High-Throughput Maps on Message-Passing Manycore Architectures: Partitioning versus Replication

Omid Shahmirzadi, Thomas Ropars, André Schiper
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

Abstract

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

Keywords: High-throughput Map, Message-Passing Manycore Architecture, Replication, Partitioning

1 Introduction

Manycore architectures, featuring tens if not hundreds of cores, are becoming available. Taking advantage of the high degree of parallelism provided by such architectures is challenging and raises questions about the programming model to be used [26, 19]. Most existing architectures are still based on cache-coherent shared memory but some provide message passing, through a highly efficient network-on-chip (NoC), as a basic means to communicate between cores [17, 5, 2]. Designing a scalable concurrent algorithm for cache-coherent architectures is a difficult task because it requires understanding the subtleties of the underlying cache coherence protocol [10]. On the other hand, message passing looks appealing because it provides the programmer with explicit control of the communication between cores. However, compared to the vast literature on concurrent programming in shared-memory systems [16], programming message-passing processors is not yet a mature research topic.

Implementing scalable data structures is one of the basic problems in concurrent programming. To increase the throughput of data structures in shared memory architectures, several well-known techniques can be used including fine-grained locking, optimistic synchronization and lazy synchronization [16]. In the case of message-passing systems, partitioning and replication are the two main approaches to improve the throughput of concurrent data structures [13]. Using partitioning, a data structure is partitioned among a set of servers that answer clients requests. Using replication, each client has a local copy of data structure in its private memory. Both have been considered in recent work on message-passing manycores [7, 29, 9], but performance comparisons are lacking. In this paper we present a performance comparison of these two approaches for the implementation of high-throughput concurrent objects in message-passing manycores, considering the case of a linearizable map. Note that existing studies made in distributed message-passing systems are only of little help because the high performance of NoCs provides a completely different ratio between computation and communication costs compared to large scale distributed systems.

Maps are used in many systems ranging from operating systems [7, 29] to key-value stores [9]. Their performance is often crucial to the systems using them. A map is an interesting case study because it is a good candidate to apply both partitioning and replication techniques. Since operations on different keys are independent, maps are easily partitionable [9]. Because a large majority of operations are usually lookup operations [6], replication can help handling a large number of local lookup requests concurrently.

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

This paper is organized as follows. Section 3 specifies the underlying assumptions and goal of the study. Section 4 presents the algorithms, modeling methodology and its validation on the TILE-Gx processor. Section 5 studies performance of the algorithms on different architectures. Related work and conclusion are presented in Sections 6 and 7.

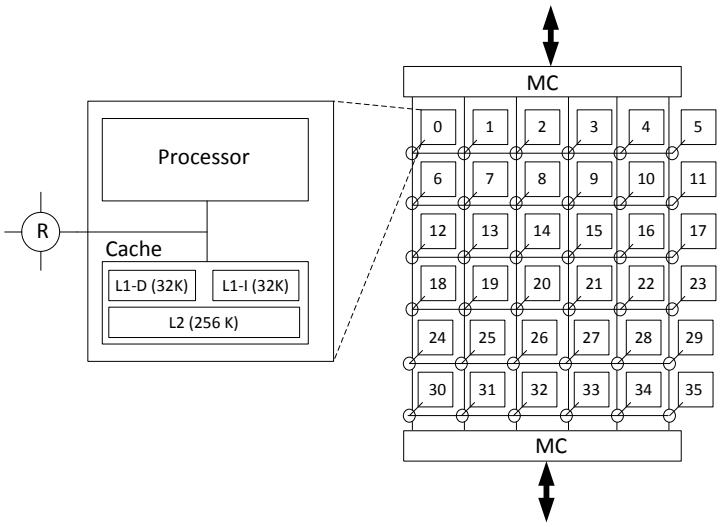


Figure 1: TILE-Gx8036 architecture

2 The Tilera TILE-Gx8036

The TILE-Gx8036 is a general-purpose manycore developed by TILERA Corporation [5]. We use this platform as the baseline architecture for our studies. In this section we describe the high level TILE-Gx8036 architecture and inter-core communication.

2.1 Architecture

The cores and the NoC of the TILE-Gx8036 are depicted in Figure 1. There are 36 full-fledged 1.2 Ghz, 64-bit processor cores with local cache, connected through a 2D mesh NoC. Each tile is connected to a router. The NoC uses high-throughput, low-latency links as well as deterministic X-Y routing. Cores and mesh operate at the same frequency.

