c$ ← busy
35:         OCBcast_send_chunk($chunk$, $Tree_{chunk.root}$)
36:         $MPBStatus_c$ ← available
```

but also some metadata, *i.e.*, the id of the core that broadcasts the message ($chunk.root$) and the id of the message the chunk is part of ($chunk.msgID$).

As mentioned before, we allow a core to receive chunks directly in its off-chip private memory when its MPB is busy with another chunk that is being sent (line 17). Thus, the sender can free its MPB. The chunks that the core is supposed to forward to other cores, are stored in a queue (lines 25-28), that is flushed when the MPB becomes available (line 15 and line 23). Note that to ensure fairness, if the MPB is free at the time the core receives an interrupt but some chunks are already queued to be forwarded (line 17), the chunk is received in the private memory and added to the queue. Thus, a chunk cannot overtake another chunk that has been in the queue already for some time. However, if no chunk is in the queue and the MPB is available, the chunk is first copied in the MPB to limit the number of data movements between the MPB and the private memory that could decrease performance.

3.2.4 Implementation

To implement asynchronous OC-BCAST on the SCC, we have developed a userspace library for interrupt handling, following the idea given in [VGvT+11]. Namely, a userspace process can register itself with a special kernel module. Every time an interrupt from another core is received, the kernel module sends a real-time UNIX signal to the registered process, which triggers a user-provided handler. We have opted for real-time signals because they can be queued if there is more than one signal pending. This way, we ensure that every interrupt is converted to a signal and the algorithm can be written entirely in userspace.

A drawback of this approach is a performance loss already observed in [VGvT+11]: To propagate an interrupt to userspace in the form of a UNIX signal, a context switch is necessary, which significantly increases end-to-end communication latency. Nevertheless, we have adopted this approach for two reasons. Firstly, such an implementation changes only absolute numbers and does not prevent us from observing changes in performance resulting from design-level decisions. The same algorithm could be implemented in the Linux kernel or directly on bare metal, which completely avoids UNIX signals and context switching. Secondly, our library is easy to integrate with RCCE and the accompanying tools, which makes it convenient for other researchers willing to use inter-processor interrupts without significant effort.

3.3 Experimental Evaluation

In this section we evaluate the different variants of OC-BCAST on the Intel SCC. We start by presenting our experimental setup (Section 3.3.1) and explaining how the other considered broadcast algorithms are implemented (Section 3.3.2). Next, we compare the latency and throughput of synchronous OC-BCAST with that of the introduced state-of-the art algorithms (Section 3.3.3), before quantifying the performance of the asynchronous version of OC-BCAST (Section 3.3.4).

3.3.1 Setup

The experiments were done using the default settings for the SCC: 533 MHz tile frequency, 800 MHz mesh and DRAM frequency and the standard look-up table 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 comparable. We define the latency of the broadcast primitive 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. 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. The kernel of every core runs the special kernel module for converting interrupts to UNIX signals, described in Section 3.2.4. This module is used only in the asynchronous implementation of OC-BCAST.

Besides our algorithms, we evaluate broadcast algorithms from the *RCCE_comm* library (their more detailed description is given below). RCKMPI [CURK11] relies on the same algorithms, but reuses an implementation that is not optimized for the SCC. Also, our experiments have confirmed that RCCE_comm currently performs better than RCKMPI. Thus, we have chosen RCCE_comm for our experiments, as it is the fastest available implementation of collective operations on the SCC, to the best of our knowledge.

Both OC-BCAST and the RCCE_comm algorithms use flags allocated in the MPBs to implement synchronization between the cores. The SCC guarantees read/write atomicity on 32B cache lines, so allocating one cache line per flag is enough to ensure consistency. OC-BCAST requires $k+1$ flags per core, and the rest of the MPB can be used for the message payload. For this, OC-BCAST uses two buffers of $M_{oc} = 96$ cache lines each. RCCE_comm, which is based on the RCCE message-passing library, uses a payload buffer of $M_{rcce} = 251$ cache lines.

As explained earlier, OC-BCAST has a parameter k, which dictates the degree of the broadcast tree. In our experiments, we use the values of 2, 7, and 47. While 2 is chosen as a minimum degree offering some parallelism (with $k = 1$, the tree degenerates to a list), 47 is chosen as the value where all the other cores are the sender's children (the depth of the tree is one). Additionally, we include results with $k = 7$, which is the minimum degree where the tree depth is two. As will be explained, in some situations this can give better performance than a tree of minimum depth, mostly because of contention on the MPBs.

3.3.2 Standard broadcast algorithms: binomial tree and scatter-algather

Here we briefly describe the algorithms that will be compared with OC-BCAST. As mentioned before, the binomial tree is mostly used to broadcast small messages. The algorithm is as follows. The set of nodes (cores in this case) is divided in 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 (for example, the node with the smallest id). Then, there is one node containing the message in both subsets. The initial step is then recursively repeated on both subsets (each of them is divided and the message is sent to the new subset) until all of the nodes have received the message.

If the message to broadcast is large, the scatter-allgather algorithm might have better performance. It has two phases. During the *scatter* phase, the message is divided into P slices, which are then distributed accross nodes, so that each node has one slice of the original message. A binomial tree can be used in this phase, similar to the one described in the previous paragraph. The difference is that only appropraite chunks are transferred, not the whole message. In the *allgather* phase, every node should obtain the remaining $P-1$ slices. In the RCCE_comm library, this is done using the Bruck algorithm [BHU+97]: The nodes form a logical ring and first, each node sends its slice to the left neighbour. In subsequent steps, a node sends to its left neighbour the slice it received from its right neighbour in the previous step. This way, the full message is available at all of the nodes after $P-1$ steps.

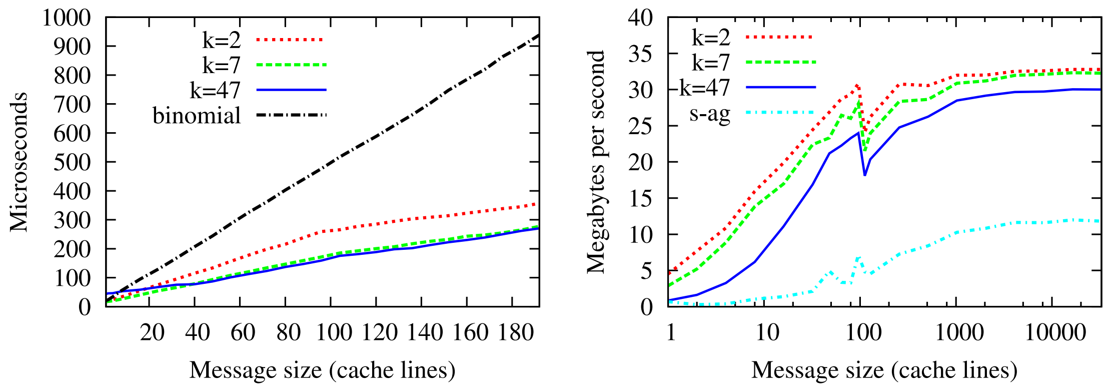
Although we do not explore that possibility here, it should be noted that the basic ideas of the presented algorithms can also be used to build one-sided algorithms (on top of *put* and *get*).

3.3.3 Synchronous broadcast

Our first goal is to see how OC-BCAST compares to the binomial and the *scatter-allgather* broadcast algorithms. We have tested the algorithms with message sizes ranging from 1 cache line (32 bytes) to 32'768 cache lines (1 MB). To evaluate the algorithms, 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 best one for a given message size.

Latency of small messages

Figure 3.2a shows the latency of messages of size $m \leq 2M_{oc}$. Even for messages consisting of one cache line, OC-BCAST with $k = 7$ provides 27% improvement compared to the binomial tree ($16.6\mu s$ vs. $21.6\mu s$). The difference grows with the message size. This is because a larger message implies more off-chip memory accesses on the critical path in the RCCE_comm algorithms. In OC-BCAST, in contrast, the number of off-chip operations remains constant, because the chunks are propagated through the MPBs, and copying them to the cores' off-



(a) Broadcast latency (b) Broadcast throughput

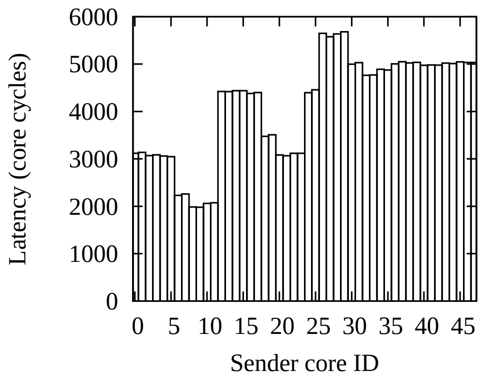
Figure 3.2 – Experimental comparison of broadcast algorithms. Legend: k=x, OC-BCAST with the corresponding value of k; binomial, RCCE_comm binomial; s-ag, RCCE_comm scatter-allgather.

chip memory is overlapped with this propagation. We will analyze this more thoroughly in Chapter 4, using a formal performance model. It can also be noticed that increasing k helps decrease the latency of 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 $k = 2$. However, this trend is not so obvious between $k = 7$ and $k = 47$. Namely, we can see that the corresponding curves almost completely overlap in Figure 3.2a, although one could expect $k = 47$ to be the optimal choice because of the lowest tree depth. This discrepancy is mostly due to MPB contention, which will be studied in more detail in Chapter 4.

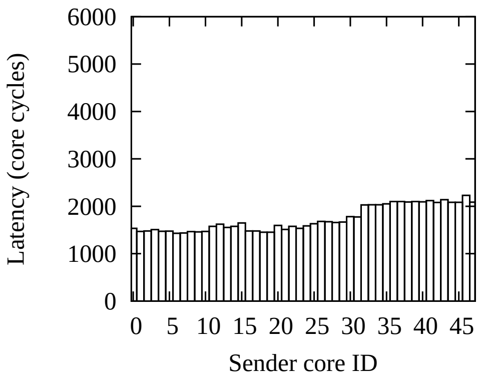
Throughput of large messages

The results of the throughput evaluation are given in Figure 3.2b (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.

We can see that OC-BCAST with $k = 47$ is slightly outperformed by the counterparts with lower values of k. Again, we attribute this to MPB contention, which becomes significant at higher levels of parallelism.



(a) With access to GIC registers



(b) Without access to GIC registers

Figure 3.3 – Latency of broadcasting an interrupt at the kernel level

3.3.4 Asynchronous broadcast

Here we evaluate the performance of the asynchronous version of OC-BCAST. Before evaluating the algorithm itself, we start by investigating the Global Interrupt Controller (GIC) on the SCC, when used to send multiple interrupts in parallel. Performance wise, this mechanism is the most critical part of the implementation.

Interrupt Hardware Performance

To evaluate how the GIC handles parallel interrupts, we have performed the following experiment: A core sends an interrupt to all cores (including itself), by issuing two instructions which write a mask of "1"-s to its request register. Then, the core measures the time until it receives its own interrupt. The results, given in Figure 3.3a, indicate a significant difference in latency observed by different cores, ranging from about 2000 to almost 6000 CPU cycles. Further experiments have confirmed that this difference grows as a function of the number of cores that the interrupt is sent to – it is barely noticeable for less than 20 cores, but then starts to increase rapidly.

The experiment presented above leads us to the conclusion that parallel notification using interrupts on the SCC does not scale well. Further investigation, however, explains that this is not a fundamental problem, but rather a consequence of its suboptimal implementation. Namely, upon receiving an interrupt, there is a fixed set of steps a core should perform. This includes reading from the status register, to determine the interrupt source, and resetting the interrupt by writing to the reset register. Since all of the registers related to interrupt handling are on the FPGA, access to them is handled sequentially. When an interrupt is sent to many cores at once, they all try to access their interrupt status register at the same time, but their requests contend and are handled one after another, which explains the observed performance loss. We believe that a proper on-chip implementation of interrupt registers would eliminate this problem, since they would be independent and could be accessed in parallel.

To confirm that the reason for bad scaling of the interrupt mechanism is contention on the FPGA, we have repeated the same experiment, but this time deliberately avoiding the FPGA registers, except on the sending core. Note that such a configuration is of little use in practice, but the goal of the experiment is only to discover the scalability bottleneck. In Figure 3.3b we see that the latencies measured across the cores are very similar and close to 2000 core cycles. Slight differences in latency are easy to explain. Namely, the FPGA is connected to the mesh via the router between tiles (2,0) and (3,0) (cf. Figure 2.1), so the round-trip time to the FPGA is shorter for cores closer to this router. Next, it takes slightly more time for cores 32 through 47 to receive their interrupt. This is because, as already described, it is possible to send at most 32 interrupts by issuing a single instruction. Therefore, when broadcasting an interrupt, a core first broadcasts to cores 0 to 31 in the first instruction, and then to the other cores, which results in slightly higher latency.

Another set of experiments, as well as comparisons with results of other authors [VGvT+11], confirmed that the latencies presented in Figure 3.3b are practically indistinguishable from the latency of sending point-to-point interrupts (about 2000 cycles). This implies that the cost of notification using interrupts is practically constant with respect to the number of cores notified. However, as we have described, sequential access to the off-chip registers for interrupt handling slows down the whole process in the current implementation on the SCC. Still, from Figure 3.3a we can see that even with this effect, broadcasting an interrupt to the 48 cores is only about 3 times more expensive than sending a point-to-point interrupt, making this mechanism interesting for use in group communication.

Asynchronous Broadcast Performance

Having seen the performance of the interrupt-generating hardware, we now analyze the performance of asynchronous OC-BCAST. The first experiment measures the latency when messages of different sizes are broadcast from one core (core 0 in this case), with $k = 47$. Table 3.1 compares the obtained latency with that of the synchronous version of OC-BCAST. The two algorithms have very similar latencies with these settings. This confirms that the interrupt hardware on the SCC is useful for designing asynchronous collective operations, even though its latency is high for point-to-point communication, as pointed out in other studies [PSMR11, VGvT+11]. However, recall that we fix the value of k to 47, which enables us to obtain the highest level of parallelism when sending the interrupts. Results with lower values of k are not shown: In that case, the latency of asynchronous OC-BCAST is largely inferior to that of its synchronous counterpart, because the high price of sending an interrupt is paid more than once (up to six times, when $k = 2$).

In the second experiment, we evaluate performance with a varying number of concurrent broadcasts from different cores. We change the number of sources, that is, the number of cores

Message Size (Number of cache lines)	1	32	64	128
OC-BCAST	44.1 μs	75.8 μs	112.7 μs	198.6 μs
Asynchronous broadcast	40.2 μs	75.5 μs	118 μs	196.7 μs

Table 3.1 – Comparing the latency of synchronous broadcast (OC-BCAST) and asynchronous broadcast for different message sizes ($k = 47$)

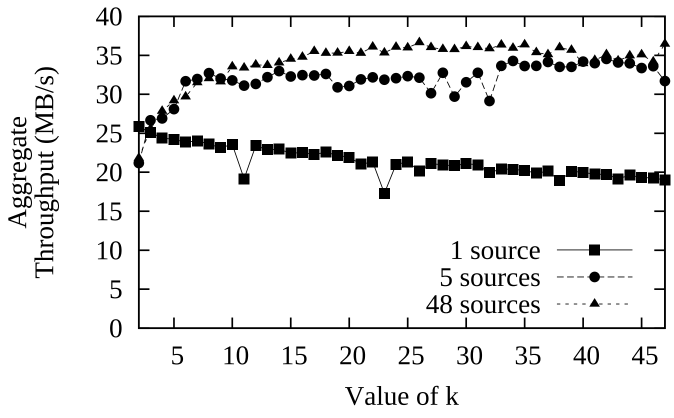


Figure 3.4 – Throughput of the asynchronous broadcast algorithm for different values of k and different number of concurrent sources

broadcasting in parallel. Each source repeatedly broadcasts a 4 KB (128 cache lines) message from its private memory, without waiting for the other cores to receive the message, thus creating a message pipeline. This way we observe the aggregate throughput of the system, that is, the amount of data broadcast in a unit of time. We repeat the experiments with different values of k. Note that low values of k are of interest in this experiment, since we measure throughput.

The result of this experiment is given in Figure 3.4. With a single source, the throughput decreases as k increases. The reason is the cost of polling flags (there are at most k flags to poll). To wait for an acknowledgment from its children, each parent has to poll k flags in its MPB and reset them afterwards. The sporadic performance variations can be explained by the fact that a core does not control when it will be signaled. In fact, when a core is about to forward a received message to the children, it can get interrupted to receive another message. If this happens, the children have to wait, which introduces performance drops.

With more than one source, the throughput increases. There are two possible reasons for this. The first one is that when a single node is broadcasting messages, the other cores are sometimes idle waiting for the next message to be available. With multiple sources, this idle time can be used to receive messages from other sources. The second reason is that if a core receives interrupts in different trees, it can often have more than one interrupt waiting to be serviced by the kernel. When this happens, all the pending interrupts will be serviced (converted to signals) one after another, and only then will the execution switch back to the userspace process. This actually means that there will not be one context switch per interrupt, but significantly fewer, resulting in a performance increase.

We can also see that the difference in throughput when broadcasting from 5 and 48 sources is not significant. This is because the system gets saturated. Based on measurements and the model presented in the next chapter, the maximum bandwidth when copying data from a core's MPB to the off-chip memory is about 55 MB/s (assuming cache line prefetching implemented in software as in iRCCE [CLRB11]). Our algorithm achieves 68% of this maximum bandwidth.

When it comes to the choice of k with multiple sources, the trend is opposite to the single-source case. This is especially visible for smaller values of k, where each increase by 1 obviously increases the aggregate throughput. To understand this, recall that the resources of every core are effectively used in this case, in the sense that there is no idle time. However, performing a broadcast operation consumes more resources on different cores if k is lower since there are more accesses to the GIC to send interrupts. Thus, the cores manage to do less useful work.

3.4 Summary

We have presented OC-BCAST, a broadcast algorithm that leverages on-chip one-sided communication. The design and implementation of OC-BCAST demonstrate how a high degree of parallelism can be attained in a broadcast algorithm for the experimental Intel SCC processor. Breaking the abstraction of two-sided communication and working directly with message passing buffers enabled a significant performance increase, in terms of both latency and throughput. The asynchronous version of OC-BCAST shows how the hardware and software overhead of interrupt-based communication, although high, can be made acceptable in collective communication if there is hardware support for interrupt multicast.

It is worth noting that there is still some room for further optimization. For example, on the SCC itself, the presented algorithms do not profit from the fact that an MPB is shared by two cores on the same tile. Taking this into account would reduce the number of necessary copy operations (although it would necessitate some extra synchronization between the cores on the same tile). Still, our goal was not to optimize the algorithm with every processor-specific feature in mind, but to present general reasoning. We believe OC-BCAST can be used without much modification on platforms with similar characteristics, such as the Parallella processor [Ada14].

4 Communication Model for the Intel SCC

Chapter 3 introduced OC-BCAST and experimentally showed its efficiency on the Intel SCC. In order to better understand the obtained performance gain, in this section we devise a communication model for the Intel SCC and use it to analytically express the performance of the different broadcast algorithms. We start by modeling the *put* and *get* primitives, the basic building blocks of all algorithm implementations on the SCC (Section 4.1). The proposed model assumes contention-free execution, so we also study contention on the SCC, in order to assess the model's validity domain. Next, we model two-sided communication (*send* and *receive*), as implemented in the RCCE library on top of *put* and *get* (Section 4.2). Finally, we derive formulas for the latency and throughput of synchronous OC-BCAST, as well as the two-sided broadcast algorithms, binomial tree and scatter-allgather (Section 4.3). The results are discussed in Section 4.4.

4.1 Modeling the *put* and *get* Primitives

Here we propose a model for the one-sided *put* and *get* primitives. It is based on the LogP model [CKP+93] and the Intel SCC specifications [HDH+10]. After introducing the model, we experimentally validate it and assess its domain of validity.

4.1.1 The model

The LogP model [CKP+93] characterizes a message-passing parallel system using the number of processors (P), the time interval or *gap* between consecutive message transmissions (g), the maximum communication latency of a single-word-sized message (L), and the overhead of sending or receiving a message (o). This basic model assumes small messages. To deal with messages of arbitrary size, it can be extended to express L, o and g as a function of the message size [CLMY96].

We adapt the LogP model to the SCC communication characteristics. The LogP model assumes that the latency is the same between all processes. However, the SCC mesh communication

$$L_w^{mpb}(d) = o^{mpb} + d \cdot L^{hop} \tag{4.1}$$

$$C_w^{mpb}(d) = o^{mpb} + 2d \cdot L^{hop} \tag{4.2}$$

$$L_r^{mpb}(d) = C_r^{mpb}(d) = o^{mpb} + 2d \cdot L^{hop} \tag{4.3}$$

$$L_w^{mem}(d) = o_w^{mem} + d \cdot L^{hop} \tag{4.4}$$

$$C_w^{mem}(d) = o_w^{mem} + 2d \cdot L^{hop} \tag{4.5}$$

$$L_r^{mem}(d) = C_r^{mem}(d) = o_r^{mem} + 2d \cdot L^{hop} \tag{4.6}$$

$$C_{put}^{mpb}(m, d^{dst}) = o_{put}^{mpb} + m \cdot C_r^{mpb}(1) + m \cdot C_w^{mpb}(d^{dst}) \tag{4.7}$$

$$C_{put}^{mem}(m, d^{src}, d^{dst}) = o_{put}^{mem} + m \cdot C_r^{mem}(d^{src}) + m \cdot C_w^{mpb}(d^{dst}) \tag{4.8}$$

$$\begin{aligned} L_{put}^{mpb}(m, d^{dst}) = o_{put}^{mpb} + m \cdot C_r^{mpb}(1) + \\ (m-1) \cdot C_w^{mpb}(d^{dst}) + L_w^{mpb}(d^{dst}) \end{aligned} \tag{4.9}$$

$$\begin{aligned} L_{put}^{mem}(m, d^{src}, d^{dst}) = o_{put}^{mem} + m \cdot C_r^{mem}(d^{src}) + \\ (m-1) \cdot C_w^{mpb}(d^{dst}) + L_w^{mpb}(d^{dst}) \end{aligned} \tag{4.10}$$

$$L_{get}^{mpb}(m, d^{src}) = C_{get}^{mpb}(m, d^{src}) = o_{get}^{mpb} + m \cdot C_r^{mpb}(d^{src}) + m \cdot C_w^{mpb}(1) \tag{4.11}$$

$$L_{get}^{mem}(m, d^{src}, d^{dst}) = C_{get}^{mem}(m, d^{src}, d^{dst}) = o_{get}^{mem} + m \cdot C_r^{mpb}(d^{src}) + m \cdot C_w^{mem}(d^{dst}) \tag{4.12}$$

Figure 4.1 – Communication Model (L – latency, C – completion time)

latency is a function of the number of routers traversed on the path from the source to the destination. In our model, the number of routers traversed by one packet is defined by the parameter d. Communication on the SCC mesh is done at the packet granularity. A packet can carry one cache line (32 bytes). We use the number of cache lines (CL) as unit for message size. Note that the SCC cores, network and memory controllers are not required to work at the same frequency. For that reason, time is chosen as the common unit for all model parameters.

For each operation, we model (i) the completion time, i.e., the time for the operation to return, and (ii) the latency, i.e., the time for the message to be available at the destination. We start by read/write on the MPBs and on the off-chip private memory. Then we model put/get operations based on read/write. A read operation, executed by some core c, brings one cache line from an MPB, or from the off-chip private memory of core c, to its internal registers[1]. The write operation, executed by some core c, copies one cache line from some internal registers of core c to an MPB, or the off-chip private memory of core c. The formulas representing our model are given in Figure 4.1, and are described in the following.

[1]The read operation, as defined here, should not be interpreted as a single instruction. Indeed, it is implemented as a sequence of instructions, which read an aligned cached line word by word. The first instruction causes a cache miss, and the corresponding cache line is moved to the L1 cache of the calling core. The subsequent instructions hit in the L1 cache. Analogous holds for write operations, except that L1 prefetching is implemented in software.

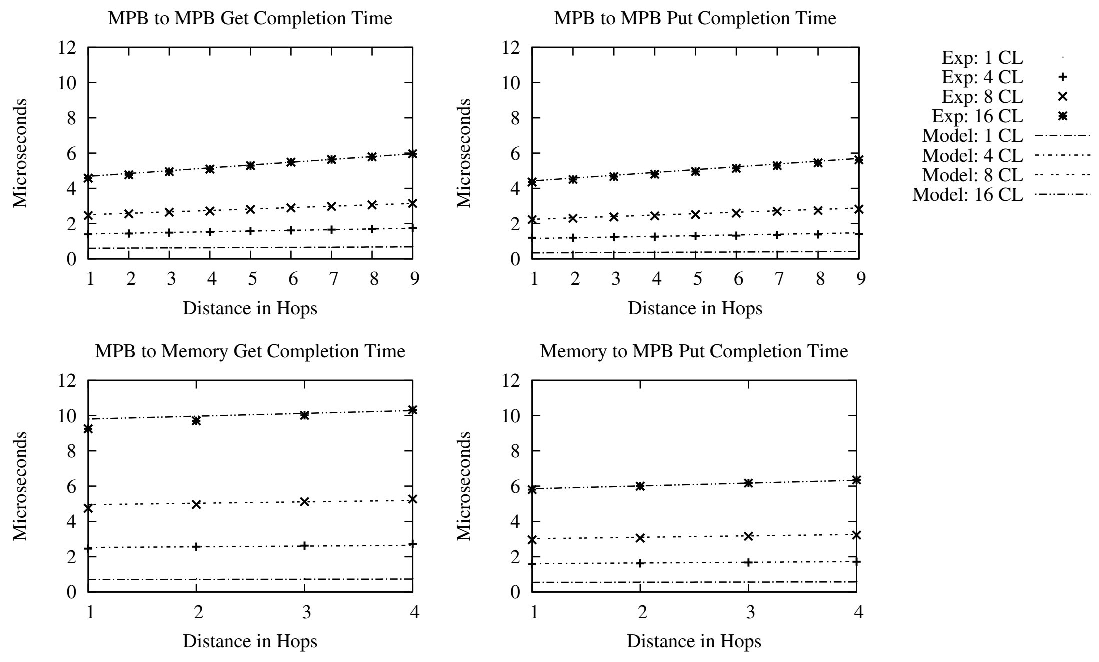


Figure 4.2 – *get* and *put* performance (CL = Cache Line)

MPB read/write

Any read or write operation of a single cache line includes some core overhead o^{mpb}, as well as some mesh overhead which depends on d (the distance between the core and the MPB). We define L^{hop} as the time needed for one packet to traverse one router; it is independent of the packet size. Therefore, the latency of writing one cache line to an MPB is given by Formula 4.1 in Figure 4.1. The write completes when the acknowledgment from the MPB is received, which adds $d \cdot L^{hop}$ (Formula 4.2).

To read one cache line from an MPB, a request has to be sent to this MPB; the cache line is received as a response. Therefore the latency and the completion time are equal (Formula 4.3).

Off-chip read/write

By o_r^{mem} and o_w^{mem}, we represent the constant overhead of reading and writing one cache line from/to the off-chip memory. Note that in the LogP model, an overhead o is supposed to represent the time during which the processor is involved in the communication. We choose to include memory read and write overheads in o_r^{mem} and o_w^{mem} for the sake of simplicity. The latency and the completion time of off-chip memory read/write correpond to Formulas 4.4-4.6, where d represents the distance between the core that executes the read/write operation and the memory controller.

Operation *put*

To model *put* (and later *get*) from MPB to MPB, we introduce o_{put}^{mpb} (respt. o_{get}^{mpb}) to define the core overhead of the *put* (respt. *get*) function apart from the time spent moving data. The corresponding o_{put}^{mem} and o_{get}^{mem} are used for operations involving private off-chip memory. A *put* operation executed by core c reads data from some source and writes it to some destination: the source is either c's local MPB (Formula 4.7) or private off-chip memory (Formula 4.8), and the destination is an MPB. We denote by d^{src} the distance between the data and core c executing the operation, and by d^{dst} the distance between c and the MPB to which the data is written. If c moves data from its local MPB then $d^{src} = 1$. Otherwise, d^{src} is the distance between c and the memory controller. Note also that the P54C cores can only execute one memory transaction at a time: moving a message of m cache lines takes m times the time needed to move one cache line.[2] The latency is a bit lower, since it does not include the acknowledgment of the last cache line written to the remote MPB (Formulas 4.9 and 4.10).