Memory components are divided into (i) L1 data and instruction cache (32 KB each), (ii) 256 KB of L2 cache, and (iii) off-chip global memory. There is full hardware cache coherence among the L1 and L2 caches of different cores. Each core has access to the off-chip global memory through one of the two memory controllers, denoted by MC in Figure 1. Regions of the global memory can be declared private or shared (a page is a unit of granularity). We see this platform as a pure message-passing manycore, where each thread binds to a specific core and has its own private memory space.

2.2 Inter-core communication

Each core has a dedicated hardware message buffer, capable of storing up to 118 64-bit words. The message buffer of each core is 4-way multiplexed, which means that every per-core buffer can host up to four independent hardware FIFO queues, containing incoming messages. The User Dynamic Network (UDN) allows applications to exchange messages directly through the mesh interconnect, without OS intervention, using special instructions. When a thread wants to exchange messages, it must be pinned to a core and registered to use the UDN (but it can unregister and freely migrate afterwards). When a message is sent from core A to core B, it is stored in the specified hardware queue of core B. The *send* operation is asynchronous and does not block, except in the following case. Since messages are never dropped, if a hardware queue is full, subsequent incoming messages back up into the network and may cause the sender to block. It is the

programmer's responsibility to avoid deadlocks that can occur in such situations. When a core executes the *receive* instruction on one of the four local queues, the first message from the queue is returned. If there are no messages, the core blocks. The user can send and receive messages consisting of one or multiple words. Moreover a core upon receipt of a new message in either of its incoming buffers, has the option of being notified by an inter core interrupt followed by executing an interrupt handler routine.

3 Assumptions and Goal

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

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

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

4 Algorithms and Analytical Modeling

This section describes the algorithms studied in this chapter and presents their performance model. We start by describing our methodology for performance modeling followed by describing and modeling the linearizable and sequential consistent map algorithms. The main reason to use an analytical model is to be able to compare replication and partitioning in a general case so that the final conclusions are not biased towards features of an existing platforms, *e.g.* TILE-Gx. However, as we will see in Section 5, analytical modeling also helps us to concretely understand the performance bottlenecks and to be able to assess the algorithms under different architectures, configurations and load distributions. Moreover it can help manycore programmers to decide about their implementation choice on different platforms.

4.1 Performance modeling

Manycore processors are usually provided with a highly efficient NoC. Therefore, we assume that the throughput of the algorithms presented in this section is limited by the performance of the cores. This assumption is validated by the experimental results presented in Section 5.2. Hence, to obtain the *maximum* obtainable throughput of one algorithm executed on a given number of cores, we need to compute T_{lup} and T_{upd}, the total number of CPU cycles[1] required to execute a $lookup$ and an $update$ operation respectively.

All algorithms make the difference between cores that execute as *clients*, *i.e.*, cores executing the user code and issuing operations on the concurrent map, and *servers*, *i.e.*, cores that are earmarked to execute map-related and/or protocol code. Depending on the number c of cores that execute the client code and the

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

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

Table 1: Model parameters

number s of cores that execute as server, clients or servers can be the bottleneck for the system throughput. Thus, for each operation op, we actually have to compute the number of CPU cycles it takes on the client (T^c_{op}) and on the server (T^s_{op}). Considering a load where the probability of having a lookup operation is p, and assuming that the load is evenly distributed among clients, the maximum throughput $\mathscr{T}^c$ achievable by clients is:

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

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

$$\mathscr{T} = min(\mathscr{T}^c, \mathscr{T}^s) \tag{2}$$

Table 1 lists the parameters that we use to describe the performance of our algorithms. To model the operations on the map, we consider a generic map implementation defined by three parameters o_{pre}, o_{lup} and o_{upd}. The underlying data structure used to implement the map is not the focus of the study. Parameter o_{pre} corresponds to the computation that a client has to do before accessing the map, *e.g.*, executing a hash function if a hash table is used to implement the map. Parameters o_{lup} and o_{upd} are the overheads corresponding to accessing the underlying data structure during a $lookup$ and an $update$ operation respectively.

We associate an overhead (*i.e.*, duration) in CPU cycles with each of the communication primitives introduced in Section 3. Additionally, we introduce the parameter T_{rtt}, representing round-trip time. More precisely, $T_{rtt}(send\_op, rcv\_op)$ is the round-trip time for messages sent with the $send\_op$ operation (*i.e.*, $send$, $broadcast$ or $multicast$) and received with the rcv_op operation (*i.e.*, rcv or $arcv$)[2]. If the round trip is initiated with $broadcast$ or $multicast$, it finishes when the answer from all destinations have been received.

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

In the following, we describe the different algorithms studied in this chapter, considering linearizable maps and sequential consistent maps respectively. For each algorithm, we provide a figure describing the

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

communication patterns where all CPU overheads appear. We deduce the performance models directly from these figures.

4.2 Linearizable map

Our goal is to propose linearizable map algorithms that are representative of the design space in a message-passing manycore. Hence, as a basic solution based on partitioning, we consider the approach proposed in [9]: the map is partitioned among a set of servers that clients access for every requests. A typical improvement of such a client/server approach is to introduce cache on client side [27]. We study a second algorithm based on this solution. Regarding replication, the solutions used in distributed systems cannot be directly applied to message-passing manycore: In a distributed system, a server is typically replicated to reduce the latency observed by clients by placing the replicas closer to the clients. In a manycore chip, the NoC provides very low latency between cores. Creating a few replicas of a server hosting a map is not an interesting approach. The only advantage it provides is to allow processing multiple lookup operations in parallel. However, this cannot make replication attractive compared to partitioning since partitioning provides the same advantage without the complexity of ensuring replica consistency during update. Thus, the only way for replication to provide benefits in the context of manycores, is to have a replica of the map on each core, so that clients can lookup the keys locally. We study three replication algorithms based on this idea. The first is based on the traditional approach consisting in using atomic broadcast to implement update operations. With such a solution, lookups require remote synchronization to ensure linearizability. Hence, one can argue that the goal of replication is not achieved. That is why we propose a second algorithm where lookups do not require any remote synchronization. In this case, update operations have to be made more complex to ensure linearizability. However both of the former replication solutions, need sequencer servers to provide total order. To come up with a server-less protocol, we bring a variant of two phase commit protocol in which the lookups are purely local without any remote synchronization. However getting rid of the servers, comes up with a price: the issuer of the update needs to abort the operation and issue it again in case of another conflicting update.

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

4.2.1 Partitioned map (PART_SIMPLE)

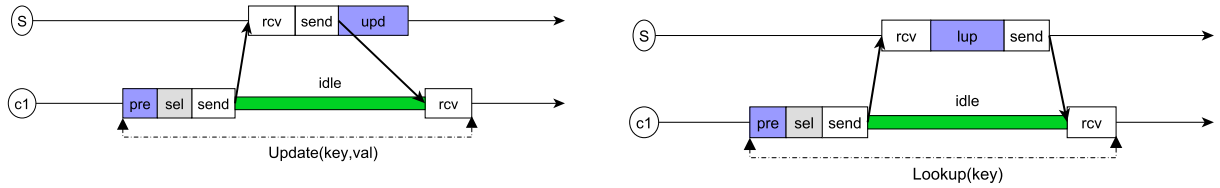
In this approach called PART_SIMPLE, each server handles a subset of the keys. In this algorithm each client contacts a corresponding server to perform lookup and update on a key. Both operations block until the client receives a response from the server, which trivially ensures linearizability. The pseudocode of this algorithm is given in Figure 3[3]. The communication pattern is described in Figure 2. It is the same for a $lookup$ and an $update$ operation. The only difference is that applying the update can be removed from the critical path of the client (see Figure 2(a)). Computing T^s_{op} (where op is upd or lup), T^c_{lup} and T^c_{upd} based on Figure 2 is trivial:

$$T^s_{op} = o_{rcv} + o_{op} + o_{send} \tag{3}$$

$$T^c_{lup} = o_{pre} + o_{sel} + T_{rtt}(send, rcv) + o_{lup} \tag{4}$$

$$T^c_{upd} = o_{pre} + o_{sel} + T_{rtt}(send, rcv) \tag{5}$$

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



(a) Update (b) Lookup

Figure 2: Simple partitioning

Algorithm 1 PART_SIMPLE (code for client c)

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

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

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

Algorithm 2 PART_SIMPLE (code for server s)

Local Variables:
1: map {map partition }

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

Figure 3: Linearizable partitioned map without caching

4.2.2 Replicated map – Lookups with remote synchronization (REP_REMOTE)

In replication approaches, lookups should be synchronized with updates to avoid violating linearizability as illustrated by Figure 4, where lookups return locally with no synchronization. Moreover all updates should be applied in total order in all replicas. The first two replication solutions provide total order among updates using atomic broadcast while the third solution ensures it using a variant of two phase commit protocol. In the first replication algorithm, called REP_REMOTE, lookups are totally ordered with respect to update operations.