Operation *get*

A *get* operation executed by core c reads data from some source and writes it to some destination: the source is an MPB, and destination is c's local MPB (Formula 4.11) or private off-chip memory (Formula 4.12). We denote by d^{src} the distance between the data and core c executing the operation, and by d^{dst} the distance between c and the MPB to which the data is written. If c moves data to its local MPB, then $d^{dst} = 1$. Otherwise, d^{dst} is the distance between c and the memory controller. In the case of a get operation, latency and completion time are equal.

4.1.2 Model validation

We perform a set of experiments to determine the value of the introduced parameters and to validate our model. Experimental settings are detailed in Section 3.3.1. Figure 4.2 presents with dots the completion time of *put* and *get* operations, as a function of the distance and the message size. The parameter values obtained are presented in Table 4.1. The performance obtained from the model is represented by lines in the same figure. It shows that our model precisely estimates communication performance. Note that, for a given message size, the performance difference between the 1-hop distance (which means accessing the MPB of the other core on the same tile) and the 9-hop distance (maximum distance) is only 30%, since the local overhead of sending a message dominates the cost of one network hop. This justifies our decision to ignore topology issues when designing and evaluating the algorithms.

[2]For this reason, we do not need to use the parameter g of the LogP model.

parameter	L^{hop}	o^{mpb}	o^{mem}_{w}	o^{mem}_{r}	o^{mpb}_{put}	o^{mpb}_{get}	o^{mem}_{put}	o^{mem}_{get}
value (μs)	0.005	0.126	0.461	0.208	0.069	0.33	0.19	0.095

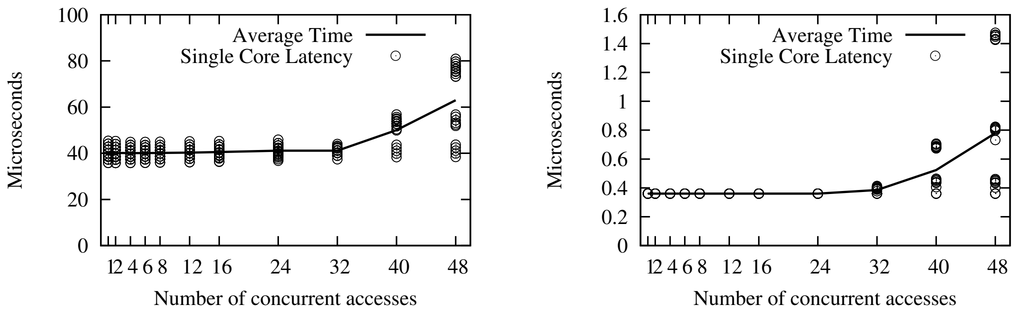
Table 4.1 – Parameters of our model

4.1.3 Contention issues

As mentioned before, the proposed model assumes contention-free execution. Bearing that in mind, we study contention on the SCC to assess the validity domain of the model. We identify two possible sources of contention related to one-sided 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 study, each core has one memory rank for itself and there is no measurable performance degradation even when the 48 cores are accessing their private portions of the off-chip memory at the same time [vTBV+11].

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. This shows that, 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 algorithms presented in Chapter 3: 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, causing most of the writes to be dead. Figure 4.3a shows the impact on latency when increasing the number of cores executing *get* in parallel. Figure 4.3b 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 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 4.3). 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 have observed non-deterministic overheads



(a) Concurrent MPB *get* completion time (128 cache lines)

(b) Concurrent MPB *put* completion time (1 cache line)

Figure 4.3 – MPB contention evaluation

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, which is taken into account by OC-BCAST by introducing the k parameter. They show that up to 24 cores accessing the same MPB do not create any measurable contention. With more than 24 cores accessing the same MPB at the same time, there is visible contention, which degrades *put* and *get* performance and diverges from the established model. Note, however, that this does not necessarily mean that contented scenarios should be avoided by all means, as we saw in Section 3.3 with the asynchronous version of OC-BCAST. Namely, even with contention, the maximum degree of the broadcast tree ($k = 47$) was still most efficient, as it enabled us to have only one expensive interrupt on the critical path.

4.2 Modeling Two-sided RCCE Communication

Before modeling the *RCCE_comm* broadcast algorithms, we need to model the two-sided $send/receive$ primitives from the RCCE library.

The RCCE *send/receive* functions are implemented on top of the one-sided *put* and *get* operations [MVDW10]. The RCCE *send* function puts the message payload from the private memory to the MPB of the sender. The RCCE *recv* function gets the message payload from the MPB of the sender to the private memory of the receiver. Both functions are synchronous: the *send* can only terminate when the corresponding *recv* has finished receiving the data. To synchronize the sender and the receiver, flags are updated after putting and after getting the data. Writing or reading a flag is modeled by a single MPB access.

In the synchronous broadcast algorithms we consider, each communication step includes only pairs of nodes. In other words, there is no sequence in which multiple cores send a message to one core, and that core executes $receive$ for each of them, or vice versa. Therefore, we assume that, for each $send/receive$ pair, $send$ and $receive$ are called simultaneously (there is no waiting time). Under that assumption we can directly model the completion time of a $send/receive$ pair. First, we express it for one chunk of data:

$$\begin{aligned}C_{sr\_chunk}(s) = C_{put}\ mem(s, d^{mem}, d^{mpb}) + C_{get}^{mem}(s, d^{mpb}, d^{mem})\\ + 2 \cdot \left(L_w^{mpb}(d^{mpb}) + C_r^{mpb}(1) + C_w^{mpb}(1)\right)\end{aligned}$$

The term $\left(L_w^{mpb}(d^{mpb}) + C_r^{mpb}(1) + C_w^{mpb}(1)\right)$ corresponds to the time needed to synchronize the sender and the receiver and to reset the corresponding flag.

To transfer a multi-chunk message, the above function is repeatedly called for every chunk, until completion, so the completion time of a $send/receive$ pair of operations is:

$$C_{sr}(s) = \lfloor \frac{s}{M_{rcce}} \rfloor \cdot C_{sr\_chunk}(M_{rcce}) + C_{sr\_chunk}(s \bmod M_{rcce})$$

However, this model is not precise enough for some practical scenarios. Namely, if a core receives a short message and sends it further immediately afterwards, the completion time of the send operation is significantly reduced because the message stays in the on-chip cache (L1 or L2) after the receive operation. To take this into account, we assume a negligible cost of reading from on-chip caches and express the completion time as (m is the message size, in cache lines):

$$\begin{aligned}C_{sr\_chunk\_cache}(m) &= o_{put}^{mem} + m \cdot C_w^{mpb}(1) + C_{get}^{mem}(m, d^{mpb}, d^{mem}) + \\ &\quad + 2 \cdot \left(L_w^{mpb}(d^{mpb}) + C_r^{mpb}(1) + C_w^{mpb}(1)\right)\\ C_{sr\_cache}(m) &= \lfloor \frac{m}{S_{max\_rcce}} \rfloor \cdot C_{sr\_chunk\_cache}(M_{rcce}) + C_{sr\_chunk\_cache}(s \bmod M_{rcce})\end{aligned}$$

When modeling the binomial and scatter-allgather algorithms, we assume that the whole message can fit in the on-chip L2 cache (256 KB per tile, i.e. 128 KB per core). Therefore, the presented models give the upper performance bound.

$$
\begin{aligned}
L^{critical}_{\text{OC-BCAST}}(P,m,k) &= C^{mem}_{put}(m) + O(log_k P)\cdot C^{mpb}_{get}(m) + C^{mem}_{get}(m) \\
&= m\cdot\Big(O(log_k P)\cdot\big(C^{mpb}_r + C^{mpb}_w\big) + \boldsymbol{C^{mem}_r} + \boldsymbol{C^{mem}_w}\Big) && (4.13) \\
L^{critical}_{binomial}(P,m) &= O(log_2 P)\cdot(m\cdot C^{mpb}_w + C^{mem}_{get}(m)) \\
&= m\cdot\Big(O(log_2 P)\cdot\big(C^{mpb}_r + C^{mpb}_w + \boldsymbol{C^{mem}_w}\big) + \boldsymbol{C^{mem}_r}\Big) && (4.14) \\
B_{\text{OC-BCAST}} &= \frac{M_{oc}}{C^{mpb}_{get}(M_{oc}) + C^{mem}_{get}(M_{oc})} = \frac{1}{2C^{mpb}_r + C^{mpb}_w + \boldsymbol{C^{mem}_w}} && (4.15) \\
B_{scatter\_allgather} &= \frac{P\cdot M_{oc}}{P\cdot(C^{mem}_{put}(M_{oc}) + C^{mem}_{get}(M_{oc})) + (2P-3)(M_{oc}\cdot C^{mpb}_w + C^{mem}_{get}(M_{oc}))} \\
&\approx \frac{1}{3C^{mpb}_r + 3C^{mpb}_w + \boldsymbol{C^{mem}_r} + 3\boldsymbol{C^{mem}_w}} && (4.16)
\end{aligned}
$$

Figure 4.4 – Latency and Throughput Model for Broadcast Operations

4.3 Modeling Synchronous Broadcast Algorithms

We analytically compare synchronous OC-BCAST with the binomial tree and *scatter-allgather* algorithms. As before, we consider their implementations from the RCCE_comm library [Cha10]. In the modeling of the algorithms we assume that no time elapses between setting the flag (by one core) and checking that the flag is set (by the other core). Since topology issues are not discussed in this study, we simply consider an average distance $d^{mpb} = 1$ for accessing remote MPBs, and an average distance $d^{mem} = 1$ for accessing the off-chip memory. Other assumptions match the experimental setup presented in Section 3.3. To highlight the most important properties, we divide the analysis in two parts: latency of small messages (OC-BCAST vs. binomial tree) and throughput of large messages (OC-BCAST vs. *scatter-allgather*).

Figure 4.4 summarizes the main formulas that will be derived in this chapter, expressing the expected latency and throughput of the different broadcast algorithms, as a function of the number of processors (P), message size (m) and, for OC-BCAST, the degree of the data propagation tree (k). A quick look at these formulas reveals the main advantage of OC-BCAST: The impact of off-chip memory accesses (C^{mem}_r and C^{mem}_w, marked in bold) on latency and throughput is lower than in the other algorithms. In the following, we derive the formulas step by step, explaining them in more detail.

4.3.1 Latency of short messages

For the sake of simplicity, we ignore notification costs here and concentrate only on the critical path of data movement in the algorithms.

Latency of OC-BCAST

Here we consider messages of size $0 < m \leq 2M_{oc}$. We define the broadcast latency as the time elapsed between the call of the broadcast procedure by the root and the time when the last core (including the root) returns from the procedure.

To express the latency of OC-BCAST, we first model the time it takes for the k children of a core to be aware of the availability of a new chunk in the MPB of their parent. The children are notified using a binary tree, whose degree as a function of k is $\lceil log_2(k) \rceil$. Thus, we get:

$$L_{notify\_children}(k) = \lceil log_2(k) \rceil \cdot (C^{mpb}_{put}(1, d^{mpb}) + L^{mpb}_{put}(1, d^{mpb}) + C^{mpb}_{r}(1))$$

At each level of the tree, it includes the time for completing the notification of the first child ($C^{mpb}_{put}(1, d^{mpb})$), the time to write the flag of the second child ($L^{mpb}_{put}(1, d^{mpb})$), and the time for the child to read it ($C^{mpb}_{r}(1)$).

Similarly, to find out that all of the children have copied the data to their MPBs, a core polls k flags, so it must read each of them at least once:

$$L_{polling\_children}(k) = k \cdot C^{mpb}_{r}(1)$$

Next, we express the depth of the tree D, as a function of k and P (number of cores). In a complete k-ary tree, there are k^i nodes at the i-th level. Therefore, the depth of the tree used by OC-BCAST is the number of levels necessary to "cover" P cores:

$$1 + k + k^2 + ... + k^{d-1} = \frac{k^d - 1}{k - 1} \geq P \tag{4.17}$$

Now we can express D, as the minimum natural number satisfying the above inequation:

$$D = \lceil log_k(P(k-1) + 1) \rceil \tag{4.18}$$

Then we model the latency of the *broadcast_chunk()* function for one chunk based on Algorithm 1 (Section 3.1). As a first step, we model T, the amount of time that the root of the tree, an intermediate node, and a leaf adds to the latency of *broadcast_chunk()*, considering that the message contains at most two chunks. For the root, T_{chunk_root} includes putting the chunk in its MPB ($C^{mem}_{put}(m, d^{mem}, 1)$), and notifying its k children in the message propagation tree:

$$T_{chunk\_root}(m, k) = C^{mem}_{put}(m, d^{mem}, 1) + L_{notify\_children}(k) \tag{4.19}$$

For an intermediate node, it includes the time needed to get the chunk from its parent's MPB

($C^{mpb}_{get}(m, d^{mpb})$), to acknowledge receipt ($C^{mpb}_{put}(1, d^{mpb})$), and to notify its children.

$$T_{chunk\_intermediate}(m,k) = C^{mpb}_{get}(m, d^{mpb}) + C^{mpb}_{put}(1, d^{mpb}) + L_{notify\_children}(k) \tag{4.20}$$

Finally, for a leaf, it includes the time to get the chunk from its parent's MPB ($C^{mpb}_{get}(m, d^{mpb})$), to notify its parent that it has the chunk, and to copy the leaf to its private memory.

$$T_{chunk\_leaf}(m) = C^{mpb}_{get}(m, d^{mpb}) + C^{mpb}_{put}(1, d^{mpb}) + L^{mem}_{get}(m, 1, d^{mem})$$

Now we can express the latency of transferring one chunk of size $m \leq M_{oc}$ between P cores, using a k-ary tree:

$$\begin{aligned} L_{bcast\_chunk}(P, m, k) = {} & T_{chunk\_root}(m,k) + (D-2) \cdot T_{chunk\_intermediate}(m,k) + \\ & + max\big(T_{chunk\_leaf}(m), C^{mem}_{get}(m, 1, d^{mem}) + \\ & + L_{polling\_children}(k)\big) \end{aligned}$$

Depending on the time needed for the last intermediate node to poll the k flags after copying a chunk to its private memory, it can finish later than the last leaf, as expressed by the last term of the formula.

Finally, the latency of OC-BCAST for a message of size $m \leq 2M_{oc}$ depends on size of the second chunk $m' = max(m - M_{oc}, 0)$:

$$\begin{aligned} L_{\text{OC-BCAST}\_short}(P, m, k) = {} & L_{bcast\_chunk}(P, min(m, M_{oc}), k) + \\ & + max\big(T_{chunk\_leaf}(m'), C^{mem}_{get}(m', 1, d^{mem}) + \\ & + L_{polling\_children}(k)\big) \end{aligned} \tag{4.21}$$

Note that $L_{chunk\_leaf}(0) = 0$ because the function is called only if the second chunk exists. Formula 4.21 will be used to compute the latency of OC-BCAST. After necessary expansions in Formula 4.21, we obtain Formula 4.13.

Latency of the two-sided binomial tree

Recall that the binomial broadcast algorithm is based on a recursive tree. The set of nodes is divided in 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. Obviously, the formed tree has $\lceil log_2(P) \rceil$ levels and in each of them the whole message is sent between the pairs of nodes, so the total latency, ignoring the cache effects discussed above, is:

$$L_{binomial\_nocache}(P, m) = \lceil log_2(P) \rceil \cdot C_{sr}(m)$$

Note that, after receiving the message, a core just sends it to other cores in the subsequent recursive invocations. This means that only the first send-receive pair, which involves the root and another core, should be modeled as C_{sr}, whereas the others are modeled as C_{sr_cache}. When this is taken into account, the latency of the binomial algorithm becomes:

$$L_{binomial}(P, s) = C_{sr}(m) + (\lceil log_2(P) \rceil - 1) \cdot C_{sr\_cache}(m) \tag{4.22}$$

Formula 4.22 will allow us to compute the latency of the binomial broadcast algorithm. By expanding C_{sr} and C_{sr_cache}, Formula 4.14 is obtained.

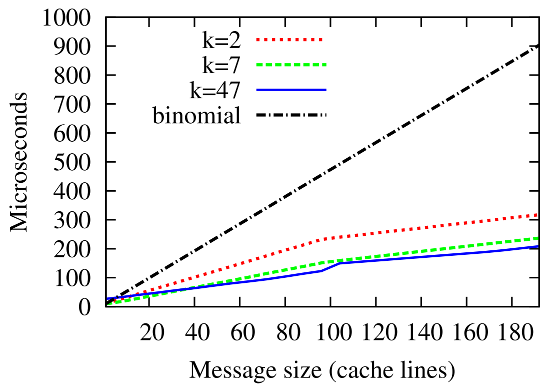
Latency comparison

Now we can directly compare the analytical expressions for the two broadcast algorithms. Figure 4.5a plots latency as a function of the message size. For OC-BCAST, different values of k are given ($k = 2$, $k = 7$, $k = 47$). As we can see, OC-BCAST significantly outperforms the binomial tree algorithm. This is not surprising, given that in Formula 4.13, which represents the latency of OC-BCAST, there are only two off-chip memory operations ($C^{mem}_{r/w}$) on the critical path for one chunk, regardless of the number of cores P. This is not the case for the binomial algorithm, represented by Formula 4.14. Moreover, as k increases, the number of on-chip copy operations on the critical path reduces for OC-BCAST. Further, the data presented in Figure 4.5 is almost identical to the result obtained by running experiments (Figure 3.2a, Section 3.3). The small difference can be explained by not taking topology into account.

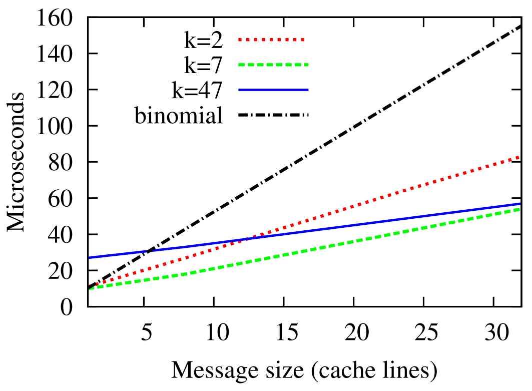
The advantage of OC-BCAST increases further when increasing the message size because of double buffering and pipelining. It can be observed in Figure 4.5a that the slope changes for messages larger than $M_{\text{OC-BCAST}}$ (96 cache lines). In Figure 4.5b, we can also notice that OC-BCAST-47 is slower for very small messages in spite of having only two levels in the data propagation tree (the root and its 47 children). The reason is that a large value of k increases the cost of polling (recall that polling is expensive because of a bug that requires us to access the local MPB through the router). For $k = 47$, the root has 47 flags to poll before it can free its MPB.

4.3.2 Throughput of large messages

Now we model throughput achievable with large messages. We compare OC-BCAST with the RCCE_comm *scatter-allgather* algorithm. To simplify the modeling, we assume a message of



(a) Modeled broadcast latency



(b) Modeled broadcast latency (zoom-in)

Figure 4.5 – Analytical latency comparison. OC-BCAST ($k = 2, 7, 47$) vs. two-sided binomial tree

size $P \cdot M_{oc}$. With OC-BCAST, such a message is transferred in P chunks of size M_{oc}. *Scatter-allgather* transfers the same message by dividing it into P slices of size M_{oc}.

Throughput of OC-BCAST

Messages are now large enough to fill the pipeline used by OC-BCAST. For such messages, every node executes a loop, where one chunk is processed in each iteration. We compute the completion time of the *broadcast_chunk()* function for such an iteration considering the root, an intermediate node and a leaf.

$$\begin{aligned}
C_{chunk\_root}(k) &= L_{polling\_children}(k) + C_{put}^{mem}(M_{oc}, d^{mem}, 1) + 2 \cdot C_{put}^{mpb}(1, d^{mpb}) \\
C_{chunk\_intermediate}(k) &= L_{polling\_children}(k) + C_{r}^{mpb}(1,1) + C_{get}^{mpb}(M_{oc}, d^{mpb}) \\
&\quad + 3 \cdot C_{put}^{mpb}(1, d^{mpb}) + C_{get}^{mem}(M_{oc}, 1, d^{mem}) \\
C_{chunk\_leaf} &= C_{r}^{mpb}(1,1) + C_{get}^{mpb}(M_{oc}, d^{mpb}) \\
&\quad + C_{put}^{mpb}(1, d^{mpb}) + C_{get}^{mem}(M_{oc}, 1, d^{mem})
\end{aligned}$$

The time to fully process a chunk is determined by the slowest node. Therefore, based on the slowest completion time, the throughput is easily expressed as:

$$B(k) = \frac{M_{oc}}{max(C_{chunk\_root}(k), C_{chunk\_intermediate}(k), C_{chunk\_leaf})} \tag{4.23}$$

This is how Formula 4.15 is obtained. Note that no matter what the values of the parameters are, the leaf is always faster than the intermediate node. However, we keep C_{chunk_leaf} in the above formula, because there is a special case with no intermediate nodes, for the maximum value of k (47 for the SCC).

Throughput of two-sided scatter-allgether

The *scatter-allgather* broadcast algorithm has two phases, as discussed in Section 3.3.2. During the *scatter* phase, the message is divided into P equal slices[3] 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.

Scatter The *scatter* phase is done using a recursive tree, similar to the one used by the binomial tree algorithm. However, in this case we do not transfer the whole message in each step, but only a part of it. Thus, expressed recursively, the completion time of the scatter part is:

$$\begin{aligned}
&C_{scat}(P, m_s) = C_{sr}(\lfloor\frac{P}{2}\rfloor \cdot m_s) + C_{scat}(\lceil\frac{P}{2}\rceil, m_s), \text{ for } P > 2\\
&C_{scat}(2, m_s) = C_{sr}(m_s)\\
&C_{scat}(1, m_s) = 0
\end{aligned}$$

Overall, the scatter phase does not benefit from temporal locality, which can be easily noticed by observing the actions of the root. Namely, the root sends different parts of the message in every recursive call (first, a half of the original message, then, a half of the non-cached half of the original message etc). Although other nodes might reuse data, the performance is dictated by the slowest node, which is the root. Therefore, the given formula holds even in the presence of caches.

Allgather As already mentioned, the *allgather* phase implemented in RCCE_comm uses the Bruck algorithm [BHU+97]: At each step, core i sends to core $i-1$ the slices it received in the previous step. In each iteration of the *allgather* algorithm, a core i sends one slice to core $i-1$ and receives one slice from core $i+1$. Therefore, the completion time of this phase, without considering the caches, is:

$$C_{allgath\_nocache}(P, m_s) = 2 \cdot (P-1) \cdot C_{sr}(m_s)$$

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

Note that in each of $P-1$ iterations, except in the first one, a core sends the slice of the message received in the previous iteration. In the first iteration, all cores have the corresponding slice of the message ready in the cache (thanks to the scatter phase), except the root, which has never accessed its slice of the scattered message before. Therefore, in the cache-aware model, each but one occurrence of C_{sr} should be replaced by C_{sr_cache}:

$$C_{allgath}(P, m_s) = C_{sr}(m_s) + (2P-3) \cdot C_{sr\_cache}(m_s)$$

Scatter-allgather Finally, the completion time of the *scatter-allgather* algorithm run on P processes with a message of size m is:

$$C_{scat-allgath}(P, m) = C_{scat}(P, \frac{m}{P}) + C_{allgath}(P, \frac{m}{P}) \tag{4.24}$$

There is no pipelining in this algorithm, so the throughput can be easily expressed by dividing the message size by the derived completion time. Thus we obtain Formula 4.16.

Throughput comparison

Table 4.2 gives throughput estimations, calculated from the above formulas and the parameters given in Table 4.1. As with latency, the numbers are very close to those experimentally obtained in Section 3.3 (Figure 3.2b). Regardless of the choice of k, the throughput is almost three times better than that provided by two-sided *scatter-allgather*. The additional terms in Formula 4.16 compared to Formula 4.15 explain the performance difference in Table 4.2: The number of write accesses to the MPBs and to the off-chip memory (C_w^{mpb} and C_w^{mem}) with OC-BCAST is three times lower than that of the *scatter-allgather* algorithm based on two-sided communication. The number of read accesses is also reduced.

Looking back at experimental results from Figure 3.2b, it can be noticed that the only significant difference with respect to the analytical predictions is for OC-BCAST with $k = 47$ (the throughput is about 16% lower than predicted). It is now clear that MPB contention is one of the sources of the observed performance degradation. Thus, large values of k might turn out to be inappropriate at large scale, since the gain in parallelism could be paid by a more significant loss related to contention.

4.4 Summary

This chapter has introduced an analytical model for the Intel SCC, which is used to derive formulas for the latency and throughput of the synchronous OC-BCAST, binomial tree and

Algorithm	OC-Bcast k=2	OC-Bcast k=7	OC-Bcast k=47	scatter-allgather
Throughput (MB/s)	35.22	34.30	35.88	13.38

Table 4.2 – Broadcast algorithms: analytical throughput comparison

scatter-allgather broadcast algorithms. The presented analysis confirms that our broadcast implementation based on one-sided operations brings considerable performance benefits, in terms of both latency and throughput. The main reason is the reduced number of off-chip memory accesses on the critical path, with respect to the two-sided algorithms. Comparing the analytical results with those from the experimental study presented in Section 3.3 reveals that the derived model enables us to estimate latency and throughput very precisely in contention-free execution. The instantiation of the presented model assumes that access to a local MPB requires passing through the local router, because of a hardware bug on the SCC. In a configuration where the MPB local to a core is directly accessed, the advantage of OC-Bcast over the alternatives would be even higher.

It would be interesting to extend the presented model to capture the existence of contention. This would enable us to look at the choice of a tree degree (k), which involves a parallelism-contention tradeoff, from a purely analytical perspective. Contention models have been studied before for shared-memory algorithms [DHW97, BWKMZ12].

Fast Mutual Exclusion on Standard and Emerging Processors

II

5 Background and Preliminaries

5.1 Motivation

A vast majority of today's software is written for shared-memory computers, as mentioned in Chapter 1. When multiple threads share data and can access it concurrently, ensuring mutual exclusion is one of the key challenges. The most common approach to solving this problem is by using *locks*: A thread acquires a lock before entering its critical section, and releases it afterwards. More complex synchronization primitives, such as *reader-writer locks* and *monitors* typically use simple locks as the basic building block.

Locks, however, may cause scalability issues. Ideally, when running a program on a manycore machine, we aim for *linear speedup*: If the program is run on n cores, we would like it to execute n times faster than on a single core. However, in reality, there are fundamental laws of parallelism that dictate how well a parallel program can scale. If only one thread at a time can execute a portion of code, which is the case when a lock is used to access shared data, that part of code is inherently *sequential*, i.e., it is not parallelizable. If a program contains a fraction of sequential code, no matter how small, an implication of *Amdahl's law* [Amd67, Gus88] is that the achievable speedup is limited. In other words, after some point, additional cores will not increase program execution speed. The larger the sequential part, the fewer cores the program can benefit from. It is important to point out that sequentiality is not only a problem of locks: It also exists when other synchronization means are used, such as nonblocking data structures [Her91] or transactional memory [HM93].

Therefore, to leverage the increasing number of cores per chip, it is necessary to make the sequential portion of a program as short as possible. How this is done largely depends on the problem we want to solve. Some problems are known to have *embarrasingly parallel* solutions, which scale perfectly with the number of cores [HB09]. Others, however, are not easily parallelizable, and require substantial effort to make them faster on a manycore processor [LDT$^+$12]. While there are countless problem-specific techniques and optimizations, we will focus on improving the performance of mutual exclusion on a critical section, as a problem-independent way of addressing one source of sequential bottlenecks, thus enabling concurrent software to scale better. More precisely, our goal in this part of the thesis is to minimize overheads of executing critical sections, using off-the-shelf hardware. At the same time, we strive to keep the programming interface as simple as possible, enabling non-experts to easily write efficient implementations of arbitrary concurrent objects.

Our study reveals that the most efficient known algorithms for critical section execution are, once implemented on a commodity processor, limited by the performance of the cache coherence protocol. We propose two ways to overcome this problem, thus improving performance. The first is leveraging hardware support for message passing, existing in some emerging processors. The second is relying on special instructions and characteristics of commonplace processors, with the goal of mimicking hardware support for message passing. Before detailing our contributions (Section 5.4), we present the assumed system model (Section 5.2) and give some background on mechanisms for efficient critical section execution. An overview of related work is given in Section 5.5.

5.2 System Model

We assume a set of T sequential threads that can communicate both by issuing operations to coherent shared memory and by directly exchanging messages. It is assumed that the system is fault-free: threads do not crash and every thread eventually makes progress.

Cache coherence. In the cache-coherent (CC) shared-memory model, threads operate on cached copies of shared variables. We assume a model adapted from the one by Sorin et al [SHW11]. A processor chip is composed of single-threaded cores. Each core has its local, private data cache. All cores have access to a globally shared memory through an interconnection network. The cache coherence protocol maintains the *single-writer-multiple-reader* invariant: At any given time, either a single core has read-write access to a cached variable, or some cores have read-only access [SHW11]. *Remote Memory References* (RMRs) are accesses to shared variables that involve communication on the interconnection network. In this model and assuming write-back caches, reading a shared variable generates an RMR if the core does not hold a copy of the variable in either mode. Writing a shared variable generates an RMR if the core does not hold a copy of the variable in read-write mode.

Shared-memory operations. The memory is an array of 64-bit locations. Similarly to most related studies, we assume that the memory is sequentially consistent. Supported operations on a memory location a are the standard *read(a)*, *write*(a, v) operations as well as some atomic read-modify-write operations, namely *FAA*(a, v) (fetch-and-add), *SWAP*(a, v) and *CAS*(a, v_{old}, v_{new}) (compare-and-set[1]), with their standard definitions.

[1] The variant of compare-and-swap that returns a boolean.

Message-passing operations. Each thread has an incoming FIFO message queue that stores 64-bit values (message queue hereafter). Supported message-passing operations are $send$, $receive$ and $is\_msg\_queue\_empty$. The operation $send(i, M)$ puts message M, which is a set of values $v_1, v_2, ..., v_n$, in the message queue of thread t_i. The $send$ operation is asynchronous, *i.e.*, it may return before M is placed in the destination message queue. Message transmission time is bounded but unknown, *i.e.*, the time between a call to $send$ and the moment when the message is placed in the corresponding queue is arbitrarily, but finitely long. If $|M| > 1$, values are placed in the destination message queue in the order $v_1, v_2, ..., v_n$. The operation $receive(k)$ returns k values from the head of the local message queue. If there are fewer than k values in the queue, the operation blocks until k values are available. Operation $is\_msg\_queue\_empty()$ returns true if the local message queue is empty.

5.3 Critical Section Execution on a Cache-Coherent Processor

This section details existing techniques for the efficient execution of critical sections on cache-coherent processors. It explains how their performance is influenced by the underlying CC protocol. We address two common ways to ensure mutual exclusion: classic locks and delegation.

5.3.1 Classic locks

Critical sections are usually implemented using locks. In this context, the basic technique to improve scalability is to introduce *local spinning* [MCS91]: Each thread polls on a different variable which stays in its local cache. As a result, the number of RMRs per lock acquisition is constant, i.e., it does not depend on the number of threads competing for the lock. This ensures that the performance of the lock does not deteriorate with increased contention, which is a problem faced by simple locks without local spinning, such as *test-and-set* [HS08].

In a lock implementation, local spinning is most often ensured by maintaining an explicit or imlicit *queue* of threads that wait to acquire the lock. When the thread t holding the lock releases it, t lets the next thread t' in the queue acquire it, by writing to a shared variable t' is spinning on. Because of this, such locks are often referred to as *queue locks*. Irrespective of the implementation, a queue lock handover on a cache-coherent system has at least two RMRs on the critical path: One from the thread that releases the lock, to write that the next thread can proceed, and one from that next thread, to read the written information. This is depicted in Figure 5.1, where thread a is handing over a lock to thread b, by setting $flag\_b$. Since b is spinning on $flag\_b$, an RMR is triggered when a tries to write to $flag\_b$, in order to bring $flag\_b$ in read-write mode in the cache of the core where a is running, and invalidate the cached copy b has. Subsequently, reads from b do not hit in the local cache anymore, which triggers another RMR, to bring a read-only copy of $flag\_b$ in the cache of the core executing b. Note that the two RMRs can partially overlap, since the second one can be triggered as soon as $flag\_b$ is invalidated in the cache of b. Nevertheless, each of them typically involves several

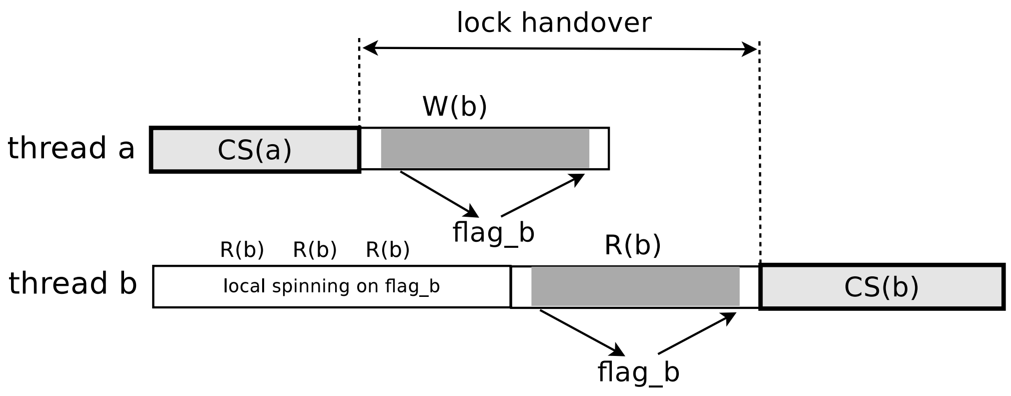


Figure 5.1 – Classic lock handover – shared-memory implementation; $R(i)$, $W(i)$ – reading from (resp. writing to) the flag of thread i; $CS(i)$ – corresponding critical section; dark grey – stalls (due to RMRs)

message exchanges between cores, cache controllers, or other implementation-dependant agents, which significantly contributes to the lock handover time.

Apart from these synchronization RMRs, each critical section execution likely implies additional overheads, inherent to locks. Namely, there are RMRs inside the critical section itself: The data protected by the lock keep bouncing between caches, as every thread accessing them brings it to its local cache, possibly invalidating the other copies. Also, on architectures with weak memory consistency models, every lock handover necessitates expensive memory fences, to make sure that the new lock holder has the most recent copy of the data protected by the lock.

5.3.2 Delegation

Delegation is a way to avoid the data bouncing problem inherent to classic locks. The key idea is that, instead of moving the data associated with a CS to the core that wants to execute the CS, the CS is executed on the core where the data are located. We can identify two approaches that exploit this idea: the server approach [LDT$^+$12, CCPG13], and the combiner approach [FK12, HIST10, OTY99, FK11, KSW14].

Remote Core Locking (RCL) [LDT$^+$12] is an efficient implementation of the server approach. A non-application thread (the server) is in charge of executing CSes. Application threads (clients) send requests to the server to execute a critical section on their behalf. Assuming that data accessed inside the CSes are never accessed by application threads outside the CSes, these data remain in the cache of the server, ensuring that the number of RMRs during CS execution is minimized. Ideally, the only RMRs that remain on the critical path of the CS execution are the ones related to synchronization between the clients and the server. Figure 5.2 illustrates the execution of an RCL server. For client-server communication in RCL, each client thread has a dedicated cache line, which it uses as a bi-directional channel. When client i wants to execute

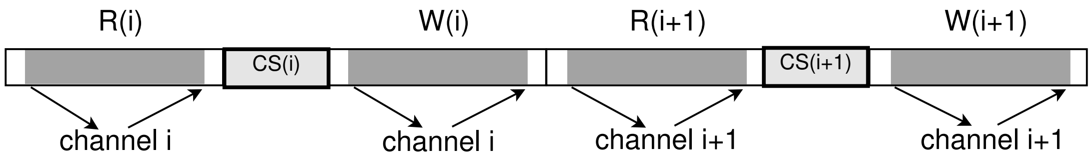


Figure 5.2 – Mutual exclusion server – shared-memory implementation; $R(i)$, $W(i)$ – resp. reading from, writing to the channel of client i; $CS(i)$ – corresponding critical section; dark grey – server stalls (due to RMRs)

a CS, it writes its request to the cache line $channel_i$, and then spins on that cache line until it receives a reply from the server. The server first reads the request from $channel_i$. Since the last access to $channel_i$ was from client i writing the request, this read triggers an RMR (server stalls are represented in dark grey). Then, the server executes the critical section. Finally, it writes to $channel_i$ to inform the client that the request has been processed. This write triggers another RMR to invalidate the client's copy of the cache line. The figure assumes high load, *i.e.*, the server is never idle, and shows that in this case there are two RMRs at the RCL server per CS. Note that Figure 5.2 is somewhat simplified, since it assumes the instructions are not at all overlapped. On a real processor, the different RMRs might partially overlap, depending on the memory consistency model and other features of the processor at hand, resulting in fewer CPU stalls. Nevertheless, these RMRs remain an important source of overhead even on a processor with weak memory consistency (see Section 6.2).

While keeping similar performance benefits, the combiner approach does not require dedicated servers [HIST10]. When a thread gets a lock associated with a CS, it becomes a *combiner* and executes operations of other threads that are waiting to access this CS, in addition to its own. To prevent the combiner from starving if the number of operations of other threads to execute is high, the combiner role is handed over to another thread when the current combiner has served a predefined number of requests. CC-SYNCH [FK12] is, to our knowledge, the most efficient combiner-based approach. Since the combiner changes over time, the synchronization mechanism is more complex than in RCL. Nevertheless, with a thread acting as a combiner, CC-SYNCH is similar to RCL with respect to RMRs: It generates one RMR to read a request from another thread, and then generates another RMR to inform that thread that the operation has been performed.

The server-based approach has the advantage of being simple and very efficient in cases where a small number of clearly identified CSes are highly contended [LDT$^+$12]. On the other hand, combining is more flexible, which comes at the expense of requiring more complex synchronization between threads. Indeed, combining automatically adapts to the load: If a CS is highly contended, all the CPU cycles of one core will be temporarily allocated to it, but if no thread tries to execute a CS, no resources are consumed.

Both with RCL and CC-SYNCH, only two RMRs related to thread synchronization remain on the critical path as far as the server is concerned. These two RMRs, however, can have a big impact on throughput if the code to execute in the CS itself contains few or no RMRs.

It should be noted that, although delegation is efficient when contention is high, it has some disadvantages compared to classic locks. First, it imposes some usability constraints. In particular, since the thread executing the CS is usually not the one requesting it, access to thread-specific state (thread-local and stack variables) and performing any thread-specific work within a CS is harder. Second, when it comes to performance, simple classic locks might be more efficient if contention is low [DGT13]. We will briefly compare the performance of classic locks and delegation in Section 6.2.3.

5.4 Contributions

A way to circumvent the limitations of cache coherence in thread synchronization is direct, explicit exchange of messages between threads. Indeed, hardware support for message passing coexists with coherent shared memory in some modern *hybrid* processors. With such hybrid architectures comes a big design space for synchronization primitives, as solutions that rely both on shared memory and message passing can be devised. We come up with such solutions, implement them on the Tilera TILE-Gx hybrid processor, and then use the lessons learned to improve synchronization performance even on shared-memory-only architectures. In more detail, our contributions are the following:

- **A hybrid lock.** We propose HYBLOCK, a lock that takes advantage of the hybrid nature of emerging processors: The lock state is kept in shared memory, but the lock is handed over between contending threads using message passing. In this way, HYBLOCK retains the classic lock interface, but promises superior performance with respect to scalable shared-memory-only queue locks. Moreover, spinning on variables is completely replaced by blocking on a local message queue, which enables energy saving.
- **A hybrid combining algorithm.** Our findings indicate that even state-of-the-art delegation algorithms, such as RCL [LDT+12] and CC-SYNCH [FK12], waste much time in CPU stalls resulting from activities related to cache coherence. To overcome this problem, we take advantage of hardware message passing and devise HYBCOMB, a universal construction based on the combining technique. Similarly to HYBLOCK, HYBCOMB is a *hybrid* algorithm that relies both on cache-coherent shared memory and message passing for synchronization: Message passing is used to exchange requests and responses between the combiner and other threads, while shared memory is used to manage combiner identity (which would be complex and inefficient to do using message passing). Besides HYBCOMB, we also present MP-SERVER, a straightforward, but very efficient and insightful adaptation of server-based delegation to systems with hardware message passing.
- **Delegation optimizations for shared-memory-only processors.** Although hardware support for message passing is very useful for improving synchronization performance,

many contemporary processors, including the dominant x86 architecture, still do not provide it. As a result, synchronization needs to be implemented entirely over shared memory, even when the concept being implemented lends itself more naturally to message passing, which is the case with server-based delegation. Using insights from the study with hardware message passing, we explore how the performance of delegation over cache-coherent shared memory can be improved by taking into account the subtleties of the underlying cache coherence protocol. We show that a significant throughput increase is achievable by employing simple, but inobvious and even counterintuitive optimizations. In a nutshell, we emulate hardware message passing (i) by avoiding collisions of hardware prefetchers and spinning threads and (ii) by using non-temporal store instructions.

- **Detailed performance evaluation.** All the aforementioned algorithms and optimizations have been implemented and evaluated side by side with their most efficient known alternatives. In particular, we have implemented HybLock, mp-server and HybComb on the Tilera TILE-Gx, comparing them with classic locks (MCS [MCS91], CLH [Cra93, MLH94]) and delegation (CC-Synch [FK12], RCL [LDT$^+$12]). We have carried out a similar comparison on two x86 processors from Intel and AMD, this time implementing our optimized delegation over shared memory. The evaluation on all of the platforms includes implementations of concurrent counters, queues and stacks, and demonstrates that a significant performance improvement is achievable with our strategies.

5.5 Related Work

In Section 5.3 we focused on the bottlenecks of locks and delegation over cache-coherent shared memory. Here we revisit delegation in a more global context. In addition, we provide a brief overview of work studying message passing and other hardware extensions as a means to implement more efficient synchronization.

5.5.1 Delegation

As already mentioned, the simplest way to implement delegation is using a dedicated server. Besides RCL [LDT$^+$12], detailed in Section 5.3, Cleary *et al* [CCPG13] also exploit the server approach, but apply it to *asymmetric synchronization*, where one thread executes the CS much more often then the others. Dedicated server threads have also been demonstrated useful in the design of different concurrent objects [MZK12, CGH13]. Note also that the server approach is not used only in the context of mutual exclusion: It has also been proposed as a way to design scalable message-passing operating systems [BBD$^+$09, WA09].

When it comes to combining, the different algorithms [FK12, HIST10, KSW14, OTY99] mainly differ in the way pending requests are managed. However, none of them considers the use of message passing. Additionally, the combining algorithm by Klaftenegger *et al* [KSW14] enables a client thread to return without waiting for the combiner to execute its request (if the request has no return value). In such cases, one RMR can be removed from the critical path of the server compared to other combining approaches. The optimizations we propose in Chapter 7 are complementary to their contribution and also work when clients need to wait for the result of their operations. It has also been demonstrated that shared-memory combining can be efficiently implemented in a multiprogramming environment by batching requests from co-located threads [FK14].

Experimental comparisons of delegation and locking techniques over CC shared memory have been conducted [CGH13, DGT13]. Results show that for data structures where fine-grained locking can be efficiently applied (*e.g.*, hash tables with large number of buckets), state-of-the-art locking solutions remain most efficient under high contention. In other cases, delegation is shown to perform better. The algorithms and optimizations we propose further increase the performance of both classic locks and delegation.

5.5.2 Hardware extensions for synchronization

Many studies propose special hardware for improving synchronization performance. Suleman *et al* [SMQP10] explore server-based delegation over dedicated hardware and evaluate how much chip real estate should be used for the server core. Token-based messaging over a dedicated network has also been considered as a way to improve critical section execution performance [AFA11]. Before software-only queue locks were devised, special queuing hardware was proposed to cope with the scalability problems of simple locks such as TAS [GVW89]. Different flavors of hardware for producer-initiated communication have been proposed [PYK$^+$13, ASHAA97], with the idea of hiding communication latency by proactively *pushing* data to the consumer, instead of waiting for the consumer to fetch them. In MP-SERVER and HYBCOMB, we accomplish the same by using hardware message passing: The server/combiner thread reads requests from a local queue, instead of fetching them from remote caches. The DeNovo project [CKS$^+$11, SKA13, SA15] aims to holistically rethink coherence in order to eliminate common scalability issues and reduce complexity.

As for message passing in particular, Sanchez et al. [SYK10] propose a hardware extension for core-to-core message exchange, Asynchronous Direct Messages (ADM), and show how it can be used to build efficient schedulers for fine-grained parallelism. ADM is very similar to the implementation of message passing on the TILE-Gx processor used in our study. In the 90's, Herlihy et al. showed, by simulating MIT's Alewife processor, that message-passing implementations of counting networks and combining trees are more efficient than their shared-memory counterparts [HLS95]. The message-passing features of the Intel SCC [HDH$^+$10] and Tilera [Til14] processors have been used in the implementation of transactional memory [GGT12] and key-value stores [BFPS11].

5.6 Outline

Our contributions are presented in the next two chapters. Chapter 6 details and evaluates HybLock, HybComb and mp-server, our algorithms that rely on the existence of hardware message passing side by side with cache-coherent shared memory. Chapter 7 is on improving the performance of delegation without hardware support for message passing.

6 Leveraging Hardware Message Passing for Efficient Critical Section Execution

In this chapter we present our new approaches to locking and delegation (Section 6.1) and evaluate them on a hybrid processor (Section 6.2). Some additional remarks are given in Section 6.3, and Section 6.4 summarizes the chapter.

6.1 Improved Mutual Exclusion Algorithms

We discuss three ways to leverage hardware support for message passing to execute critical sections efficiently. We start by presenting a hybrid lock that uses both shared memory and hardware message passing for thread synchronization. Then we move to delegation: In this context, we first explain how hardware messaging can be beneficial by addressing the client-server approach. Second, we present a novel hybrid combining algorithm that takes advantage of the given insights.

6.1.1 The classic lock (HybLock)

We saw in Section 5.3 that even the most efficient classic locks require at least two RMRs during lock handover, since the next thread to take the lock is signalled using a flag in shared memory. If the hardware allows direct exchange of messages, it is natural to think of an algorithm that would use a message to hand over the lock to the next thread: Instead of writing to and spinning on a flag, the current lock holder should signal the next one by sending a message, as shown in Figure 6.1. In this way, the overhead of cache coherence is removed from the lock handover (although one full message latency remains on the critical path). Still, shared memory remains a convenient means to implement synchronization. Indeed, to implement mutual exclusion over message passing only, one can use a distributed mutual exclusion algorithm, but they are typically expensive in terms of number of messages. For example, even the well-known NTA algorithm [NTA96] necessitates at least $O(\log n)$ messages over a long run in a system with n processes, whereas $O(1)$ is easily achievable if a shared memory is also present, as we will now demonstrate.

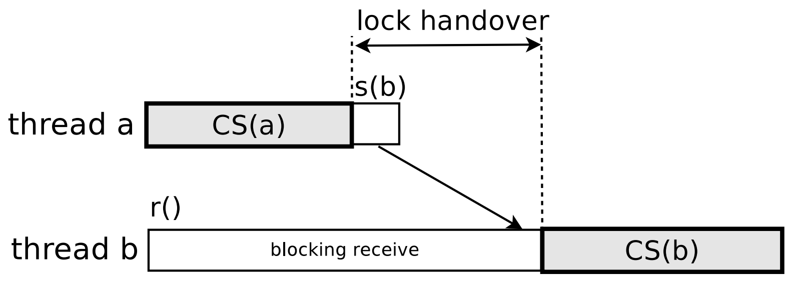


Figure 6.1 – Classic lock handover – message-passing implementation; $r()$ – receive message; $s(t)$ – send message to thread t

HYBLOCK is designed to take advantage of both message passing and cache coherent shared memory. When the lock is not contended, threads simply acquire and release it by modifying a shared-memory variable. In the event of contention, the lock is handed over between threads using message passing, avoiding synchronization RMRs and thus improving throughput. More precisely, threads form a logical queue. Upon finishing its critical section, every thread directly transfers the lock to the next one in the queue just by sending a message, without modifying the shared state.

The pseudocode of HYBLOCK is given in Algorithm 3. The lock state is represented by an integer. A special value, *UNLOCKED*, denotes that the lock is free. Otherwise, this variable contains the id of the last thread requesting the lock. Besides, every thread has a private (thread-local) variable $next$, used in some execution scenarios to store the id of the next thread the lock will be handed over to.

To acquire the lock, a thread t first executes a SWAP operation on the lock state, writing its id and returning the previous value (line 8). If that value is *UNLOCKED*, thread t takes the lock and no message passing is needed. Otherwise, t is queued after $prev$, which was the previous thread to request the lock, *i.e.*, execute the SWAP at line 8. Thread t then sends a message with its id to $prev$, thus letting $prev$ know it should eventually reply with *ALLOW* to hand over the lock. Thread t then waits at line 11 for that to happen. When *ALLOW* comes in, t has successfully acquired the lock. Note, however, that the incoming message might not be *ALLOW*: As soon as t has executed the SWAP at line 8, another thread may request the lock, in which case it will line up directly after t, sending t a message at line 10. In this case, t stores the id of its successor in $next$, for use when it releases the lock.

When t decides to release the lock, it first checks if $next$ contains a valid thread id, in which case the appropriate thread is contacted and $next$ is reset. If the identity of the next thread is not known, this means that either (i) no thread has lined up to take the lock after t, or (ii) its message is yet to be received. Next, if the message queue is empty, there is a chance that (i) is satisfied, so t can try to unlock using a CAS on the lock state (line 21). If the CAS succeeds, there were no contending threads and the lock is successfully released. On the other hand, if the message queue is not empty, or the CAS fails, we can conclude that the message from the successor has either arrived, or will eventually arive. Therefore, t blocks, waiting for the message from its successor, before finally informing the successor that it can acquire the lock.

Properties HYBLOCK achieves a number of properties desirable in theory and practice, outlined as follows:

- **Fast contention-free execution.** HYBLOCK is very lightweight in absence of contention, and one operation on shared data is sufficient to both take and free the lock. With this respect, HYBLOCK behaves similarly to MCS [MCS91]: Lock acquisition consists of a single SWAP, whereas release is a (successful) CAS.
- **Fast contended execution.** Unlike shared-memory only locks, based on spinning, HYBLOCK is able to remove RMRs from the critical path of lock handover. Indeed, when multiple threads are waiting for the same lock, the handover is done without any operations on shared data, i.e., the CAS at line 21 is never executed, only message-passing operations.[1]
- **Fairness.** Obviously, threads are served in the order of arrival (see line 8), as with queue locks based only on shared memory.
- **Energy-friendliness.** The use of synchronous message passing instead of spinning on shared-memory variables enables avoiding wasted CPU cycles, thereby potentially improving energy efficiency.

When it comes to limitations of HYBLOCK, it is clear that nested locking is not supported out of the box, since messages from different lock instances can arbitrarily interleave. If desired, this limitation can be removed by associating special tags with messages, as well as by replacing the local $next$ variable with a local stack, which would keep one $next$ value for every level of nesting. This, however is likely to decrease performance.

It should also be noted that lock handover performance, although improved, is still limited by the latency of the message from the current to the next lock owner. We will shortly see that with delegation even this message latency can be removed from the critical path.

6.1.2 The server approach (MP-SERVER)

A client-server approach, such as RCL, is a natural fit for message passing. Indeed, RCL's client-server communication layer can be seen as an implementation of message passing over shared memory. Instead, we simply leverage hardware message passing support to implement client-server communication. We refer to this solution as MP-SERVER. Based on the model

[1]Note that the throughput bottleneck is not necessarily lock handover. It can also be the execution of SWAP (line 8), depending on how atomic operations are implemented.

Algorithm 3 HYBLOCK locking algorithm – code for thread id