Before detailing the algorithm, we need to discuss the atomic broadcast (*abcast*) implementation. To choose among the five classes of atomic broadcast algorithms presented in [12], we use three criteria. First, the number of messages exchanged during *abcast* should be minimized to limit the CPU cycles used for communication. This implies that solutions relying on multiple calls to broadcast should be avoided. Second, the solution should allow to increase the throughput by instantiating multiple instances of the *abcast* algorithm. Indeed, to obtain a linearizable map, only the operations on the same key have to be ordered. Thus, if *abcast* is the bottleneck, being able to use multiple instances of *abcast*, each associated with a subset of the keys, can increase the system throughput. Finally, the performance of the algorithm should not be impacted if some processes do not have messages to broadcast. Indeed, if multiple instances of *abcast* are used, we cannot assume that all processes will always have requests to *abcast* for each subset of keys. Only

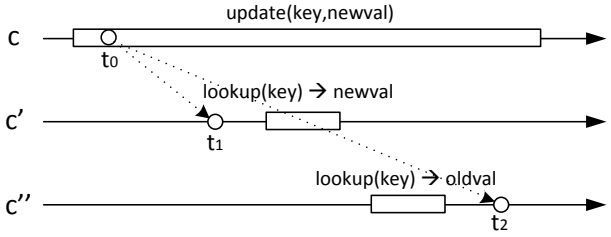


Figure 4: Non-linearizable execution with a replicated map

fixed-sequencer-based algorithms meet all the criteria.

In a fixed sequencer atomic broadcast algorithm, one process (called server in the following) is in charge of assigning sequence numbers to messages. After contacting the sequencer, the thread calling *abcast* can broadcast the message and the sequence number. The communication pattern of REP_REMOTE for an update issued by client c is shown in Figure 5(a). For each lookup, the client has to contact the server in charge of the key to know the sequence number sn of the last update ordered by this server (see Figure 5(b)). Then, the lookup terminates once the client has delivered the update with sequence number sn. Pseudecode of this algorithm is given in Figure 6 and its correctness trivially follows. Note that in this algorithms delegating the task of broadcast to the server could lead to violation of linearizability: if an update on a key finishes on the issuing client before the corresponding value on the server is updated, a later lookup could still return the old value.

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

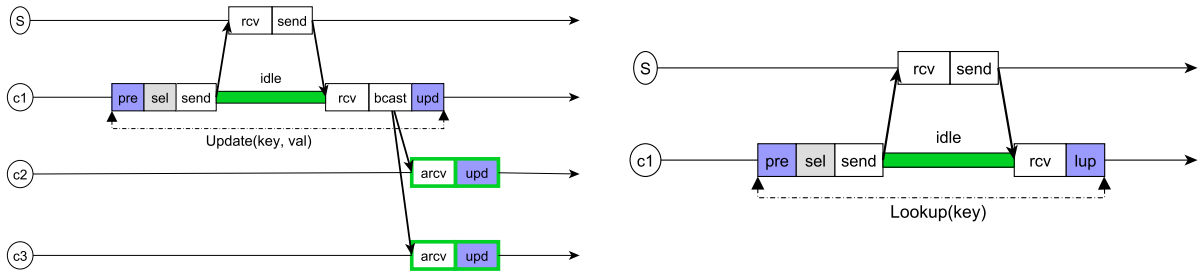
Computing the throughput of clients in this algorithm is complex because clients can be interrupted to deliver updates. But handling the interrupts is not always on the critical path of the clients. Indeed, one can notice that clients are idle during an operation while waiting for an answer from the server. An interrupt handled during this period would not be on the critical path. We define $\mathscr{O}^c$ as the maximum amount of time spent in interrupts handling that can be removed from the critical path of clients execution and update formula 57 in the following way:

$$\mathscr{T}^c = \frac{c}{p \cdot T^c_{lup} + (1-p) \cdot T^c_{upd} - \mathscr{O}^c} \tag{6}$$

We deduce the cost of an update and a lookup operation from Figure 5.

$$T^s_{op} = o_{rcv} + o_{send} \tag{7}$$
$$T^c_{lup} = o_{pre} + o_{sel} + o_{lup} + T_{rtt}(send, rcv) \tag{8}$$
$$T^c_{upd} = o_{pre} + o_{sel} + o_{upd} + T_{rtt}(send, rcv) + o_{bcast} + (c-1) \cdot (o_{arcv} + o_{upd}) \tag{9}$$

$\mathscr{O}^c$ depends on T_{idle}, the idle time on a client during one operation, n_{idle}, the average number of idle periods per operation, T_{int}, the time required to handle an interrupt (green-border rectangles in Figure 5(a)),



(a) Update (b) Lookup

Figure 5: Replication with remote synchronization for lookups

and n_{int}, the average number of asynchronous requests per operation:

$$T_{idle} = T_{rtt}(send, rcv) - o_{send} - o_{rcv} \tag{10}$$
$$n_{idle} = 1 \tag{11}$$
$$T_{int} = o_{arcv} + o_{upd} \tag{12}$$
$$n_{int} = (c-1) \cdot (1-p) \tag{13}$$