```
1: const UNLOCKED {* lock is not taken *}
2: const ALLOW {* next thread can take lock *}
3: type Lock int

Local Variables:
4: next: int ← ⊥

5: lock_init(lock: Lock)
6:     lock ← UNLOCKED

7: lock_acquire(lock: Lock)
8:     prev ← SWAP(lock, id)
9:     if prev ≠ UNLOCKED then
10:        send(prev, id)
11:        tmp ← receive(1)
12:        if tmp ≠ ALLOW then
13:            next ← tmp
14:            receive(1)

15: lock_release(lock: Lock)
16:     if next ≠ ⊥ then
17:         send(next, ALLOW)
18:         next ← ⊥
19:         return
20:     if is_msg_queue_empty() then
21:         if CAS(lock, id, UNLOCKED) then
22:             return
23:     tmp ← receive(1)
24:     send(tmp, ALLOW)
```

introduced in Section 5.2, Figure 6.2 explains why MP-SERVER may have better performance than its shared-memory counterpart. Compared to Figure 5.2, stalls can be avoided for two reasons. First, the server reads requests from the local message queue, without any remote actions that would cause it to stall. Second, the server does not wait for the actual message transmission to take place when it sends a response. When and how the messages are actually sent to their destinations is the responsibility of the underlying hardware message passing implementation. Therefore, if hardware message passing is used, we expect to be able to *completely remove* stalls related to synchronization from the critical execution path.

6.1.3 The combiner approach (HYBCOMB)

We now detail HYBCOMB, our combining algorithm tailored to take advantage of message passing. We start by describing the main principles of combining techniques over shared memory, to identify how message passing can be used to improve performance.

Main principles In combining algorithms, threads interact for two purposes: (i) *electing* a combiner; (ii) exchanging information between the combiner and threads that have operations

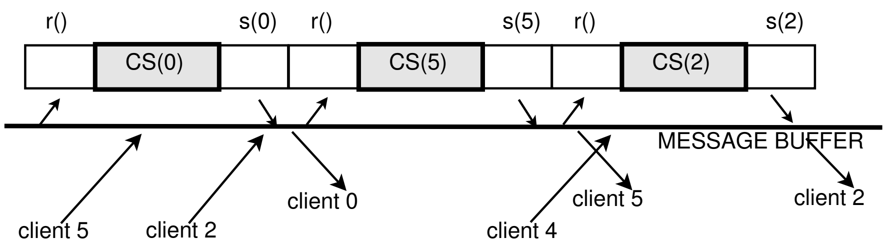


Figure 6.2 – Mutual exclusion server – message-passing implementation; $r()$ – receive message; $s(t)$ – send message to thread t; request from client 0 is already available in the server's message queue

to be executed in mutual exclusion. In shared-memory combining algorithms [OTY99, HIST10, FK12], these two tasks are handled by a single shared object: a list of requests. To execute an operation, a thread adds a request to the list. The current combiner traverses the list to fetch and execute requests. When the current combiner wants to return, it hands over the combiner role to the thread owning the next request in the list (if there are no requests to be executed, the next thread that inserts a request will become the combiner).

HybComb uses hardware message passing for synchronization between the combiner and the other threads. As long as the combiner does not change, synchronization works as with mp-server (Figure 6.2). Still, we use shared memory for managing combiner identity. In a nutshell, HybComb works as follows: When a thread t wants to execute a request, it first checks the identity of the combiner through a shared variable. If a combiner is available and ready to handle the request, t sends a message to that combiner. If not, t tries to promote itself to a combiner, by executing CAS on the variable that keeps the combiner identity.

Managing combiner identity using message passing would be complex and probably inefficient. The main problem is that a thread acting as a combiner has to stop combining at some point, which must be synchronized with actions of other threads. To get its operation executed by a combiner, a thread has to get the identity of the combiner thread and send a request to it. If the combiner identity changes in the meantime, the operation will never get executed. Dealing with this problem using message-passing would require either a delegated thread (which is exactly what the combiner approach is trying to avoid), or intensive communication between threads (*e.g.*, broadcast).

Detailed description Algorithm 4 describes HybComb. The interface is the same as that of CC-Synch: When a thread wants to execute a critical section, it calls the *apply_op* method, providing the corresponding HybComb instance, a pointer to the function to execute and its arguments. Note, however, that HybComb is not just a simple adaptation of existing combining algorithms, where message passing is used instead of a shared list to make the combiner thread aware of the requests to execute. As already mentioned, using message passing requires us to be able to identify the combiner thread to which requests should be sent. This should be carefully handled, especially at the time the combiner changes. This problem does not exist in combining techniques fully based on shared memory since it is the combiner thread that fetches requests from a shared data structure.

The code executed by the active combiner are lines 26-46. Algorithm 4 ensures that these lines are executed in mutual exclusion, *i.e.*, that there is a single active combiner at a time. To manage combiner identity, a data structure called $Node$ is used. Each thread owns a reference to a different node (my_node). The id of the thread owning a node is saved in the field $Node.thread\_id$. Managing combiner identity is done using the shared pointer $last\_registered\_combiner$. To become a combiner (lines 20-24), a thread t tries to execute a CAS operation on $last\_registered\_combiner$ to make it point to its node. If the CAS succeeds, t keeps a pointer to the node corresponding to the previous $last\_registered\_combiner$ in its local variable $last\_reg$. This mechanism can be seen as building a logical queue where the head of the queue is the current active combiner and the tail is $last\_registered\_combiner$, each thread in the queue having a reference to the predecessor in its $last\_reg$ variable. The $Node.combining\_done$ flag is used to synchronize the threads in the queue. Before starting executing as a combiner, a thread spins on the $combining\_done$ flag of its predecessor (line 22), which is set by the predecessor when it finishes combining (line 45).

Upon calling $apply\_op$, a thread t first tries to register its request with the last combiner ($last\_registered\_combiner$), by performing a fetch-and-increment on the $Node.n\_ops$ field of the corresponding node (line 14). This field guarantees that one combiner will receive and execute at most MAX_OPS requests of other threads. If the threshold MAX_OPS is not reached, t sends its request to the combiner using message passing (line 16), and waits for a response (line 17). If the last registered combiner cannot accept any new request, t tries to register itself as a combiner as already explained.

Once t becomes the active combiner, it first executes its own request (line 26). Then it reads messages from its message queue, processes requests and sends responses. When its message queue is empty, t decides to stop combining and announces it by writing MAX_OPS to its n_ops field (line 33). Since it does so using SWAP, it retains the old value of n_ops (in $total\_ops$), which is the total number of requests it has to serve as a combiner. It then finishes its combining round by serving the remaining requests, if any (lines 37-40).

Before returning, t must get the node it will use next time it calls $apply\_op$ (we want to avoid allocating a new node for every $apply\_op$ call). Obviously, t cannot use the same node because that requires the $combining\_done$ field to be reset, but t cannot know when the next combiner will have read this field. As a solution, only one additional node is allocated for all n threads, and t gets the node that was used by the previous combiner (pointed by $departed\_combiner$) (lines 42-45)[2]: t knows that the $combining\_done$ field of this node can be reset since t was the thread spinning on this node. Finally, note that t must not reset the

[2]The use of a SWAP operation at line 42 to exchange the two nodes is only for brevity. An atomic operation is not needed since these lines are executed in mutual exclusion anyway.

Algorithm 4 HYBCOMB combining algorithm – code for thread id

```
const $MAX\_OPS$ {* max. operations per combiner *}
type $Node${$thread\_id$: int, $n\_ops$: int, $combining\_done$: bool}
type $HybcombLock${$last\_registered\_combiner$: $Node$ ptr, $departed\_combiner$: $Node$ ptr}

Local Variables:
my_node: $Node$ ptr $\leftarrow$ {$id$, $MAX\_OPS$, false}

init($lock$: $HybcombLock$)
    $new\_node \leftarrow$ {$\bot$, $MAX\_OPS$, true}
    $lock.departed\_combiner \leftarrow new\_node$
    $lock.last\_registered\_combiner \leftarrow new\_node$

apply_op ($lock$: $HybcombLock$, $func\_ptr$, $args$)
    $ops\_completed \leftarrow 0$
    loop
        $last\_reg \leftarrow lock.last\_registered\_combiner$
        {* try to register with last registered combiner *}
        if $FAA(last\_reg.n\_ops, 1) < MAX\_OPS$ then
            {* success. send message to combiner and wait *}
            $send(last\_reg.thread\_id, \{id, func\_ptr, args\})$
            return $receive(1)$
        else
            {* failure: try to register as combiner *}
            if $CAS(lock.last\_registered\_combiner, last\_reg, my\_node)$ then
                $my\_node.n\_ops \leftarrow 0$
                while $\neg last\_reg.combining\_done$ do
                    $nop$
                break

    {* became combiner. do your own op first *}
    $retval \leftarrow func\_ptr(args)$

    {* as long as message queue is not empty, handle requests *}
    while $\neg is\_msg\_queue\_empty()$ do
        $\{sender\_id, fptr, fargs\} \leftarrow receive(3)$
        $send(sender\_id, fptr(fargs))$
        $ops\_completed \leftarrow ops\_completed + 1$

    {* close combining for new requests *}
    $total\_ops \leftarrow SWAP(my\_node.n\_ops, MAX\_OPS)$
    if $total\_ops > MAX\_OPS$ then
        $total\_ops \leftarrow MAX\_OPS$

    {* serve remaining requests *}
    while $ops\_completed < total\_ops$ do
        $\{sender\_id, fptr, fargs\} \leftarrow receive(3)$
        $send(sender\_id, fptr(fargs))$
        $ops\_completed \leftarrow ops\_completed + 1$

    {* exchange your node, inform next combiner and return *}
    $my\_node \leftarrow SWAP(lock.departed\_combiner, my\_node)$
    $my\_node.combining\_done \leftarrow$ false
    $my\_node.thread\_id \leftarrow id$
    $lock.departed\_combiner.combining\_done \leftarrow$ true
    return $retval$
```

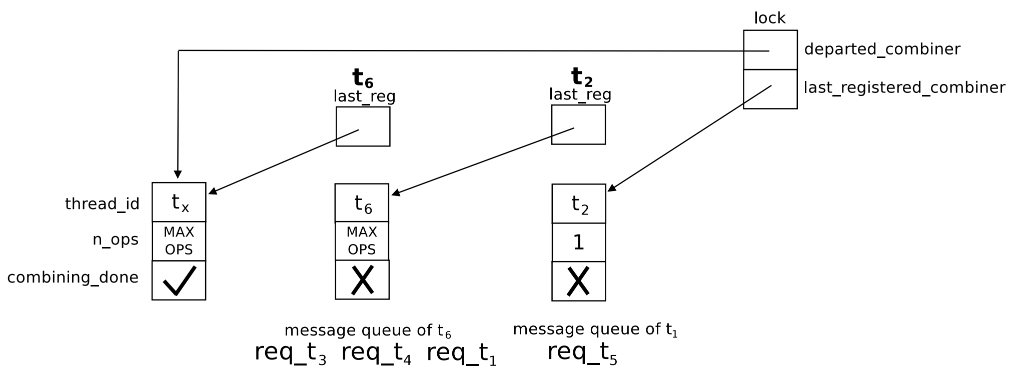


Figure 6.3 – An execution of HYBCOMB ($MAX\_OPS = 3$). The current combiner is t_6, the next one is t_2, polling on t_6's node. The requests of t_1, t_3, t_4, and t_5 will be combined by t_6 and t_2.

n_ops field of its new node at this point because other threads might still have an old reference to this node in their $last\_reg$ variable (lines 12-14): if n_ops were reset, these threads could send requests to t while it is not a combiner. Thus, t will reset n_ops only once it registers as a combiner again (line 21).

Figure 6.3 illustrates an execution of HYBCOMB, assuming $MAX\_OPS = 3$, where threads t_1 - t_6 are simultaneously calling $apply\_op$ to execute their critical sections. Thread t_6 is the current combiner. It executes its own request, and will subsequently execute the requests of t_3, t_4, and t_1, which are waiting in its message queue. Since at most three requests can be executed by a thread on top of its own, the request from t_2 could not be served by t_6. Consequently, t_2 has executed CAS (line 20 of Algorithm 4) and registered as a new combiner: It is now spinning on $n_{t6}.combining\_done$, waiting for t_6 to hand over the combiner role. Thread t_5 will have its request executed by t_1, once the latter starts combining. Note that the nodes of the current and future combiners (t_6 and t_2 in Figure 6.3) form an implicit queue: The head of the queue is the dummy node (pointed to by $lock.departed\_combiner$), whereas the tail is the last registered combiner (pointed to by $lock.last\_registered\_combiner$).

Additional comments Before sketching the proof of correctness, we make a few remarks on the way HYBCOMB works. First, we can note that registering as a combiner (line 20) and resetting the n_ops counter (line 21) are not atomic. This does not affect the correctness of the algorithm. In the very unfortunate event where a thread t' executes the FAA at line 14 while t is between those two lines, t' will simply not manage to register its request with t, and so, will try to become the next combiner. This could merely result in a performance penalty as t would only have its own request to execute as a combiner. Results presented in Section 6.2 show that this rarely occurs in practice.

Note also that the first *while* loop in the request execution part (lines 28 to 31) is not necessary for correctness: The thread can decide to stop combining as soon as it has executed its own request. Still, this loop is beneficial for performance, as postponing the SWAP at line 33 increases the combining potential.

HYBCOMB uses a CAS operation like some other combining algorithms [OTY99, HIST10], but unlike CC-SYNCH [FK12]. It is well known that CAS can impair performance (because it can repeatedly fail, causing contention) as well as fairness (a thread can starve if it executes CAS in a loop and persistently fails). We still choose to use CAS and not SWAP at line 20 for the following two reasons: i) if SWAP is used and several threads try to register as combiners, they all succeed but some of them only have their own request to execute as a combiner, whereas with CAS only one thread manages to register as a combiner, and potentially execute all other requests; ii) the CAS is not expected to be a hot spot in HYBCOMB as it is only executed when a thread wants to register as a combiner. Experiments presented in Section 6.2 confirm the second point. If desired, a middle ground would be to use SWAP only if CAS fails several times.

Correctness proof (sketch). The key idea is to show that Algorithm 4 maintains a queue of $Nodes$, denoted by CS_{queue} (queue for entering the CS corresponding to lines 26 to 46), where each node represents a thread (except for the head). As discussed above and shown in Figure 6.3, the head of the queue is the node pointed to by $lock.departed\_combiner$.

Other nodes in the queue, if any, correspond to current and future combiners, *i.e.*, threads that want to enter the CS. The operation $insert$ into CS_{queue} corresponds to a successful execution of CAS at line 20. The operation $remove$ from CS_{queue} corresponds to the execution of lines 42 to 46. The queue CS_{queue} is represented as follows: The tail of CS_{queue} is the node pointed to by the field $lock.last\_registered\_combiner$ (line 8). The predecessor of node n_t (representing thread t) is the node pointed to by $last\_reg_t$ (line 12). The head of CS_{queue} is node n_t such that $last\_reg_t.combining\_done = true$.

In addition to nodes representing threads, CS_{queue} includes one dummy node, initially new_node (line 6). The dummy node is the only node in CS_{queue} where combining is marked complete ($combining\_done = true$). An empty queue contains only the dummy node and $lock.last\_registered\_combiner$ points to the dummy node.

Note that the field $lock.departed\_combiner$ plays no role with respect to CS_{queue}. It points to the dummy node: whenever some thread t leaves the CS, the node n_t becomes the new dummy node, and the previous dummy node becomes n_t. The reason for this has been explained at the end of the paragraph *Detailed description* above.

Lemma 1. *Algorithm 4 maintains the queue structure just described.*

Proof. The queue structure is modified by a successful execution of CAS (line 20). We prove by induction that the queue structure always holds.

Base step: The empty queue structure holds initially by lines 6, 8.

Induction step: By lines 12 and 20 (CAS), if the queue structure holds before executing CAS, then it is easy to see that the queue structure also holds after successfully executing CAS. We also show that pointer $lock.departed\_combiner$ always points to the dummy node. Initially by line 7 $lock.departed\_combiner$ correctly points to the dummy node. Moreover, $lock.departed\_combiner$ is updated at line 42, pointing to node n_t representing thread t that just left the CS. By line 45 (setting $combining\_done$ to $true$), the node becomes the new dummy node. □

Proposition 1. *Lines 26 to 46 are executed in mutual exclusion (one combiner at a time).*

Proof. A thread t can execute lines 26 to 46 only after a successful execution of CAS (line 20). By Lemma 1, t is correctly introduced in CS_{queue}. By the same lemma, only node $n_{t'}$ at the head of CS_{queue} is such that $last\_reg_{t'}.combining\_done = true$. Thus, by line 22, only one thread can enter the CS. □

It follows from Proposition 1 that Algorithm 4 is safe: Because CS is executed in mutual exclusion, thread operations (pointed to by the $func\_ptr$ argument) are also executed in mutual exclusion. It can be shown that linearizability follows (since an operation of thread t can only be executed between the moments of t entering and leaving $apply\_op$). We now show that liveness also holds, *i.e.*, that Algorithm 4 is *deadlock-free* (if there are threads calling $apply\_op$, some thread eventually executes its operation and returns from $apply\_op$).

Lemma 2. *For all nodes n_t with $n_t.n\_ops < MAX\_OPS$, we have n_t in CS_{queue}.*

Proof. We show that the converse holds. Assume that node n_t is not in CS_{queue}. When n_t is in its initial state, we trivially have $n_t.n\_ops \geq MAX\_OPS$. Consider now t entering the CS and later leaving the CS. To reset $n_t.n\_ops$, a thread executes line 21. This can happen only if the CAS at line 20 succeeds, which means that the thread enters CS_{queue}. Before leaving the CS, by line 33, we trivially have $n_t.n\_ops \geq MAX\_OPS$, which terminates the proof. □

Lemma 3. *At lines 29 and 38, only requests (operations to execute) can be received (not responses).*

Proof. For a contradiction, assume r to be the first response received at line 29 or 38, by some thread t. Response r must have been sent by some thread t' at lines 16, 30 or 39. Response r cannot have been sent at line 16, because only requests are sent at line 16.

So assume that r is sent by some thread t' at line 30 or 39. However, thread t' can only have received a request at line 29 or 38 (since t is the first to have received a response at those lines). Therefore t' has sent a response at line 30 or 39: a contradiction. □

Lemma 4. *If the message queue of thread t contains a request, then n_t is in CS_{queue}.*

Proof. Assume that thread t' sends a request to thread t. By lines 12 and 16, the message is sent to $t = last\_reg_{t'}.thread\_id$. We have to show that n_t is in CS_{queue} when t receives the request from t'. Clearly, n_t is at the tail of CS_{queue} when t' executes line 12. If t' sends its request to t at line 16, then line 14 was successfully executed by t', i.e., $n_t.n\_ops < MAX\_OPS$ before t' executes line 14. By Lemma 2 n_t is in CS_{queue} at this time. By Lemma 3, and since every thread must execute line 14 before sending a request to t, thread t cannot leave the CS before the request from t' is received. Therefore, n_t is still in CS_{queue} when the request from t' arrives in the message queue of t. □

Lemma 5. *At line 17, thread t cannot receive a request (i.e., t can only receive the response to the request sent at line 16).*

Proof. Assume t receives a request at line 17, while n_t is not in CS_{queue}. Therefore the message queue of t contains a request. By Lemma 4, node n_t is in CS_{queue}: A contradiction. □

Lemma 6. *If t sends request r at line 16, it eventually receives a response at line 17.*

Proof. By Lemma 4, t's request is received by some thread t' that is in CS_{queue}, *i.e.*, is or will become the combiner. By line 14, $t'.n\_ops$ is larger or equal to the number of requests sent to t' (can be larger since line 14 is executed before line 16). The local variable $ops\_completed_{t'}$ counts the number of requests executed by t'. By lines 33-35, and because only requests are received at lines 29 or 38 (Lemma 3), $total\_ops_{t'}$ is equal to the number of requests sent to t'. By lines 37 and 40, t' leaves the while loop only when all requests sent to t' have been executed. Therefore t' eventually executes request r and sends the response to t. By Lemma 5, t will not receive a request at line 17. Therefore t receives at line 17 the response sent by t. □

Finally:

Proposition 2 (liveness)**.** *Algorithm 4 ensures deadlock freedom.*

Proof. If thread t wants to execute some operation op, then either t eventually gets the response (Lemma 6), or t tries to enter CS_{queue} (line 20). In the latter case, if t succeeds (executes the CAS successfully), then t eventually executes op (line 26) and leaves CS_{queue} (Lemma 5). If t does not succeed the CAS, $lock.last\_registered\_combiner$ has changed in the meantime, which means that some other thread has successfully executed the CAS and will thus eventually execute its operation. □

Recall that starvation freedom is not guaranteed in Algorithm 4 because of the CAS, but it can be easily ensured if a SWAP is introduced (as discussed in the paragraph *Additional comments* above).

6.2 Evaluation

In this section we implement and thoroughly evaluate the algorithms presented in Sections 5.3 and 6.1. We begin by introducing the used hybrid processor and our experimental setup. Next, we present experiments that evaluate different implementations of a concurrent counter. The analysis is then extended to more complex concurrent objects, namely queues and stacks. Finally, we discuss the generality of our results and their applicability to other platforms.

6.2.1 Platform

We use the Tilera TILE-Gx8036, which integrates 36 cores, works at 1.2 GHz and features complete hardware support for both coherent shared memory and message passing [Til14]. The software part comprises GCC 4.4.6 and version 2.6.40.38-MDE-4.1.0.148119 of Tilera's custom Linux kernel. The memory consistency model is relaxed compared to x86, so a careful use of memory fences is necessary to avoid inconsistency. 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. While exchanging messages, a thread 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 thread executes *receive* on one of the four local queues, the first message from the queue is returned. If there are no messages, the thread blocks. Messages consist of one or multiple words.

6.2.2 Methodology and setup

We have implemented HybLock, mp-server and HybComb on the TILE-Gx, as well as several algorithms purely based on shared memory: the MCS [MCS91] and CLH [Cra93, MLH94] queue locks, a test-and-set (TAS) lock, the CC-Synch combining algorithm [FK12] and shm-server, a server-based approach. shm-server can be seen as a simplified version of RCL [LDT$^+$12], since it implements the same core mechanism (an array of cache lines, one for each client), but lacks support for some advanced features, such as nested critical sections (note that this simplification does not decrease performance). The implementations have been carefully optimized and compiled with the O3 flag. Because of the relaxed memory model of the TILE-Gx, we have inserted memory fences where necessary to ensure correctness. In particular, with classic locks, a fence is always necessary before releasing the lock, in order for the next lock owner to read the most recent data. In the delegation implementations, fences are not necessary after every CS execution, but only on combiner switching. This is because we assume that shared data is accessed only inside CSes, which holds for the concurrent objects we evaluate. A more conservative use of memory fences would be necessary when this is not the case [CCPG13]. To obtain the best possible performance, we augment all of the delegation implementations with a simple interface that allows a thread to send a unique opcode of the CS to the servicing thread, rather than a function pointer. This allows the compiler to inline the function calls that the servicing thread makes for every CS, which results in a visible performance increase in most cases [CCPG13]. It is worth noting that the results are qualitatively the same without this optimization.

We use the methodology commonly found in related studies [FK12, HIST10, MS96, MA13]: In each experiment, a specified number of application threads repeatedly execute operations on a concurrent object. After every operation, a thread executes a random number of empty loop iterations (at most 50). This simulates local work and prevents *long runs*, in which a thread would execute bursts of operations on a concurrent object in its local cache. To minimize interference caused by context switching, we assume a uniprogrammed environment, where each thread runs on a separate core (multiprogramming is discussed in Section 6.3). We pin threads to cores in ascending order, *i.e.*, thread i is pinned to core i. With server-based approaches (SHM-SERVER and MP-SERVER), the server code is executed by thread 0, and other threads execute application code (the server position has a negligible performance impact). In case of classic locks and combining, all threads run the same code. Unless otherwise stated, the maximum number of requests a thread can combine in HYBCOMB and CC-SYNCH is set to 200 (we analyze this choice later in this section). Every value reported in the graphs is an average over ten one-second runs.

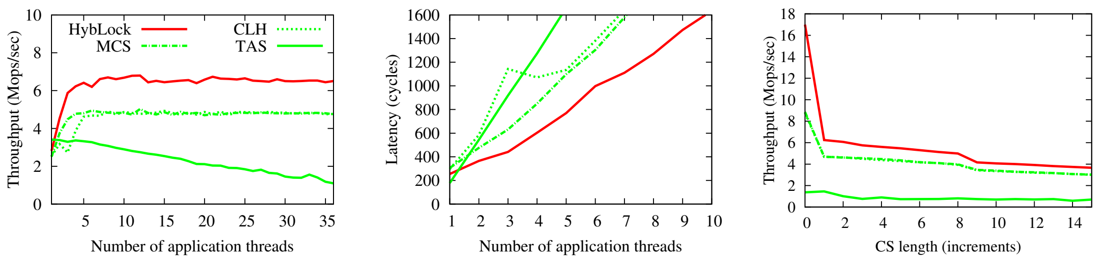
6.2.3 Microbenchmarks

For the sake of clarity, we evaluate classic locks and delegation separately. After discussing the key results related to each of the techniques, we compare them directly to complete the analysis.

Classic Locks

We first use each of the locks to implement a simple concurrent object, a counter. Figure 6.4a gives the counter throughput. HYBLOCK is the best performer, reaching 1.35x higher throughput than the MCS and CHL locks in high concurrency levels. Even in lower degrees of concurrency, HYBLOCK still performs visibly better than the other queue locks. We attribute this improvement to the optimized way of handing over the lock: HYBLOCK uses a message, whereas MCS and CLH use spinning. As expected, TAS scales poorly.

Figure 6.4b shows average operation latencies observed by the threads in the same benchmark. Even with one thread (no concurrency at all), HYBLOCK is more efficient than most alternatives,



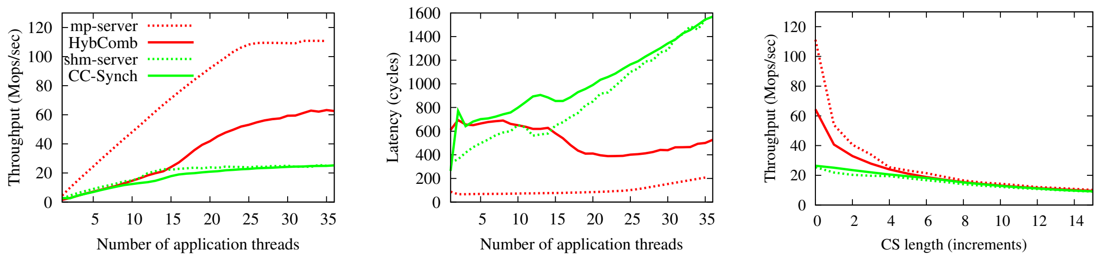
(a) Counter throughput (b) Counter latency (c) Varying CS length (36 threads)

Figure 6.4 – Performance of classic locks

except for TAS. Although message passing is not used in this case, HYBLOCK has the advantage that it does not need any node housekeeping, necessary with MCS and CLH. We can also see that MCS and CLH deliver nearly indistinguishable performance, except with 2-4 threads, where CLH is more robust. We believe this is architecture and implementation-dependent. In any case, the general trend we can observe with all of the classic locks is that the average latency increases rapidly as concurrency grows, even with HYBLOCK as the most efficient option.

We now examine what happens when the CS body is longer. We implement a CS in which the elements of an integer array are incremented. We vary the number of increments in the CS and observe the maximum throughput (with 36 threads) in Figure 6.4c. When the critical section contains no shared-memory accesses (zero increments), we are left with the pure synchronization overhead of the lock/unlock pair. This overhead is about 2x lower with HYBLOCK than with the shared-memory queue locks. As soon as we add shared data manipulation in the critical section, even only one increment, there is a sharp performance decrease, mostly because the shared data bounces between cores, as described in Section 5.3. As the number of increments increases past one, the performance decreases more slowly. This is because one cache miss brings over eight array elements, so only one in eight increments is particularly expensive because of a cache miss. Also, it should be noted that prefetching can hide some of the latency of the subsequent cache misses. Hence, with 25 increments the advantage of HYBLOCK over MCS and CLH is still about 1.18x.

In conclusion, HYBLOCK is visibly more efficient than MCS and CLH, both in low and high concurrency levels. What limits further performance improvement, however, are overheads inherent to classic locks. Those are in the first line cache misses inside the critical section, but the memory fences associated with every critical section play an important role on this platform as well. Next, we will see that more significant performance gains are possible with delegation, since the mentioned inherent overheads do not exist.



(a) Counter throughput (b) Counter latency (c) Varying CS length (36 threads)

Figure 6.5 – Performance of delegation ($MAX\_OPS = 200$ with CC-SYNCH and HYBCOMB)

Delegation

Again, we start by implementing a concurrent counter. Figure 6.5a shows the throughput of the counter implementations. The approaches that use hardware message passing are clearly faster: MP-SERVER is most efficient in all concurrency levels. It reaches 4.3x higher throughput than SHM-SERVER, indicating that message passing supported natively is much more efficient than emulation over shared memory. When it comes to combining, HYBCOMB consistently outperforms CC-SYNCH. This is especially pronounced in higher concurrency levels, where HYBCOMB reaches about 2.5x higher throughput. CC-SYNCH and SHM-SERVER have very similar performance, indicating that CC-SYNCH manages to avoid dedicating cores at virtually no performance cost. On the other hand, the difference between MP-SERVER and HYBCOMB is much more visible. We will shortly identify the source of this difference, and explain how it can be minimized. We can also see a big improvement compared to the throughput of classic locks (Figure 6.4a), which confirms that delegation is more resilient to contention.

Figure 6.5b shows the average request latency observed by application threads. Again, MP-SERVER has by far the lowest latency even in low concurrency levels, indicating that hardware message passing is useful even latency-wise. HYBCOMB has lower latency than CC-SYNCH, which becomes especially visible as concurrency increases. The only noteworthy exception is single-threaded performance, where CC-SYNCH is better than HYBCOMB. We believe this is mainly because an isolated thread running CC-SYNCH executes only one atomic instruction per operation, whereas HYBCOMB executes three. Since atomic instructions on the TILE-Gx are not executed in the local L1 cache, but in the L2 cache of the core that is home to the corresponding memory word (most likely a distant core), this results in a higher latency. As concurrency increases, the latency of both CC-SYNCH and HYBCOMB dips at one point before continuing to grow (between 14 and 17, resp. 14 and 24 application threads). This is due to more intensive combining, as we will confirm shortly.

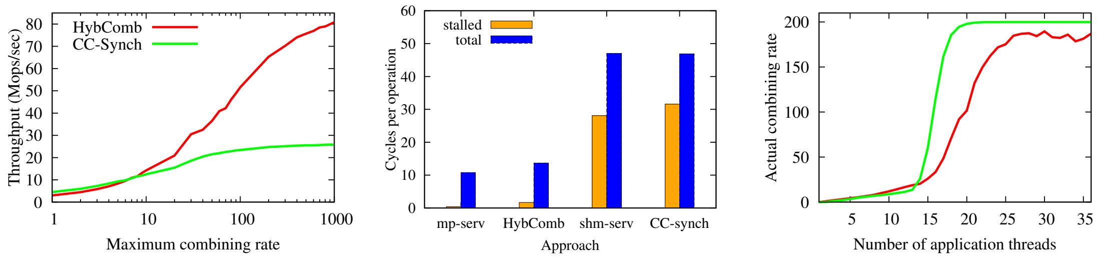
As with classic locks, we continue by replacing one counter by an array of 64 counters, incremented in a loop with a varying number of iterations. Figure 6.5c presents the results: MP-SERVER and HYBCOMB can lead to better performance mainly when CSes are short. This is mainly because synchronization is very cheap with delegation, so the time to execute a slightly longer CS body already dominates synchronization overheads. At 15 loop iterations, the difference between the best (MP-SERVER) and the worst (SHM-SERVER) performer drops to about 10%, since the time to execute the CS body (which is the same with all of the implementations, if we ignore combiner switching) dominates the entry/exit overhead.