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

$$k = \lfloor \frac{T_{idle}}{T_{int}} \rfloor \tag{14}$$
$$n_{full} = min(k \cdot n_{idle}, n_{int}) \tag{15}$$
$$n_{partial} = min(n_{int} - n_{full}, n_{idle}) \tag{16}$$
$$\mathscr{O}^c = n_{full} \cdot T_{int} + n_{partial} \cdot (T_{idle} - k \cdot T_{int}) \tag{17}$$

4.2.3 Replicated map – Lookups without remote synchronization (REP_LOCAL)

In REP_LOCAL, lookups do not require any remote synchronization (Figure 7(b)) but updates are more complex than in REP_REMOTE (Figure 7(a)). To provide linearizability, this algorithm ensures that during an update, no lookup can return the new value if a lookup by another client can return an older value. To do so, the update operation includes two phases of communication as shown in Figure 7(a). When client c runs an update, it first atomically broadcasts the update message to all clients. Then it waits until all clients acknowledge the reception of this message. Finally, it broadcasts a second message to validate the update. If a client tries to lookup the key after it has received the update message, the lookup cannot return until the validation has been received. This way a lookup that returns the new value always finishes after all clients have received the update message, which is enough to ensure linearizability [4]. The pseudocode of this algorithm is given in Figure 8. Theorem 4.1 proves the correctness of this algorithm.

Theorem 4.1 *Algorithm 8 ensures linearizability with respect to the map operations.*

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

Algorithm 3 REP_REMOTE (code for replica c)

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

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

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

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

Algorithm 4 REP_REMOTE (code for server s)

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

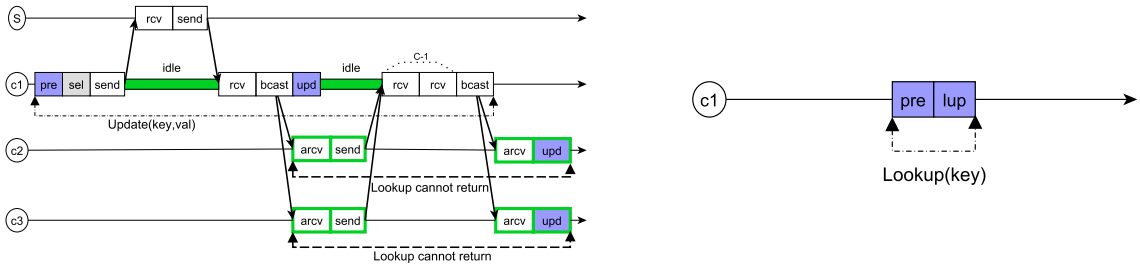
2: **upon rcv** ($command, key$) **from replica** c
3: **if** $command = SNREQ$ **then**
4: $send(abCtr)$ *to* c
5: **else**
6: $abCtr \leftarrow abCtr + 1$
7: $send(abCtr)$ *to* c

Figure 6: Linearizable replicated map with local lookups with remote synchronization

Proof. If we prove that a map with only a single key is linearizable, due to the composability of linearizability, the whole map, which is composed of a set of independent key entries, is linearizable too. Considering only one key, the total order of updates is trivially ensured. Moreover if two updates on a key are executed with no timing overlap, in the global history of updates the second update is placed after the first one, since the first one is assigned with a smaller sequence number. Considering lookups, we show that the two following scenarios are not possible: (1) having two non-overlapping lookups, where the former one returns the new value and the latter one returns the old value, as it is shown in Figure 9(a); and (2) having a non-overlapping update and lookup, where the update happens before the lookup and lookup returns the old value, as it is shown in Figure 9(b). Apart from these two cases, all other scenarios, with respect to the relative position of two operations, are safe with respect to linearizability, *i.e.* a linearizable history can be made.

Case (1): suppose this scenario happens according to Figure 9(a). In this case, we assume replica c_1 updates the new value. Assume t_1 and t_2 are the beginning and the end of the lookup operation on c_2 and t_3 and t_4 are the beginning and the end of the lookup operation on c_3 and $t_1 > t_4$. Replica c_3 should receive the $ACKALL$ for this update at some point before t_4, called A (execution of line 18). However replica c_1 should have sent this update to the replica c_2 at some point after t_1, called B (execution of line 13). Note that B is not necessarily before t_2. This means that $B \rightarrow A$ since replica c_2 should have sent the ACK message to the replica c_1 (execution of line 23), before replica c_1 could send the $ACKALL$ message to replica c_3 (execution of line 16). Therefore $t_1 \rightarrow B$, $B \rightarrow A$, $A \rightarrow t_4$, and so $t_1 \rightarrow t_4$. This means that $t_1 \leq t_4$, a contradiction.

Case (2): Suppose this scenario happens according to Figure 9(b). Assume t_1 and t_2 are the beginning and the end of the lookup operation on c_2 and t_3 and t_4 are the beginning and the end of the update operation on c_1 and $t_1 > t_4$. It means that replica c_2 receives the update from replica c_1 at some point after t_1, called A (execution of line 13). Moreover it means that replica c_1 receives the ACK for this update from replica c_2



(a) Update (b) Lookup

Figure 7: Replication with no remote synchronization for lookups

at some point before t_4, called B (execution of line 15). Therefore we have $t_1 \rightarrow A$, $A \rightarrow B$, $B \rightarrow t_4$, and so $t_1 \rightarrow t_4$. This means $t_1 \leq t_4$, a contradiction. □ □

Since this algorithm introduces idle time on clients and uses interrupts, computing the throughput of clients is based on Formula 6. But it can be noticed that update operations introduce two idle periods with different durations as well as two different costs for handling interrupts:

$$T_{idle\_1} = T_{rtt}(send, rcv) - o_{send} - o_{rcv} \tag{18}$$
$$T_{idle\_2} = max\big(T_{rtt}(bcast, arcv) - o_{bcast} - (c-1) \cdot o_{rcv} - o_{upd}, 0\big) \tag{19}$$
$$n_{idle\_1} = n_{idle\_2} = 1 - p \tag{20}$$
$$T_{int\_1} = o_{arcv} + o_{send} \tag{21}$$
$$T_{int\_2} = o_{arcv} + o_{upd} \tag{22}$$
$$n_{int\_1} = n_{int\_2} = (c-1) \cdot (1-p) \tag{23}$$

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

Computing the exact value of $\mathscr{O}^c$ in this case is a complex problem. Instead, we approximate this value using Formulas 14-17 with weighted averages for T_{idle} and T_{int}. For instance, here are the values we use for n_{idle} and T_{idle}:

$$n_{idle} = n_{idle\_1} + n_{idle\_2} \tag{24}$$
$$T_{idle} = \frac{T_{idle\_1} \cdot n_{idle\_1} + T_{idle\_2} \cdot n_{idle\_2}}{n_{idle}} \tag{25}$$

Finally, we deduce the cost of lookups and updates from Figure 7:

$$T^s_{upd} = o_{rcv} + o_{send} \tag{26}$$
$$T^c_{upd} = o_{pre} + o_{sel} + T_{rtt}(send, rcv) + max\big(T_{rtt}(bcast, arcv), o_{bcast} + o_{upd} + (c-1) \cdot o_{rcv}\big) + o_{bcast} + (c-1) \cdot (2 \cdot o_{arcv} + o_{send} + o_{upd}) \tag{27}$$
$$T^c_{lup} = o_{pre} + o_{lup} \tag{28}$$

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

Previous replicated solutions rely on some dedicated servers to assign the sequence numbers to the messages, in order to ensure total order delivery of updates. However one might save these dedicated servers, by

Algorithm 5 REP_LOCAL (code for replica c)

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

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

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

17: **upon arcv**$(ACKALL, key)$ from some replica c'
18: $flag[c'] \leftarrow nil$

19: **upon adel**(UPD, key, val, sn) from some replica c'
20: $map.update(key, val)$ {asynchronous total order delivery}
21: $maxsn \leftarrow maxsn + 1$
22: $flag[c'] \leftarrow key$
23: $send(ACK, key)$ *to* c'

Algorithm 6 REP_LOCAL (code for server s)

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

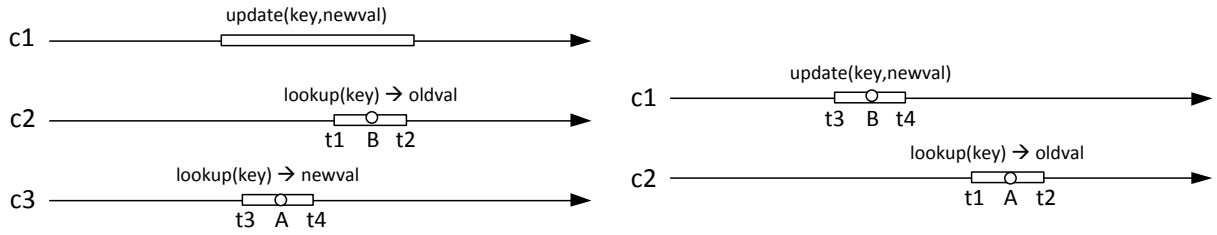
2: **upon rcv** $(SNREQ, key)$ **from some replica** c
3: $abCtr \leftarrow abCtr + 1$
4: $send(abCtr)$ *to* c