One might question the choice of the maximum allowed combining rate (MAX_OPS). If MAX_OPS is too low, less combining is possible, which negatively affects throughput. On the other hand, increasing it above a certain limit does not increase throughput further, as the cost of combiner switching becomes negligible, but can result in higher latency observed by the combining thread. The optimal value heavily depends on the application needs and anticipated concurrency level. In Figure 6.6a, we examine how the maximum achievable counter throughput changes with MAX_OPS. Very high MAX_OPS values provide little benefit in terms of throughput of CC-SYNCH. On the other hand, as we increase MAX_OPS up to 1,000, the throughput of HYBCOMB continues to grow, barely showing signs of saturation. Combining is so fast with HYBCOMB, that the impact of combiner switching is visible even when MAX_OPS is high. This explains the difference between MP-SERVER and HYBCOMB observed in Figure 6.5a (recall that MAX_OPS is set to 200 there). The throughput of HYBCOMB levels off at about 88 Mops/sec, with MAX_OPS set to 5'000. Therefore, one can achieve nearly as high throughput with HYBCOMB as with MP-SERVER, if willing to trade the throughput increase for sporadic latency "hiccups" for some requests (when the requesting thread becomes a combiner). We have chosen a moderate value of 200 for our experiments, since it already provides the highest possible throughput with CC-SYNCH and decent results with HYBCOMB.

Now we more precisely identify the reason for the observed performance improvement with HYBCOMB and MP-SERVER in the counter benchmark. Figure 6.6b shows the average number of CPU stalls per operation on the servicing thread under maximum load, as well as the total number of cycles per operation.[3] The advantage of HYBCOMB and MP-SERVER becomes clearer: The servicing thread is virtually never stalled, whereas CPU stalls account for more than 50% of the cycles of the servicing thread in CC-SYNCH and SHM-SERVER. There are no event counters that would provide more fine-grained information on the source of stalls, but we believe they mostly originate from the load-store unit, which has to wait for the cache coherence protocol to fetch data. This confirms the reasoning from Section 5.3: Cache-coherence related stalls are an important source of overhead, and hardware message passing is helpful in avoiding them.

Figure 6.6c shows the average combining rate with HYBCOMB and CC-SYNCH. Ideally, we expect it to reach MAX_OPS under high load. At the beginning, the actual combining rate steadily grows, and is approximately equal to the number of threads minus one. This is because a combiner manages to combine one request for all of the other threads. At that point, no thread has started the subsequent operation yet, so the combiner returns. As concurrency

[3]To be able to use per-core event counters, only in this experiment we modified HYBCOMB and CC-SYNCH to have a fixed combiner for the whole run, which is equivalent to setting $MAX\_OPS = \infty$.



(a) Impact of the combining rate (b) CPU Stalls (c) Combining rate

Figure 6.6 – Analyzing the performance of the different synchronization techniques

grows, more requests arrive at the combiner concurrently. As it takes more time to service them, there is more time for other requests to arrive before the combiner returns, and so forth. This circular effect leads to a sudden sharp increase in the combining rate, which explains the latency dip we observed in Figure 6.5b. As we can see in Figure 6.6c, in high concurrency levels CC-SYNCH reaches the desired combining rate, whereas HYBCOMB is slightly below it. This is because registering as a combiner and resetting the n_ops field are not atomic. As explained in Section 6.1.3, an unfortunate thread interleaving could leave one combiner with no work to do because a new thread would register as a combiner before any request is associated with the current one. However, we can see that this has only a marginal effect on the combining rate in practice: In spite of somewhat lower combining rate, HYBCOMB still has much better performance than CC-SYNCH (Figure 6.5).

Finally, recall that HYBCOMB uses CAS, but the presented graphs indicate that this does not cause visible performance degradation. This is because, when concurrency is high, threads rarely execute CAS: They mostly send their requests to an active combiner. Indeed, we have measured as few as 0.1 executed CAS per operation (call to $apply\_op$) in high concurrency levels. This number is a bit higher when concurrency is not high enough to trigger high combining rates, but even then, there are not more than 0.7 CAS per operation in multithreaded executions. Regarding fairness, we have measured the ratio between the highest and lowest number of operations executed by some thread (so 1 denotes ideal fairness). Across the whole concurrency spectrum, the highest value of this ratio with HYBCOMB is 1.2 and the average is 1.16. Even MP-SERVER, in which all requests are read from a hardware FIFO queue, has a ratio of nearly 1.1, only because some cores are nearer to the server, so they execute slightly more operations. Hence, the use of CAS in HYBCOMB does not impair fairness on this platform.

Locking vs. Delegation

In Section 5.3, we pointed out key differences between classic locks and delegation. Recall that, from a pure performance perspective, delegation is more resilient to contention, whereas classic locks are expected to achieve better low-concurrency performance, which we now verify.

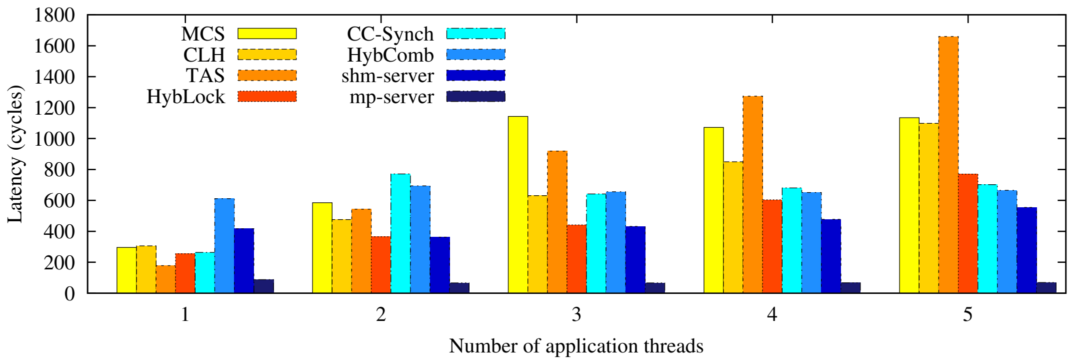


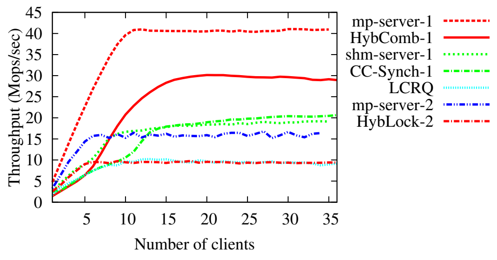
Figure 6.7 – Classic locks vs. delegation: latency comparison

Figure 6.7 gives the latency of a concurrent counter implemented using the different locks and delegation approaches: It is a subset of latency data from Figures 6.4b and 6.5b, represented as a bar chart. The results confirm that delegation is more resilient to contention: With five threads, all delegation implementations outperform the locks. But even in lower concurrency, the locks turn out not to be superior in all cases. In particular, perhaps surprisingly, MP-SERVER is by far the best performer, even with a single application thread (i.e., no contention at all): It even outperforms a simple TAS lock in this case. This might look surprising, as MP-SERVER includes communication with the server, which is avoided in classic locks and combining. Recall, however, that even a TAS lock requires an atomic operation (which is executed in a remote L2 cache on the TILE-Gx), and memory fences (to make sure that the next lock owner's view of protected data stays consistent). This turns out to be more costly than contacting the server using hardware message passing.

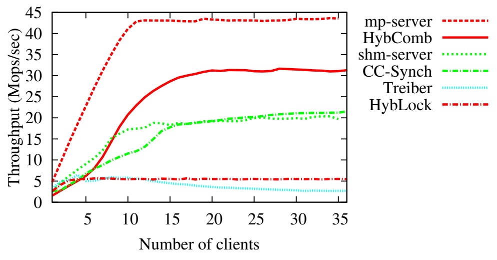
According to the presented data, hardware support for message passing might justify server-based CS execution even for uncontended locks. This, however, depends on the particular scenario, because of the already discussed downsides of this approach (server dedication, false serialization, the need to appropriately encapsulate critical sections, etc.). Note that a detailed comparison of classic locks and delegation is out of the scope of this study: The purpose of this subsection is merely to present global trends and to point out that hardware message passing support changes the landscape of synchronization performance.

6.2.4 Queues and stacks

Because of their ubiquity, concurrent linearizable queues and stacks are typically used to evaluate the performance of universal synchronization constructions [HIST10, FK12, FK11]. Following this observation, we implement some well-established queues and stacks from the literature and analyze their performance. With these experiments, we study an important use case where CSes are usually short. The implementations store 64-bit values, and are evaluated under balanced load. For brevity, we focus only on throughput analysis. The latencies show trends similar to those presented in Section 6.2.3. As delegation is able to achieve much higher



(a) **Queue.** X-1 – one-lock MS-Queue implemented using approach X; X-2 – two-lock MS-Queue, implemented using approach X; LCRQ – nonblocking queue (see text)



(b) **Stack.** X – coarse-lock stack implemented using approach X; Treiber – nonblocking stack (see text)

Figure 6.8 – Performance of concurrent queues and stacks under balanced load

throughput than classic locks, we leave the latter out from the plots to avoid clutter, except for HYBLOCK, as the best performer in that category.

Queues One of the best-known blocking queues is the fine-grained Michael and Scott queue (MS-Queue) [MS96]. It is based on a linked list accessed using two CSes, so enqueues and dequeues can take place in parallel. Its performance mostly depends on the way CSes are implemented. We implement MS-Queue using HYBCOMB, CC-SYNCH, the two server-based approaches (which requires two dedicated servers per queue instance), and HYBLOCK. Besides the two-lock version, we implement the same queue using a single lock. We also test LCRQ [MA13], a nonblocking queue that takes advantage of the wide spectrum of atomic operations supported by x86 processors. The TILE-Gx supports most of the necessary instructions, so adapting the LCRQ code written in C for x86 was relatively easy.[4]

The queue performance is shown in Figure 6.8a. The single-lock MS-Queues ("-1" suffix in the legend) perform best. Among them, MP-SERVER and HYBCOMB are most efficient: They obtain respectively up to 2x and 1.5x higher throughput than the third best implementation. LCRQ, as well as the two-lock versions of MS-Queue[5], level off sooner than the rest, which we now explain in more detail.

One might expect fine-grained locking to always outperform a coarse lock. However, fine-grained locking involves a tradeoff, since the additional synchronization it includes might outweigh the gain that comes from increasing parallelism [HIST10]. Given Tilera's relaxed memory model, the enqueue and dequeue methods of the two-lock queue must be carefully coded if they can run in parallel – memory fences are necessary to ensure queue consistency.

[4]We made the following modifications: the lacking bitwise test-and-set (BTAS) was replaced with a simple CAS loop; for lack of the 128-bit CAS (CAS2), we modified LCRQ to store 32-bit values, and used a 64-bit CAS.

[5]To avoid clutter, we omit HYBCOMB-2, SHM-SERVER-2, and CC-SYNCH-2 from the graph, as they are outperformed by MP-SERVER-2.

When delegation is used, it turns out that the necessity of inserting fences far outweighs the benefit from fine-grained access. Therefore, a simple sequential queue implemented using MP-SERVER or HYBCOMB yields best results. On the other hand, fine-grained synchronization pays off when classic locks are used, as HYBLOCK-2 yields an almost twofold performance increase over HYBLOCK-1 (not shown). This is because synchronization is much more expensive in this case, and halving its cost by moving to two locks is not canceled out by the additional fences.

In spite of its excellent performance on x86 [MA13], LCRQ is less efficient on the TILE-Gx, and achieves performance similar to that of the HYBLOCK queue. We speculate this is primarily because of the way atomic instructions work on this processor. Namely, L2 caches are in charge of executing them. This means that two atomic instructions might collide on an L2 cache even if they have independent data sets, leading to frequent *false serialization.* A better performance might be achievable by optimizing LCRQ with the cache hierarchy of the TILE-Gx in mind, but this falls outside the scope of this study.

Stacks The stack is known to be hard to parallelize, since both push and pop operations access its top. One way to obviate its seemingly inherent sequential nature is to use the *elimination* technique [ST95, CGH13]: if a push and pop operation are executed concurrently, they can be *eliminated* to avoid accessing the stack. Still, if an operation cannot be eliminated, it has to access the top of the stack. As elimination is orthogonal to the content presented here, we evaluate the performance of a non-elimination concurrent stack (which, of course, can be used to back up an elimination-based stack).

We evaluate six implementations: a sequential linked-list based stack, turned concurrent using MP-SERVER, HYBCOMB, CC-SYNCH, SHM-SERVER, and HYBLOCK, as well as well-known Treiber's nonblocking stack [Tre86]. Their performance is given in Figure 6.8b. MP-SERVER and HYBCOMB stacks are again the best performers – and the numbers nearly match those given in Figure 6.8a for the single-lock MS queue. This is not surprising, as both concurrent objects are represented as linked lists protected by a coarse lock. Treiber stack performance is inferior to that of the other implementations, because the head of the stack is accessed using CAS. This causes growing contention as concurrency increases, as most CAS operations repeatedly fail.

6.2.5 Observations

One might wonder to what extent our results are processor-specific. To answer this question, we have measured the throughput of a concurrent counter implemented using MCS, CC-SYNCH and SHM-SERVER on two single-socket x86 processors: a 10-core Intel Xeon E7-L8867 (without and with Hyperthreading enabled), and a 6-core AMD Opteron 6176. In virtually all of the cases, peak throughput is significantly lower on x86: Most of the results are presented in Chapter 7. For delegation, we have also measured the number of stalls per operation of the servicing thread (as in Figure 6.6b) and got proportionally larger numbers than on the TILE-Gx. Therefore, we believe HYBLOCK, MP-SERVER and HYBCOMB would outperform their shared-memory-only counterparts also on x86 hardware, if it provided native message passing support. Moreover, since there are more stalls on x86, the potential performance improvement is higher.

Still, it is noteworthy that we did observe some platform-specific effects. Since the implementation of atomic instructions differs on the TILE-Gx and the x86, algorithms that use them intensively (typically nonblocking ones) may behave differently. This is visible on the example of LCRQ, which has substantially higher throughput on the x86 processors than on the TILE-Gx. Also, because of the different memory consistency model, two-lock MS-Queue outperforms its one-lock counterpart on the Xeon and Opteron (cf. Figure 6.8a), in contrast to what we have observed on the TILE-Gx. Note, however, that these differences are specific to implementations of a certain concurrent object, a queue in this case. In other words, Figure 6.8a (showing queue performance) would look different on an x86, but the qualitative advantage of HybLock, mp-server and HybComb over the shared-memory constructions for mutual exclusion, which is central to this study, would in all likelihood remain the same.

Finally, the advantage provided by mp-server and HybComb is due to the way hardware message passing is implemented, and more specifically, to the fact that receive operations read from a local buffer, and that send operations are asynchronous. These features are not too specific, and so, we believe they can be easily provided by future implementations of hardware message passing. Note also that HybComb depends a lot on the performance of the fetch-and-add instruction, since every client must execute it on the same variable before sending a request to the current combiner. Fetch-and-add on x86 processors is typically fast, as it is guaranteed to succeed [MA13]. It should be noted, however, that x86 currently implements fetch-and-add (and other atomic instructions) in the L1 cache, so the latency of moving the cache line from one core to another would contribute to the critical path of HybComb in a hypothetical x86 implementation.

6.3 Additional Considerations

This section discusses some practical issues that arise when message passing is used.

Oversubscribing and thread migration The results presented in Section 6.2 assume a uniprogrammed environment, with at most one thread pinned to a core. This is not an inherent limitation of the hardware message passing approaches.

Indeed, on the TILE-Gx, oversubscribing is easily achieved thanks to the possibility to multiplex the hardware queue of each core (cf. Section 6.2.1), which means that up to four threads can share a core and still have their exclusive message queue. With HybLock, mp-server, and HybComb, application threads can freely migrate to another core in between requests, as long as they are able to reserve a hardware queue on that core. Upon making a request, a thread t is only expected to have a valid identifier, corresponding to its current core and hardware queue.

As long as t remains pinned to the current core while its request is pending, other threads will be able to reach it using that identifier.

More generally speaking, any constraints related to thread migration and oversubscribing can be easily solved with relatively straightforward support at the hardware and OS level. For instance, the *Asynchronous Direct Messages* (ADM) mechanism [SYK10] resembles Tilera's message-passing hardware in many ways, but in addition includes a small associative memory that caches (thread ID, core) pairs. This enables threads to migrate freely, while the OS keeps track of thread-to-core mappings.

Deadlocks Bearing in mind the limited capacity of the hardware message queues, another practical issue with message passing is the possibility of deadlocks, if messages back up in the network and block the sender. Obviously, there are no such problems with HYBLOCK, as a core's message queue contains at most two messages at any point during algorithm execution. Also, the message queues of MP-SERVER clients or HYBCOMB non-combiner threads cannot overflow since they contain at most one message. Therefore, the servicing thread never blocks when sending a response to a request.

In our experiments, the message queue of a servicing thread cannot overflow, as it contains at most 35 3-word requests at any time, which fits in the message queue. More generally, overflows can happen if the hardware queue is not big enough to keep one request per application thread. In this case, some clients could be blocked when sending a request, but this is not an issue since every such *send* is anyway immediately followed by a blocking *receive*.

6.4 Summary

In this chapter we studied how hardware message passing can be used for efficient critical section execution. We proposed three generic constructions tailored to take advantage of hardware message passing: HYBLOCK, a hybrid lock, MP-SERVER, a server-based approach, and HYBCOMB, a hybrid combining algorithm. Experiments on Tilera's TILE-Gx processor show that HYBLOCK, MP-SERVER, and HYBCOMB largely outperform their shared-memory-only counterparts, when used to implement ubiquitous linearizable concurrent objects (counters, queues, stacks).

Our results show that hardware message passing can provide more efficient thread synchronization, and thus, improve the scalability of concurrent code. The hybrid design of HYBLOCK and HYBCOMB demonstrates that processors providing both CC shared memory and message passing are appealing, as they allow us to take the best of both worlds. However, it also illustrates that significant algorithmic effort can be necessary in order to exploit the resources of a hybrid machine.

7 Optimizing Delegation on Processors without Message Passing

Here we study how to optimize delegation performance on prevalent x86 processors, which currently do not feature message-passing hardware. Optimizations are proposed in Section 7.1 and evaluated in Section 7.2. The results are discussed in Section 7.3, before a summary in Section 7.4.

7.1 Optimizing Delegation over CC Shared Memory

In this section, we first describe the server-based delegation algorithm that is used as a starting point for our work. Then, we explain its main bottleneck and detail how it can be optimized for execution over CC shared memory by taking into account characteristics of modern processors.

7.1.1 Baseline algorithm

In the description of the baseline algorithm, we make the following assumptions: (a) Participating threads are known in advance; (b) Data exchanged between clients and servers can fit into one cache line. Assumption (a) allows us to pre-allocate per-client buffers and thus eliminate the cost of synchronization on a shared buffer[1]. Since work to delegate is usually encapsulated inside a function, considering the typical case of 64-byte cache lines, one cache line can store a function pointer, a flag, and several arguments, which justifies assumption (b).

The code is given in Algorithm 5. It uses an array of cache-line-sized slots, one per client thread. Every slot contains a flag with two possible values, $REQUEST$ and $RESPONSE$. To make a request, a client writes the function pointer and arguments in the corresponding slot, and then sets the flag. The server repeatedly scans the client slots, and if there is a request, it is immediately executed, and a response is sent to the client by writing it in the slot and appropriately setting the flag. Algorithm 5 is essentially a stripped-down version

[1]If participating threads are not known in advance, one solution is to assign communication buffers to cores, and to use a simple locking algorithm to arbitrate between threads that would execute on the same core.

Algorithm 5 Baseline delegation algorithm

```
1: CacheLine $channel[0..n-1]$    {$channel[i]$: communication channel between client $i$ and the server}
2: $channel[0..n-1].flag \leftarrow RESPONSE$
3: $channel[0..n-1].msg \leftarrow NULL$

   {Server code}
4: function run_server()
5:    $client\_id \leftarrow 0$
6:    while true do
7:       if $channel[client\_id].flag = REQUEST$ then
8:          $\{func, args\} \leftarrow channel[client\_id].msg$
9:          $channel[client\_id].msg \leftarrow func(args)$
10:         $channel[client\_id].flag \leftarrow RESPONSE$
11:      $client\_id \leftarrow (client\_id + 1) \bmod n$

   {Code of client $i$}
12: function delegate($func\_ptr$, $args$)
13:    $channel[i].msg \leftarrow \{func\_ptr, args\}$
14:    $channel[i].flag \leftarrow REQUEST$
15:    while $channel[i].flag \neq RESPONSE$ do
16:       $nop$
17:    return $channel[i].msg$
```

of RCL [LDT+12]. RCL is more complex because it provides additional features that are not central to this work such as support for nested critical sections or the possibility to have one server managing CSes on several shared objects.

7.1.2 Opportunities for optimization

The server in Algorithm 5 experiences at least two RMRs for each client request. This is illustrated in Figure 7.1, which is essentially a more detailed representation of Figure 5.2. To recapitulate, accesses to shared cache line `channel` is shown, during the communication between a client and the server for the execution of one CS. Figure 7.1 also shows the cache line status during the execution: state `M` (*modifed*) corresponds to read-write mode; state `S` (*shared*) corresponds to read-only mode. When the client wants to execute a CS, it writes the request `channel`, and then keeps spinning. The server reads the request, which results in an RMR, since the last access to `channel` was from the client. The CS is then executed by the server. Afterwards, the server writes to `channel` to respond to the client, triggering another RMR, since the client's copy of `channel` needs to be invalidated.

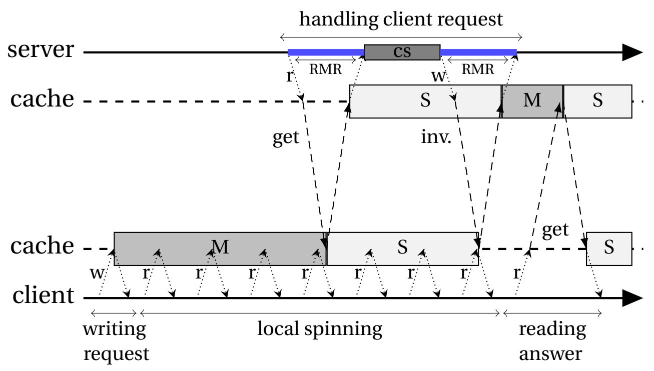


Figure 7.1 – Communication between the server and a client – Baseline algorithm (2 RMRs on the server per request).

7.1.3 Proposed optimizations

By carefully analyzing hardware-level details of executing the presented algorithm on a typical multisocket multicore processor, we identified two optimizations that can considerably improve its performance: backoff in local-spin loops and streaming (non-temporal) stores. We now discuss each of them in more detail.

Backoff with local spinning

Contemporary processors usually have automatic prefetchers, which detect regular data access patterns and proactively bring data closer to the processor core before it is referenced, thus hiding access latency. In Algorithm 5, the server repeatedly iterates over consecutive cache lines, which results in a very regular cache line access pattern, likely to trigger the prefetcher. In our case, prefetching a cache line in read-only mode could hide the latency of reading the client's request, but an RMR would still be generated to upgrade the cache line to read-write mode, at the time the server writes to the channel. Prefetchers are actually able to detect write-access patterns and bring the cache line to the server cache directly in read-write mode. However, the cache line will get downgraded to read-only mode immediately as illustrated by Figure 7.2, since the client is spinning on that cache line, waiting for a response. Therefore, even local spinning, usually considered to be the first condition for a concurrent algorithm's scalability [MCS91], can be detrimental to performance, since it hinders the automatic prefetcher. We will refer to this problem as the *spinning-prefetching collision*.

A way to avoid this collision is to introduce a well-tuned backoff in the client's spin loop. Instead of constantly checking the flag in a loop, the client introduces a fixed waiting time between consecutive checks. Ideally, the backoff should be such that there is only one check, right after the server has written the response, as shown in Figure 7.3. If the waiting time is too short, the spinning might conflict with the prefetcher; If it is too long, the client will unnecessarily keep waiting even though the response is already available. The right value

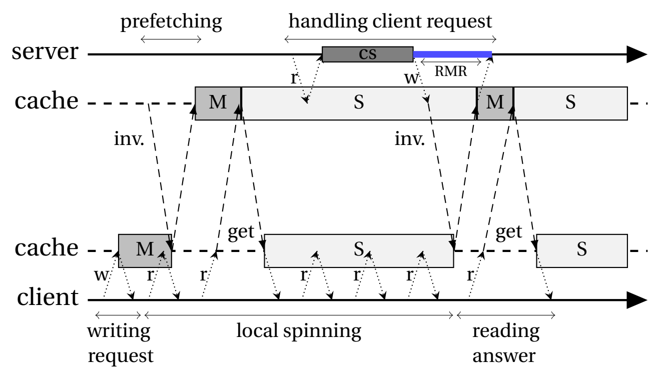


Figure 7.2 – Communication between the server and a client – Spinning-prefetching collision (one RMR on the server per request).

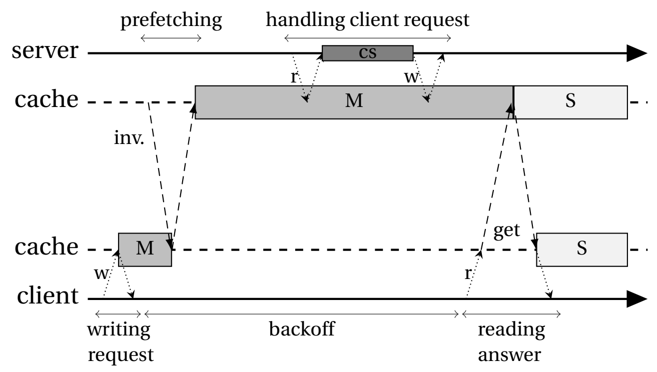


Figure 7.3 – Communication between the server and a client – Backoff with local spinning (no RMRs on the server).

depends on many factors, such as the current load, the way prefetching works etc. Here we tune the backoff manually, i.e., we measure performance with different fixed backoff values, but it would be interesting to study how the waiting time can be re-calculated and updated at runtime.

It is important to stress that, although backoff is a well-known technique in concurrent programming, it is most often used to deal with a completely different problem. Namely backoff is usually used to reduce contention on a shared variable that is concurently accessed by an arbitrary number of threads [Her93, MCS91]. In our case shared variables are not contended since only one client and the server can access the same cache line concurently, but introducing backoff in the spin loop of the client allows avoiding interference with prefetching on the server side.

Another way to prevent the spinning-prefetching collision from happening would be to use the MONITOR/MWAIT instructions, supported by x86 processors. With these instructions, a thread can switch to a low-power state and get notified when a memory location changes, instead of spinning on it. Although this is conceived as an energy-saving feature, it might also have visible performance benefits in our case, since spinning is avoided. However, the

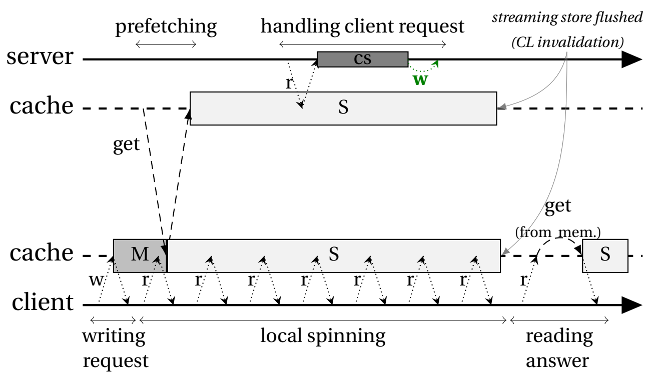


Figure 7.4 – Communication between the server and a client – Streaming stores (no RMRs on the server, streaming stores in green).

MONITOR/MWAIT instuction pair is available only in kernel mode on the processors we could get access to. This is not a problem per se, because MONITOR/MWAIT can be exposed to userspace applications via a special piece of kernel code – a loadable kernel module in case of Linux. Even though we have written such a kernel module, as a simple character device, it turned out to be of little use, because the kernel itself becomes the bottleneck in contended scenarios. This is so even if a separate kernel module is used for every core, and we used a very recent kernel (Ubuntu's Linux 3.2.0-64-generic from June 2014). We speculate that this is due to concurrent access to the kernel data structures for managing character devices. Still, the possibility of introducing userspace access to MONITOR/MWAIT is left open [Int14], which would make it an interesting alternative to study.

Streaming stores

To make the implementation of Algorithm 5 more efficient, we explore an alternative store instruction, a *streaming store*, also referred to as *non-temporal store*. Streaming stores differ from ordinary stores in two aspects: (a) They are weakly ordered and (b) they do not bring the data to the core's cache for writing, but write directly to memory instead. Algorithm execution using streaming stores is illustrated in Figure 7.4. Hence, (a) allows a server store operation to be asynchronously completed and to be overlapped with subsequent requests' handling. Note also that (b) implies that the spinning-prefetching collision described in the previous subsection is not a concern in this case as the prefetcher will try to fetch cache lines in read-only mode (since the server does not issue read-write access requests anymore).

The server's stores still cannot become visible to other cores in a fully arbitrary order: Program order needs to be preserved between stores belonging to the same operation (i.e., the flag must not be written before the actual data). An obvious way to ensure this is to put a memory fence between writing the data and the flag, but such a fence at the server side would force the write buffers to be flushed, incurring overhead that defeats the purpose of using streaming stores. To avoid this, one can take advantage of the fact that the server only sends a function's return value (if there is one) back to the client, so the data and the one-bit flag can fit a variable that can be atomically read and written, thus ensuring that the flag is never updated before the data. There might be other platform-specific ways to ensure this.

In spite of potential performance benefits, it should be noted that streaming stores cannot be applied in all cases because of their weak ordering semantics. Namely, a server implemented with streaming stores can only be used if the data accessed by the server is never accessed by any other thread. This is the case when the server is used to replace a coarse-grained lock on a concurrent object (since threads do not access the object outside the lock). Also, the constraint is satisfied by algorithms based on fine-grained locking, as long as the different locks protect disjoint data sets (*e.g.*, hash tables). However, if data sets are not disjoint, the streaming store that acknowledges request handling can become visible to other cores before the stores that changed the object. An example is the Michael and Scott blocking queue algorithm [MS96] that we adapt in Section 7.2 to use delegation. The original algorithm uses two locks, one for enqueue the other for dequeue operations, that we replace by two servers. Streaming stores cannot be used in this case since data enqueued by one server are eventually dequeued by the second one, breaking the above constraint. Of course, falling back to one lock ensures correctness.

7.2 Evaluation

The goal of this section is twofold: to examine the effectiveness of the proposed optimizations when delegation is implemented on real-world processors (Section 7.2.2) and to compare the performance of optimized delegation with that of most relevant related approaches (Section 7.2.3). Before presenting experimental results, we describe our setup.

7.2.1 Experimental setup

We use two x86 machines throughout this section: a Supermicro SuperServer 5086B-TRF consisting of eight 10-core Intel Xeon Westmere E7-L8867 (2.13 GHz) chips with 2-way SMT (Hyperthreading), *i.e.* 160 hardware threads in total, and an IBM x3755-M3 with four 12-core AMD Opteron Magny-Cours 6176 (2.3 GHz) packages without SMT, for a total of 48 hardware threads. The Xeon runs Red Hat Enterprise Linux Server 6.4 with Linux 2.6.32-358.6.2.el6.x86_64, and the Opteron runs SUSE Linux Enterprise Server 11 with Linux 2.6.32.46-0.3-default. All of the implementations are written in C, carefully optimized and compiled with the O3 flag (maximum optimization level) using GCC 4.4.7 (resp. 4.7.2) on the Xeon (resp. Opteron).

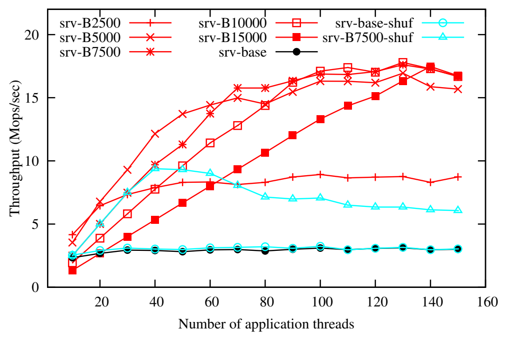
Besides the optimized server-based solutions that implement Algorithm 5, we also evaluate CC-Synch [FK12], as a representative of combining approaches, as well as H-Synch, its NUMA-aware version. H-Synch follows the general idea of grouping operations originating from the same node and executing them together in batches, thus incurring fewer cross-socket cache line transfers and significantly increasing throughput. Even though the Opteron is a multisocket NUMA platform, we do not present H-Synch results on it, since its internal characteristics incur cross-socket communication even if only cores from one socket are involved, thus making typical NUMA-aware strategies unsuccessful [DGT13]. Our experiments have confirmed this. Note also that H-Synch was not evaluated in Chapter 6 because the TILE-Gx is not a NUMA machine.

To evaluate the performance of an algorithm, we use it to implement a concurrent object and stress-test it using a varying number of threads, as we did in Chapter 6. Each thread repeatedly executes operations on the concurrent object, with a short pause of random duration (up to 1000 CPU cycles) between two consecutive requests. We increase the number of clients and measure aggregate throughput, *i.e.* the total number of executed operations by all threads in a unit of time. Every point in the graphs is an average over 10 one-second runs. To avoid OS scheduler interference, we explicitly pin threads to respective cores and run at most one thread on each core. When increasing the number of clients, we pin them to cores from different sockets in a round robin fashion, in order to uniformly distribute threads across the sockets[2]. In server-based implementations, the server is pinned to hardware thread 0. If two servers are used, the second server is pinned to thread 1. On the Xeon, whenever a server thread is used, we do not pin any thread to the other hardware thread that belongs to the same physical core as the thread running the server. This is to avoid undesirable interference with the server, which can impact performance and thus render result analysis significantly more difficult. Note that this is unnecessary on the Opteron since it does not have SMT support.

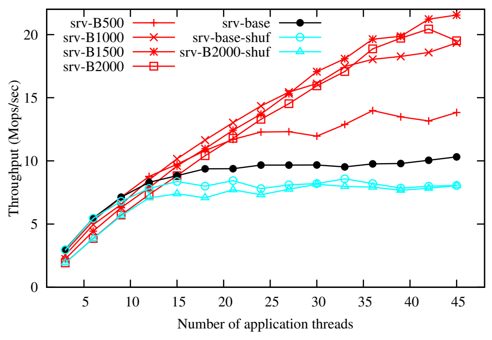
Unless otherwise stated, the client-server communication slots in implementations of Algorithm 5 on the Opteron are allocated as a contiguous array of cache lines, to maximize automatic prefetching. The slots are homed at the server's socket. On the Xeon, instead of using consecutive cache lines, every second cache line is used. We do so because of the *adjacent line prefetcher*, which on every cache miss prefetches the first neighbouring cache line, thus making cache lines always move in pairs [Int14]. This turned out to result in unfavorable interference in our experiments, which we avoid by skipping every second cache line when allocating client slots. In experiments where memory management is needed (stacks and queues), cache-aligned memory chunks are allocated and deallocated using per-thread pools (we use the implementation provided by the authors of CC-Synch [FK12]).

In all delegation implementations, clients pass pointers to functions that the servicing thread should execute. An alternative, used in Chapter 6, is to pass an opcode (usually an integer) the server can use to decide what to execute, thus avoiding function pointers [CCPG13]. However, we do not use that optimization here because it did not show performance benefits on the x86 processors: Synchronization overheads dominate the overhead of a function call. Moreover, we have observed that the function call is mostly "absorbed" by the surrounding code, *i.e.* it is executed in cycles that would otherwise remain idle.

[2]We have also done single-socket experiments on the Xeon, but our optimizations are not a good fit for that case, because intra-socket cache coherence has very different characteristics, such as relying on the inclusive L3 cache, and very short communication latencies. The Opteron has only 6 cores per socket, which is not enough parallelism to make strong conclusions in this case.



(a) Xeon E7-L8867



(b) Opteron 6176

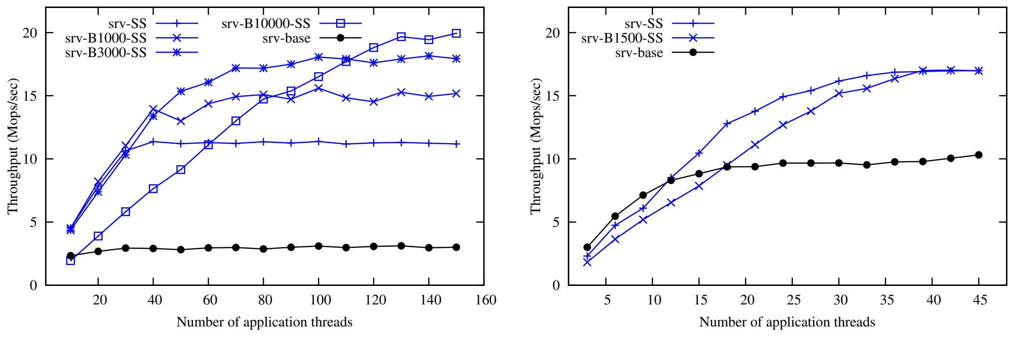
Figure 7.5 – Impact of local backoff on delegation throughput. *srv-base* is the implementation of Algorithm 5 before our optimizations. Suffix *Bx* corresponds to an implementation with a backoff of *x* CPU cycles. Suffix *shuf* denotes cache line shuffling (cache lines are not sequentially, but randomly read by the server).

7.2.2 Analysis of the optimization performance

We present the performance of Algorithm 5 with and without the optimizations proposed in Section 7.1. To do so, we implement a concurrent counter, which supports only one operation, $fetch\_and\_add$ (atomically increment the counter and returns its previous value).

First we evaluate the impact of local backoff in Figure 7.5. We can see that it significantly improves throughput in most concurrency levels on both processors. The performance increase is up to 6x (2x) on the Xeon (Opteron). Increasing the backoff duration above a certain value does not increase the throughput further, most likely because the backoff is sufficient to fully avoid collision with the prefetcher. To confirm that the performance increase comes from minimizing the spinning-prefetching collision, we include an implementation where the server does not access client slots sequentially, but randomly. This results in an irregular access pattern at the server, which is harder to track by the prefetcher. As can be seen in the figure, such shuffling of client slots greatly reduces performance when backoff is used, which is due to less prefetching. On the other hand, shuffling has little or no effect when backoff is not employed (see *srv-base* vs *srv-base-shuf*): Due to the spinning-prefetching collision, every response written by the server still causes a cache miss, so the bottleneck stays the same as without prefetching.

Figure 7.6 shows the impact of using streaming instead of ordinary stores. There is a visible throughput increase of 3.5x (1.7x) on the Xeon (Opteron) with respect to the baseline performance, which confirms that streaming stores are a good choice for throughput optimization. Further, we examine local backoff effectiveness in this case. The results are different on the two tested processors. On the Opteron, backoff on top of streaming stores does not result in a further performance increase, meaning that applying either backoff or streaming stores



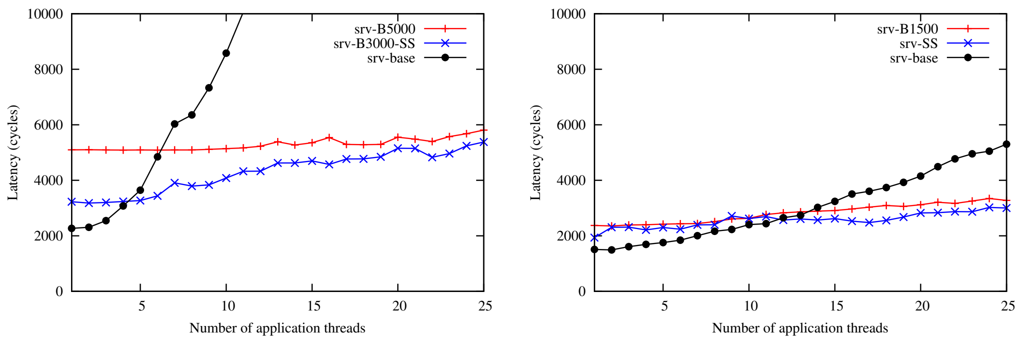
(a) Xeon E7-L8867 (b) Opteron 6176

Figure 7.6 – Impact of streaming stores on delegation throughput (*SS* denotes that streaming stores are used). *srv-base* is the implementation of Algorithm 5 before our optimizations. Local backoff is also evaluated: suffix *Bx* corresponds to an implementation with a backoff of *x* CPU cycles.

in isolation is already enough for attaining the highest throughput. This is not a surprise, since the spinning-prefetching collision is not expected when streaming stores are used (cf. Section 7.1). However, the result on the Xeon does not follow this logic – adding local backoff helps even on top of streaming stores. Because the implementation of streaming stores is not documented in detail, we have done additional experiments to get a better understanding of this behavior. These experiments indicate that there is a conflict: If there is an outstanding streaming store to a cache line from core A, its performance is significantly impaired by core B spinning on the same cache line. The pending streaming store invalidates the copy on core B, which immediately issues another read request, since it is spinning. This newly generated read request apparently obstructs the streaming store, causing it to take about 3x more time to complete. This obstruction is avoided by adding backoff. Higher backoff values help because such conflicts become less probable. This is a strong hint that the spinning-prefetching collision is not the only reason why local spinning can hamper performance: Other characteristics of the machine at hand may incur it as well. The conflict was irreproducible with both normal and streaming stores on the Opteron, and with normal stores on the Xeon.

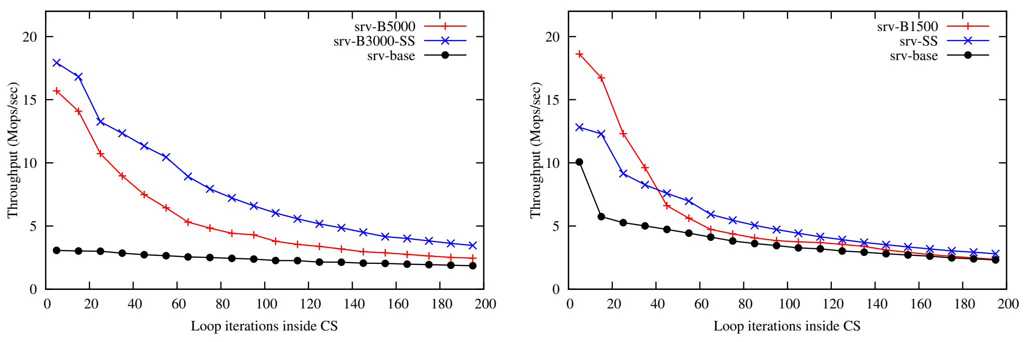
In the above experiments, we can see that there is a tradeoff involved in choosing the best backoff duration. Increasing it improves throughput (to some extent), but at the expense of worsening low-concurrency performance. Choosing the right value depents on the targeted application. In the rest of this section, we have chosen values that attain high throughput without unreasonably increasing latency in low concurrency levels (*srv-B5000* and *srv-B3000-SS* on the Xeon, *srv-B1500* and *srv-SS* on the Opteron).

We quantify more precisely the impact of local backoff and streaming stores in cases of little or no concurrency, by observing average request latency in Figure 7.7. Not surprisingly, the baseline implementation performs best in the lowest concurrency levels. The latency of



(a) Xeon E7-L8867 (b) Opteron 6176

Figure 7.7 – Latency evaluation with local backoff and streaming stores. Suffix *SS* – streaming stores are used; suffix *Bx* – backoff of *x* CPU cycles is used.

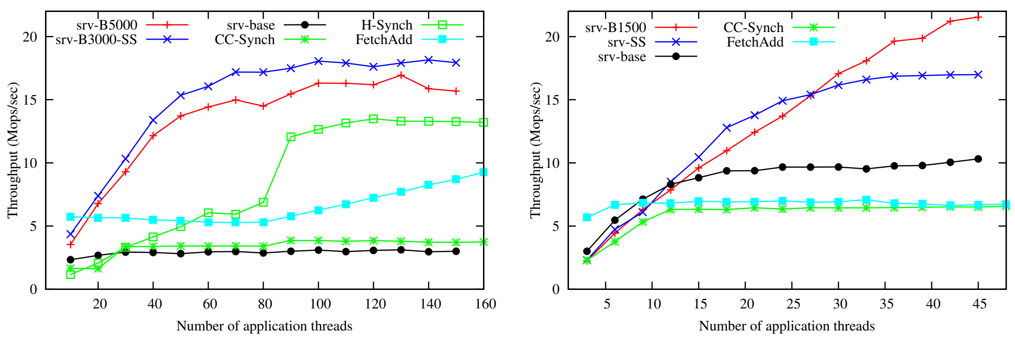


(a) Xeon E7-L8867 (b) Opteron 6176

Figure 7.8 – Maximum throughput with long critical sections. Inside the CS, elements of an array of 64 integers are incremented in a loop, one increment per loop iteration. Suffix *SS* – streaming stores are used; Suffix *Bx* – backoff of *x* CPU cycles is used.

backoff-based implementations is mostly dependant on the chosen backoff duration, which only adds overhead in case of few active threads. However, even the backoff values in the figure, chosen for high throughput, do not lead to excessively high latency. With the exception of *srv-B5000*, they are within 1.6x of the latency of *srv-base* even with only one client thread. Note that the small diference is partly due to the test configuration, which stresses the general case of cross-socket communication: Delegation within a socket, when possible, would exhibit lower latencies. Overall, backoff and streaming stores are not the best fit for low-concurrency cases, but as the level of concurrency increases, they become a more and more appealing alternative. This is expected, because both optimizations deliberately trade low-concurrency for high-concurrency performance.

Now we measure performance with a longer critical section. Instead of one counter increment



(a) Xeon E7-L8867 (b) Opteron 6176

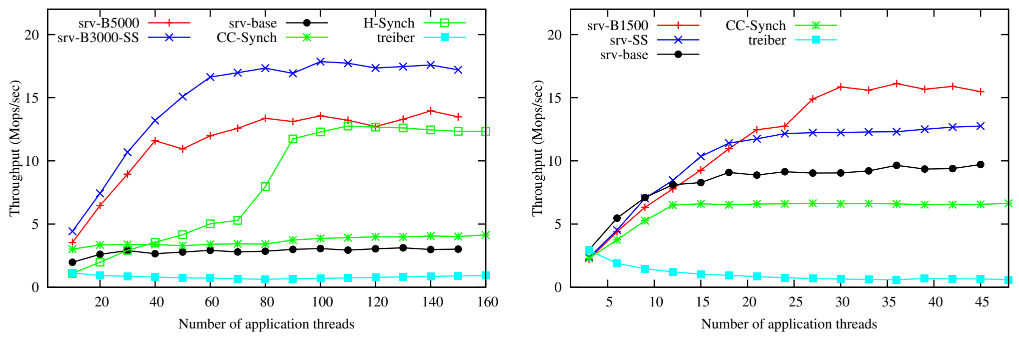
Figure 7.9 – Performance of concurrent counters. *srv-** – server-based implementations; *CC-Synch, H-Synch* – combining implementations [FK12]; *FetchAdd* – hardware fetch-and-add instruction

as in previous experiments, we allocate an array of 64 integers and the critical section consists of incrementing each integer sequentially (modulo 64) in a loop. The number of loop iterations varies. We stress the server with the maximum number of clients (158 on the Xeon and 47 on the Opteron) executing this CS, and we plot the result in Figure 7.8. As the critical section size increases, it starts dominating the synchronization overhead and optimizations become less and less relevant. However, even at 200 loop iterations there are still visible benefits: the version optimized using streaming stores outperforms the baseline implementation by 1.84x (1.19x) on the Xeon (Opteron). Still, it should be noted that this experiment serves only as a rough estimate of what happens with longer critical sections, as it does not simulate many things that a real-life critical section might do, such as cache and TLB misses, floating-point operations, etc. Thus, actual performance impact should be evaluated on a case-by-case basis, which we do next, by evaluating concurrent objects that should benefit the most from the proposed optimizations.

7.2.3 Concurrent data structures

Here we use delegation to come up with efficient implementations of some ubiquitous concurrent objects, counters, stacks and queues, and we compare them with well-known existing implementations.

Figure 7.9 gives the performance of different concurrent counters. Besides the server-based implementations, CC-Synch, and H-Synch, we also include a concurrent counter trivially implemented using the atomic *fetch-and-add* instruction. In high concurrency levels, our optimized *srv* implementations consistently outperform all other counters. CC-Synch achieves performance similar to that of *srv-base*, which is not surprising, given that the servicing threads in both implementations have a similar communication pattern – two cache misses

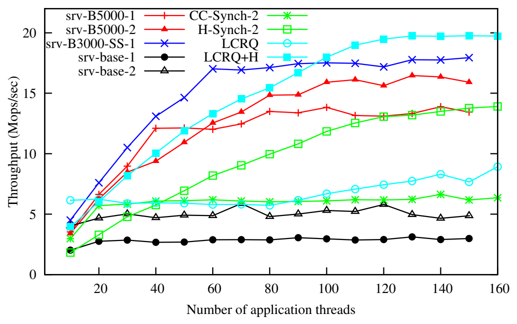


(a) Xeon E7-L8867 (b) Opteron 6176

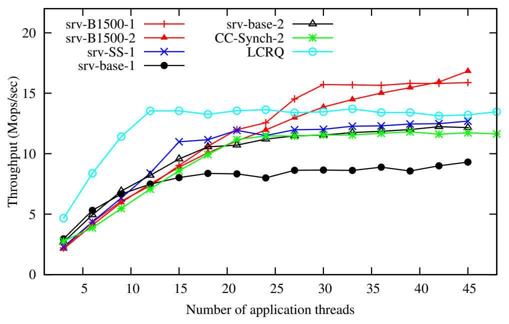
Figure 7.10 – Performance of concurrent stacks (initially empty) under balanced load (every thread alternates between *push* and *pop*). *srv-** – server-based implementations; *CC-Synch, H-Synch* – combining implementations [FK12]; *treiber* – Treiber's nonblocking stack [Tre86]

at the server (combiner) per operation and no further optimizations. On the Xeon, H-Synch gives a significant performance improvement over CC-Synch because of its NUMA-awareness, indicating a striking difference in inter- and intra-socket communication costs. Still, optimized *srv* performs even better in most concurrency levels, although it does not take into account the processor's NUMA characteristics. This shows that cross-socket communication does not necessarily need to be eliminated to achieve high throughput: Identifying important latencies and removing them from the critical path, as we do here, can yield even better results. Perhaps surprisingly, even the *fetch-and-add* counter reaches far lower throughput than srv. This is mostly because every core has to bring the counter to the local cache in order to increment it, so the cache line containing the counter bounces between operations, which often includes a cross-socket transfer. On the Xeon, we can also see that *fetch-and-add* performance, after a period of stability, suddenly grows again with more than 80 threads. This is because each newly added thread is co-located with an existing thread on the same physical core (because of Hyperthreading). When the counter's cache line is brough to a core's cache, increments of both threads sharing that core are often executed together, which avoids cache misses and thus improves performance.

Now we examine implementations of concurrent stacks (Figure 7.10). They are implemented straightforwardly from the sequential specification of a stack, by putting the code of push and pop operations inside a critical section. As in Chapter 6, we do not include an elimination layer in our implementations, as it is orthogonal to the main topic of this work. Stacks based on a server, CC-Synch, H-Synch, as well as the nonblocking Treiber stack [Tre86] are presented. The results with CC-Synch, H-Synch, and a server are very similar to those in Figure 7.9 (counter), which is not a surprise because both counters and stacks are implemented with only one server. There is only a small difference in the peak throughput: Since the stack's push and pop operations include more work than incrementing a counter (data allocation/deallocation and a short linked list manipulation), critical sections are longer and the server can execute



(a) Xeon E7-L8867



(b) Opteron 6176

Figure 7.11 – Performance of concurrent queues (initially empty) under balanced load (every thread alternates between *enqueue* and *dequeue*). *srv-** – server-based implementations of blocking MS-Queue [MS96]; *CC-Synch, H-Synch* – combining implementations [FK12] of blocking MS-Queue [MS96]; *LCRQ, LCRQ+H* – nonblocking queues for x86 [MA13]; suffix *-x* is the number of locks used in MS-Queue implementations

fewer operations in a unit of time. The performance drop is more visible on the Opteron: a possible reason is that the Xeon's more complex prefetching logic is able to hide the latency of the cache miss that happens when a new stack element is allocated.

Finally, we compare concurrent FIFO queue implementations. In contrast to the implemented counters and stacks, where the object is coarsely locked, queues allow a certain degree of fine-grained locking. For instance, as mentioned in Chapter 6, the Michael and Scott blocking queue (MS-Queue), uses two separate locks, operating at opposite ends of the queue (one for enqueue, and the other for dequeue operations) [MS96]. We implement MS-Queue using two servers, as well as using two CC-Synch/H-Synch combiners. Again, besides two-lock implementations, we also evaluate one-lock ones, for two reasons. First, that enables us to directly quantify the benefit from introducing fine-grained locking. Second, the streaming-store optimization is not applicable to fine-grained MS-Queue, as explained in Section 7.1. In addition to these different implementations of MS-Queue, nonblocking LCRQ [MA13], as well as its hierarchical version, LCRQ+H, are included (with the ring size of 2^{17}, and for LCRQ+H, timeout set to 400 Kcycles). LCRQ is specifically designed with x86 processors in mind and is therefore expected to perform well.