Figure 8: Linearizable replicated map with local lookups with no remote synchronization

applying an atomic commit protocol. In this way the solution does not rely on any sequencer, but upon detecting another update on the same key the current update should be aborted. Therefore in REP_2PC, lookups do not require any remote synchronization (Figure 10(b)), but updates are more complex compared to REP_LOCAL (Figure 10(a)).

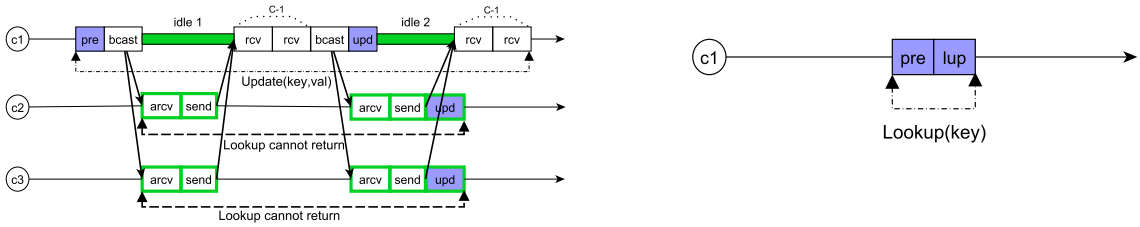
Two phase commit provides total order of updates since as far as an update on a key is executing, other conflicting updates on that key will abort. To be more precise, upon issuing an update, a $VREQ$ message is broadcasted to all the replicas and the issuer is blocked until it receives a vote from all. A YES vote from replica c means that another update on that key is executing on replica c. In this case, the issuer broadcasts an $ABORT$ message to all replicas to abort the current update and returns unsuccessfully. Otherwise it sends a commit message to all other replicas, meaning that it is safe for them to apply the update on that key. Each replica after applying the update sends an ACK back. Upon receiving the ACK from all, the issuing replica terminates the update successfully. However lookups still need to use a similar synchronization technique which is used in REP_LOCAL to ensure linearizability: as far as $flag$ is not equal to -1, meaning that an update is pending on a key, the lookup is not allowed to return the value of that key. The pseudocode of this algorithm is given in Figure 11. Correctness of this algorithm can be proved similarly to the proof of Theorem 4.1.

The parameters needed to obtain the maximum throughput of this algorithm are computed using Figure 10. Note that in this algorithm there is no notion of server, and so server selection cost. The calculation



(a) Case 1 (b) Case 2

Figure 9: Scenarios used to prove the Theorem 4.1



(a) Update with no conflicts (b) Lookup

Figure 10: Replication using two phase commit

methods are similar to those of REP_LOCAL:

$$T_{idle\_1} = max\big(T_{rtt}(bcast, arcv) - o_{bcast} - (c-1)\cdot o_{rcv}, 0\big) \tag{29}$$

$$T_{idle\_2} = max\big(T_{rtt}(bcast, arcv) - o_{bcast} - (c-1)\cdot o_{rcv} - o_{upd}, 0\big) \tag{30}$$

$$n_{idle\_1} = n_{idle\_2} = 1 - p \tag{31}$$

$$T_{int\_1} = o_{arcv} + o_{send} \tag{32}$$

$$T_{int\_2} = o_{arcv} + o_{send} + o_{upd} \tag{33}$$

$$n_{int\_1} = n_{int\_2} = (c-1)\cdot(1-p) \tag{34}$$

$$\begin{aligned} T^c_{upd} = {} & o_{pre} + T_{rtt}(bcast, arcv) + max\big(T_{rtt}(bcast, arcv), o_{bcast} + o_{upd} + (c-1)\cdot o_{rcv}\big) \\ & + (c-1)\cdot(2\cdot o_{arcv} + 2\cdot o_{send} + 2\cdot o_{rcv} + o_{upd}) \end{aligned} \tag{35}$$

$$T^c_{lup} = o_{pre} + o_{lup} \tag{36}$$

4.2.5 Partitioned map - With local caches (PART_CACHING)

The PART_CACHING algorithm extends PART_SIMPLE to introduce caching on client side. If a lookup hits the cache, the pattern is the same as the one in Figure 7(b). Otherwise, the communication pattern is shown in Figure 12(b). It includes a first local lookup that fails and an update of the local cache once the value has been retrieved from the server.