The results are shown in Figure 7.11. First we discuss the MS-Queue implementations. Fine-grained locking significantly improves the performance of the *srv-base*, CC-Synch and H-Synch queues implemented using a single lock (*CC-Synch-1* and *H-Synch-1* not shown to avoid clutter), but has a much less pronounced impact on the optimized server implementations, especially on the Opteron, where it does not give any tangible benefits over the coarse-grained version. We believe one reason is the hardware prefetcher, which has a more complex task in this case. When there is a coarse lock on an object, there is only one server executing all critical sections, so cache misses mostly originate from client-server communication, since the data structure itself, once allocated, stays in the server's cache. In case of fine-grained locking, however, data locality is suboptimal because the queue is directly accessed by two server threads and the data needs to move – typically, when a dequeuing server dequeues an item, it incurs a cache miss, since the item is in the enqueuing server's cache. With more cache misses coming from data accesses, the pattern observed by the prefetcher becomes less regular and the performance drops.

Nevertheless, our optimized implementations still provide competitive performance (even those without fine-grained locking): In high concurrency levels, they reach the highest throughput on the Opteron, and are only outperformed by LCRQ+H on the Xeon. However, it should be noted that the NUMA-awareness strategy used by LCRQ+H trades performance for fairness. In the presented experiment, the *fairness ratio* of LCRQ+H, *i.e.*, the ratio between the highest and the lowest number of operations executed by some thread during a time interval, was typically 1.4x. At the same time, the server and combiner-based implementations exhibit almost perfect fairness (every thread executed nearly the same number of operations). In more detail, with LCRQ+H, at every point there is one *active* NUMA socket – any operations from other sockets are paused for a certain amount of time, and then they try to make their socket active [MA13]. The duration of this pause is a tradeoff – higher values give a better NUMA locality and thus higher throughput, but some nodes are increasingly likely to starve. The result shown in Figure 7.11 is for the pause of 400 Kcycles. With a 1 Mcycle pause, maximum throughput grows over 30 Mops/sec, significantly outperforming the other queues, but with lower fairness – a typical fairness ratio in high concurrency was 4.

7.3 Discussion

The above experiments show that local backoff and streaming stores can dramatically improve delegation performance in many cases. It turns out that simple hardware-aware optimizations play a key role in optimizing concurrent code, which corroborates recent results, stating that synchronization performance is mainly a hardware property [DGT13].

It is also noteworthy that there is a number of other details at the level of cache coherence protocols that can affect delegation performance. For example, we have experimented with different placements of client communication slots across sockets (recall that they are allocated at the server's socket in Section 7.2). This turned out to have a surprisingly big performance impact, most likely because of different work distributions between coherence agents. However, exploring this in more details is hard without knowing the inner workings of the cache coherence protocol.

Even with the proposed optimizations, there is still ample space for further improvement. In terms of throughput, we can see that the best result is about 20 Mops/sec on both processors for a concurrent counter, which means that the server takes about 100 CPU cycles to process every request. Since the critical section itself is very short in this case (only a couple of cycles to access a variable in the L1 cache), we can conclude that the rest is pure synchronization overhead. Indeed, hardware event counters indicate that the server core is stalled most of the time, even after applying our optimizations. We believe this is mostly due to unsuccessful or partial prefetching, as well as to the cost of flushing the streaming-store buffer.

In Chapter 6, we showed that on a processor provided with hardware message-passing support, such stalls on the server can be fully avoided. Reaching the same result exclusively over cache-coherent shared-memory would probably require being able to specify the cache of a remote core where data should be placed, as proposed in [PYK+13]. Such a solution would allow a client to specify that its request should be moved to the server cache, avoiding the need to rely on hardware prefetchers to transfer cache lines in time to avoid stalls. Our experiments also show that the solutions that optimize throughput are detrimental to latency in low concurrency. On the contrary, hardware message-passing or cache-aware instructions [PYK+13] allow achieving both high throughput and low latency. Considering the relatively low performance our optimized technique achieves compared to a solution based on hardware message passing, and the huge number of experiments we had to conduct to understand how these optimizations interact with the cache coherence system, we argue that the easiest and most efficient approach to thread synchronization at large scale is to provide hardware features such as those previously mentioned.

7.4 Summary

The chapter presents two optimizations for delegation over cache-coherent shared memory: (i) backoff in local-spin loops to minimize collision with hardware prefetchers and (ii) weakly-ordered streaming stores to avoid memory-model limitations. Although simple, these two optimizations subtly interact with the cache-coherency protocol and the hardware prefetchers of modern x86 processors to achieve unprecedented throughput for the execution of critical sections. Hence, concurrent counters, stacks, and queues implemented with our optimized delegation solution outperform the most efficient NUMA-oblivious and NUMA-aware, both blocking and nonblocking alternatives in most cases, especially under heavy contention.

Nevertheless, the performance results are visibly inferior to those obtained with the help of hardware message passing (Chapter 6). This confirms that hardware features that enable better control over inter-core communication are very desirable for making thread synchronization faster.

Concluding Remarks and Bibliography

8 Conclusions

We have presented novel algorithms, models, and optimizations for manycore machines. For emerging message-passing processors, the thesis introduces two flavors of OC-BCAST, a novel tree-based broadcast algorithm based on one-sided communication. OC-BCAST leverages parallelism offered by on-chip one-sided operations to significantly decrease latency and increase throughput with respect to well-known alternatives, built on top of the traditional two-sided interface. The advantages of OC-BCAST have been confirmed (i) experimentally, on the Intel SCC processor, and (ii) analytically, using a performance model that captures all communication costs on the critical path of the different broadcast algorithms.

For traditional shared-memory processors, we have demonstrated how hardware extensions for message passing can be leveraged to obtain *hybrid* mutual exclusion algorithms. Indeed, HYBLOCK and HYBCOMB are the first of their kind: They use both message passing and cache-coherent shared memory. This enables unprecedented performance, thanks to message passing; Still, the algorithms remain reasonable in terms of complexity, thanks to shared memory. Our algorithms have been shown to outperform their state-of-the-art counterparts in many scenarios, including implementations of concurrent counters, queues and stacks on the Tilera TILE-Gx processor.

Besides the direct algorithmic contributions, the thesis gives insights into pros and cons of message passing and shared memory. Our study confirms that message passing, with appropriate hardware support, is a very powerful tool for designing fast communication and synchronization algorithms with modelable performance. This stems from the fact that nothing is abstracted away from the programmer, as data exchange between cores is explicit and direct. That being said, the convenience and ease of use that coherent shared memory provides should by no means be neglected, as it removes the burden of having to think about each and every data exchange, even when these are not crucially important from the performance point of view. Thus, hardware support for both programming models is useful and lets the programmer choose where each of them is a better fit. Indeed, our hybrid algorithms demonstrate how coexistence of both on the same chip can be put to good use.

We believe that many of the presented insights can be used in the design of communication and synchronization patterns not covered by this thesis. Some ideas for future work are the following:

- **Extending the broadcast study to other collective operations.** The one-sided interface of the Intel SCC and similar processors can be used to provide efficient implementations of other collectives. In particular, other one-to-all operations should be easily amenable to the ideas used in OC-BCAST.
- **Using hardware message passing in other contexts.** Besides mutual exclusion, we believe that hardware support for message passing can be employed in other kinds of algorithms. For example, the RCU synchronization technique [GMTW08] lends itself nicely to it, since some of its efficient implementations involve exchange of OS-level signals, which can be replaced by hardware messages. Next, the use of private queues and message passing has been discussed as a way to implement fence-free work stealing in parallel frameworks [ACR13]: Hardware messaging could possibly improve the performance of this technique.
- **Studying energy efficiency.** This thesis mainly deals with performance aspects of many-core communication and synchronization. In Section 6.1, however, we point out that replacing local spinning with waiting for a message could have a positive impact on energy efficiency. Quantifying this potential with different synchronization algorithms and benchmarks, as well as developing energy-aware algorithms, is an interesting direction for future work.
- **Applying delegation to existing concurrent software with minimal effort.** One of the problems of delegation is its interface, which requires critical sections to be encapsulated as functions. Because of this, applying it to software that uses traditional locks requires either significant manual effort or special tools [GMTW08, KSW14]. Still, adapting delegation algorithms to the standard lock/unlock interface can be done, using a low-level technique that involves context transfer (program counter and registers) between application threads. It would be interesting to investigate the overhead of this technique, and more precisely, whether it cancels out the potential performance gain that delegation provides.

Bibliography

[ACR13] U. Acar, A. Chargueraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013. URL: http://doi.acm.org/10.1145/2442516.2442538, `doi:10.1145/2442516.2442538`.

[Ada14] Adapteva. Parallella manycore processor. http://www.parallella.com, 2014. Accessed: 09-12-2014.

[AFA11] J. L. Abellán, J. Fernández, and M. E. Acacio. GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, 2011. URL: http://dx.doi.org/10.1109/IPDPS.2011.87, `doi:10.1109/IPDPS.2011.87`.

[AHA$^+$05] G. Almási, P. Heidelberger, C.J. Archer, X. Martorell, C.C. Erway, J.E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2005.

[AISS95] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, 1995. URL: http://doi.acm.org/10.1145/215399.215427, `doi:10.1145/215399.215427`.

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), 1967. URL: http://doi.acm.org/10.1145/1465482.1465560, `doi:10.1145/1465482.1465560`.

[ASHAA97] H. Abdel-Shafi, J. Hall, S. Adve, and V. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, 1997. `doi:10.1109/HPCA.1997.569661`.

[BBD$^+$09] A. Baumann, P. Barham, P-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009. URL: http://doi.acm.org/10.1145/1629575.1629579, doi:10.1145/1629575.1629579.

[BFPS11] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, 2011. URL: http://dx.doi.org/10.1109/IGCC.2011.6008565, doi:10.1109/IGCC.2011.6008565.

[BHU$^+$97] J. Bruck, C-T. Ho, E. Upfal, S. Kipnis, and D. 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. doi:10.1109/71.642949.

[BJK$^+$96] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996. URL: http://www.sciencedirect.com/science/article/pii/S0743731596901070, doi:http://dx.doi.org/10.1006/jpdc.1996.0107.

[BMR05] O. Beaumont, L. Marchal, and Y. Robert. Broadcast trees for heterogeneous platforms. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 2005.

[BWKMZ12] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, 2012.

[CCPG13] J. Cleary, O. Callanan, M. Purcell, and D. Gregg. Fast asymmetric thread synchronization. *ACM Transactions on Architecture and Code Optimization*, 9(4):27:1–27:22, January 2013. URL: http://doi.acm.org/10.1145/2400682.2400686, doi:10.1145/2400682.2400686.

[CGH13] I. Calciu, J. Gottschlich, and M. Herlihy. Using elimination and delegation to implement a scalable numa-friendly stack. In *5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

[Cha10] E. Chan. RCCE comm: A Collective Communication Library for the Intel Single-chip Cloud Computer. 2010.

[CKP$^+$93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. 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, 1993. URL: http://doi.acm.org/10.1145/155332.155333, doi:http://doi.acm.org/10.1145/155332.155333.

[CKS$^+$11] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. Adve, V. Adve, N. Carter, and C-T. Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011. URL: http://dx.doi.org/10.1109/PACT.2011.21, doi:10.1109/PACT.2011.21.

[CLMY96] D. Culler, L. T. Liu, R. Martin, and C. Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. In *IEEE Micro*, February 1996.

[CLRB11] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, july 2011. doi:10.1109/HPCSim.2011.5999870.

[Cra93] T. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, University of Washington, February 1993.

[CURK11] I. Compres Urena, M. Riepen, and M. Konow. RCKMPI–Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). *Recent Advances in the Message Passing Interface*, 2011.

[DGT13] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013. URL: http://doi.acm.org/10.1145/2517349.2522714, doi:10.1145/2517349.2522714.

[DGY14] T. David, R. Guerraoui, and M. Yabandeh. Consensus inside. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, 2014. URL: http://doi.acm.org/10.1145/2663165.2663321, doi:10.1145/2663165.2663321.

[DHW97] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *J. ACM*, 44(6):779–805, November 1997. URL: http://doi.acm.org/10.1145/268999.269000, doi:10.1145/268999.269000.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998. doi:10.1109/99.660313.

[FK11] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, 2011. URL: http://doi.acm.org/10.1145/1989493.1989549, doi:10.1145/1989493.1989549.

[FK12] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012. URL: http://doi.acm.org/10.1145/2145816.2145849, doi:10.1145/2145816.2145849.

[FK14] P. Fatourou and N. Kallimanis. Brief announcement: The power of scheduling-aware synchronization. In *Proceedings of the 28th international conference on Distributed computing*, DISC '14, 2014.

[GBPN03] R. Gupta, P. Balaji, D.K. Panda, and J. Nieplocha. Efficient collective operations using remote memory operations on VIA-based clusters. In *Proceedings of International Parallel and Distributed Processing Symposium, 2003.*, 2003.

[GFB+04] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2004.

[GGT12] V. Gramoli, R. Guerraoui, and V. Trigonakis. TM2C: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*, 2012. URL: http://doi.acm.org/10.1145/2168836.2168872, `doi:10.1145/2168836.2168872`.

[GMTW08] D. Guniguntala, P.E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, 2008. `doi:10.1147/sj.472.0221`.

[Gus88] J. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. URL: http://doi.acm.org/10.1145/42411.42415, `doi:10.1145/42411.42415`.

[GVW89] J. Goodman, M. Vernon, and P. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, 1989. URL: http://doi.acm.org/10.1145/70082.68188, `doi:10.1145/70082.68188`.

[HB09] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 1st edition, 2009.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International IEEE Solid-State Circuits Conference Digest of Technical Papers*, 2010.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. URL: http://doi.acm.org/10.1145/114005.102808, `doi:10.1145/114005.102808`.

[Her93] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions Programming Languages and Systems*, 15(5):745–770, November 1993. URL: http://doi.acm.org/10.1145/161468.161469, `doi:10.1145/161468.161469`.

[HIST10] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, 2010. URL: http://doi.acm.org/10.1145/1810479.1810540, `doi:10.1145/1810479.1810540`.

[HLS95] M. Herlihy, B-H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, November 1995. URL: http://doi.acm.org/10.1145/210223.210225, `doi:10.1145/210223.210225`.

[HM93] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. URL: http://doi.acm.org/10.1145/173682.165164, `doi:10.1145/173682.165164`.

[HS08] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HSR07] T. Hoefler, C. Siebert, and W. Rehm. A Practically Constant-Time MPI Broadcast Algorithm for Large-Scale InfiniBand Clusters with Multicast. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*. IEEE, 2007.

[Int14] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*, February 2014. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[K$^+$08] P. Kogge et al. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA, 2008.

[Kal14] Kalray. Kalray manycore processors. http://www.kalray.eu, 2014. Accessed: 09-12-2014.

[KK79] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks (1976)*, 3(4):267–286, 1979.

[KRGF12] A. Kohler, M. Radetzki, P. Gschwandtner, and T. Fahringer. Low-latency collectives for the Intel SCC. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, Sept 2012. `doi:10.1109/CLUSTER.2012.58`.

[KSSS93] R. Karp, A. Sahay, E. Santos, and K. Schauser. Optimal broadcast and summation in the LogP model. In *Proceedings of the Fifth Annual ACM Symposium on*

Parallel Algorithms and Architectures, SPAA '93, 1993. URL: http://doi.acm.org/10.1145/165231.165250, `doi:10.1145/165231.165250`.

[KSW14] D. Klaftenegger, K. Sagonas, and K. Winblad. Brief announcement: Queue delegation locking. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, 2014. URL: http://doi.acm.org/10.1145/2612669.2612714, `doi:10.1145/2612669.2612714`.

[LDT+12] J-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.

[LMP04] J. Liu, A.R. Mamidala, and D.K. Panda. Fast and scalable MPI-level broadcast using infiniband's hardware multicast support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[LWK+03] J. Liu, J. Wu, S. Kini, P. Wyckoff, and D. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, 2003. URL: http://doi.acm.org/10.1145/782814.782855, `doi:http://doi.acm.org/10.1145/782814.782855`.

[MA13] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013. URL: http://doi.acm.org/10.1145/2442516.2442527, `doi:10.1145/2442516.2442527`.

[McC08] M.D. McCool. Scalable programming models for massively multicore processors. *Proceedings of the IEEE*, 96(5):816–831, May 2008. `doi:10.1109/JPROC.2008.917731`.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991. URL: http://doi.acm.org/10.1145/103727.103729, `doi:10.1145/103727.103729`.

[MF98] C. Moritz and M. Frank. LoGPC: Modeling network contention in message-passing programs. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '98/PERFORMANCE '98, 1998. URL: http://doi.acm.org/10.1145/277851.277933, `doi:10.1145/277851.277933`.

[MHS12] M. Martin, M. Hill, and D. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, July 2012. URL: http://doi.acm.org/10.1145/2209249.2209269, `doi:10.1145/2209249.2209269`.

[MJ13] J. Matienzo and N.E. Jerger. Performance analysis of broadcasting algorithms on the Intel Single-Chip Cloud computer. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, April 2013. doi:10.1109/ISPASS.2013.6557167.

[MLH94] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, 1994.

[MS96] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996. URL: http://doi.acm.org/10.1145/248052.248106, doi:10.1145/248052.248106.

[MVDW10] T. Mattson and R. Van Der Wijngaart. RCCE: a Small Library for Many-Core Communication. *Intel Corporation, May*, 2010.

[MVdWF08] T. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[MZK12] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHASH: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012. URL: http://doi.acm.org/10.1145/2145816.2145874, doi:10.1145/2145816.2145874.

[Nis09] R. Nishtala. *Automatically Tuning Collective Communication for One-Sided Programming Models.* PhD thesis, University of California at Berkeley, 2009.

[NTA96] M. Naimi, M. Trehel, and A. Arnold. A log (n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, April 1996. URL: http://dx.doi.org/10.1006/jpdc.1996.0041, doi:10.1006/jpdc.1996.0041.

[ONH$^+$96] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, 1996. URL: http://doi.acm.org/10.1145/237090.237140, doi:10.1145/237090.237140.

[OSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, 2009.

[OTY99] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop* *on Parallel and Distributed Computing for Symbolic and Irregular Applications*, 1999.

[PRS14] D. Petrović, T. Ropars, and A. 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, Orlando, Florida, USA, February 2014. URL: http://doi.acm.org/10.1145/2555243.2555251.

[PRS15] D. Petrović, T. Ropars, and A. Schiper. On the performance of delegation over cache-coherent shared memory. In *Proceedings of the 16th International Conference on Distributed Computing and Networking*, ICDCN '15, Goa, India, January 2015. URL: http://dx.doi.org/10.1145/2684464.2684476.

[PSMR11] S. Peter, A. Schpbach, D. Menzi, and T. Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *3rd Many-core Applications Research Community (MARC) Symposium, Fraunhofer IOSB, Ettlingen, Germany*, 2011.

[PSRS12a] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper. Asynchronous Broadcast on the Intel SCC using Interrupts. In *The 6th Many-core Applications Research Community (MARC) Symposium*, Toulouse, France, July 2012. URL: https://hal.archives-ouvertes.fr/hal-00719022.

[PSRS12b] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper. High-performance RMA-based Broadcast on the Intel SCC. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, Pittsburgh, Pennsylvania, USA, June 2012. URL: http://doi.acm.org/10.1145/2312005.2312029.

[PYK$^+$13] J. Park, R. Yoo, D. Khudia, C. Hughes, and D. Kim. Location-aware cache management for many-core processors with deep cache hierarchy. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013. URL: http://doi.acm.org/10.1145/2503210.2503224, `doi:10.1145/2503210.2503224`.

[Rei07] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[RH13] S. Ramos and T. 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*, HPDC '13, 2013. URL: http://doi.acm.org/10.1145/2462902.2462916, `doi:10.1145/2462902.2462916`.

[Rot11] R. Rotta. On Efficient Message Passing on the Intel SCC. In *3rd Many-core Applications Research Community (MARC) Symposium, Fraunhofer IOSB, Ettlingen, Germany*, 2011.

[SA15] H. Sung and S. Adve. DeNovoSync: Efficient support for arbitrary synchronization without writer-initiated invalidations. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[SBM+05] S. Sur, U. Bondhugula, A. Mamidala, H.W. Jin, and D. Panda. High performance RDMA based all-to-all broadcast for InfiniBand clusters. *High Performance Computing–HiPC 2005*, 2005.

[SHW11] D. Sorin, M. Hill, and D. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[SKA13] H. Sung, R. Komuravelli, and S. Adve. DeNovoND: Efficient hardware support for disciplined non-determinism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013. URL: http://doi.acm.org/10.1145/2451116.2451119, doi:10.1145/2451116.2451119.

[SMQP10] M. A. Suleman, O. Mutlu, M. Qureshi, and Y. Patt. Accelerating Critical Section Execution with Asymmetric Multicore Architectures. *IEEE Micro*, 30(1):60–70, January 2010.

[SOHL+98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core.* MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[SST09] P. Sanders, J. Speck, and J. Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.*, 35(12):581–594, December 2009. URL: http://dx.doi.org/10.1016/j.parco.2009.09.001, doi:10.1016/j.parco.2009.09.001.

[ST95] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *Proceedings of the 7th annual ACM symposium on Parallel algorithms and architectures*, 1995. URL: http://doi.acm.org/10.1145/215399.215419, doi:10.1145/215399.215419.

[SVDG00] M. Shroff and R.A. Van De Geijn. Collmark: MPI collective communication benchmark. Technical report, 2000.

[SYK10] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010. URL: http://doi.acm.org/10.1145/1736020.1736055, doi:10.1145/1736020.1736055.

[TA00] Infiniband Trade Association. *InfiniBand Architecture Specification: Release 1.0.* InfiniBand Trade Association, 2000.

[Til14] Tilera. Tilera manycore processors. http://www.tilera.com, 2014. Accessed: 09-12-2014.

[Tor09] J. Torrellas. Architectures for Extreme-Scale Computing. *IEEE Computer*, 42(11):28 –35, nov. 2009.

[Tre86] R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[TRG05] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[vdWMH11] R.F. van der Wijngaart, T.G. Mattson, and W. Haas. Light-weight communications on Intel's Single-Chip Cloud computer processor. *ACM SIGOPS Operating Systems Review*, 45(1):73–83, 2011.

[VGvT$^+$11] 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 Many-core Applications Research Community (MARC) Symposium, Fraunhofer IOSB, Ettlingen, Germany*, 2011.

[vTBV$^+$11] M.W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C.R. Jesshope. Efficient Memory Copy Operations on the 48-core Intel SCC Processor. In *3rd Many-core Applications Research Community (MARC) Symposium, Fraunhofer IOSB, Ettlingen, Germany*, number 1098, 2011.

[WA09] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43(2):76–85, April 2009. URL: http://doi.acm.org/10.1145/1531793.1531805, `doi:10.1145/1531793.1531805`.

Darko Petrović

Curriculum Vitae

Education and Research

2010–2015	**Ph.D., Computer Science**, *EPFL*, Lausanne. ○ *thesis advisor*: prof. André Schiper ○ *title*: Efficient Synchronization and Communication on Manycore Processors
2008–2010	**Master, Embedded Systems Design**, *ALaRI, University of Lugano*. USImpresa Fund award for the highest GPA
2003–2008	**Dipl.Ing., Computer Engineering**, *ETF, University of Belgrade*, Serbia.

Industrial Experience

2013–2014	**Research Intern**, *ABB Corporate Research*, Baden-Dättwil, Switzerland.
2008	**Software Developer**, *Merit Solutions*, Belgrade, Serbia.
2007	**Student Collaborator**, *ELSYS Eastern Europe*, Belgrade, Serbia.

Lead-author Publications

ICDCN 2015	**On the Performance of Delegation over Cache-Coherent Shared Memory**.
PPoPP 2014	**Leveraging Hardware Message Passing for Efficient Thread Synchronization**, *invited to ACM TOPC (under review)*.
MARC 2012	**Ansynchronous Broadcast on the Intel SCC using Interrupts**.
SPAA 2012	**High-Performance RMA-Based Broadcast on the Intel SCC**.
AINA 2012	**Implementing Virtual Machine Replication: a Case Study using Xen and KVM**.

Languages

Serbo-Croatian – native

English – fluent

French – fluent

German – intermediate

Italian – conversational