When a key is updated, local copies of the associated value need to be invalidated. As shown in Figure 12(a), the server invalidates local copies using multicast. Once the server has received an acknowledgment from all clients involved, the operation can terminate. This algorithm could also be viewed as a hybrid solution between partitioning and replication, since the local caches are *replicated* on different clients and

Algorithm 7 REP_2PC (code for replica c)

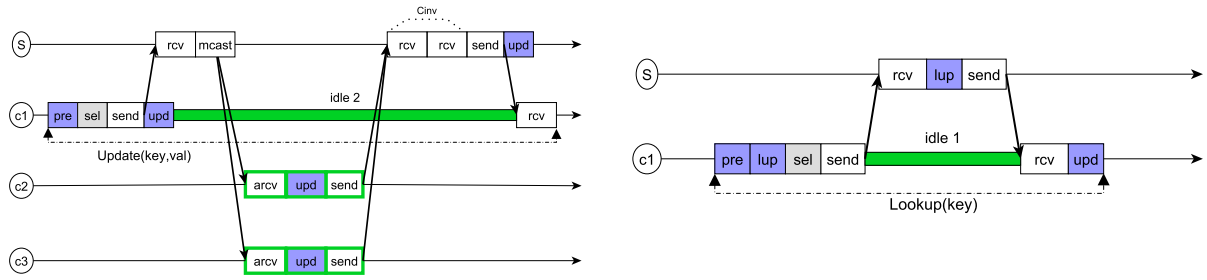
Global Variables:
1: C {total number of clients}
Local Variables:
2: map {map replica }
3: $flag$ {local flags to synchronize lookups on key to ensure linearizability}

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

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

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

Figure 11: Linearizable replicated map with local lookups using two phase commit



(a) Update with invalidation of local caches

(b) Lookup with cache miss

Figure 12: Partitioning with local caches

need to remain consistent among each other using invalidations. The pseudocode of this algorithm is given in Figure 13. Theorem 4.2 proves the correctness of this algorithm.

Theorem 4.2 *Algorithm 13 ensures linearizability with respect to the map operations.*

Proof. Similarly to the proof of Theorem 4.1, we consider the two cases of Figures 14(a) and 14(b), and show that they cannot happen:

Case (1): Suppose that the scenario of Figure 14(a) happens. Assume t_1 and t_2 are the beginning and the end of lookup operation on c_1 and t_3 and t_4 are the beginning and the end of lookup operation on c_2 and $t_1 > t_4$. There are four different subcases considering these two lookup operations. (i) Both lookups are remote: in this case the mentioned scenario in Figure 14(a) is not possible clearly since the second lookup returns a value which is not older than the $newval$. (ii) The first lookup is remote and the second lookup is local: this means that client c_1 receives the invalidation at some point after t_1, called B (not necessarily before t_2). Moreover assume that update of the new value at server s finished at point C (execution of line 6 of the server code). We will have $C \rightarrow t_4$, $B \rightarrow C$ and $t_1 \rightarrow B$, which means $t_1 \rightarrow t_4$, a contradiction.

Algorithm 8 PART_CACHING (code for client c)

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

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

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

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

Algorithm 9 PART_CACHING (code for server s)

Local Variables:
1: map {map partition}

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

Figure 13: Linearizable partitioned map with caching

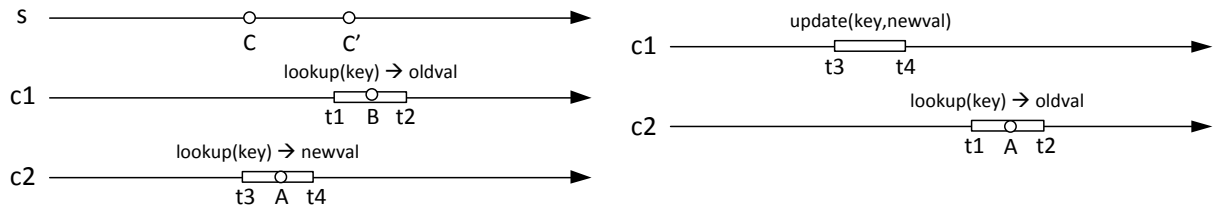
(iii) The first lookup is local and the second lookup is remote: this means that client c_2 updates its local value to the $newval$ at some point before t_4 which is called A (execution of line 11 of the client code). Assume server s sends the new value to client c_2 at point C (execution of line 11 of the server code) and B is the point when the client c_1 executes line 9 of the client code. This means that $C \rightarrow A$, where point A is the time when the client c_2 receives the new value from the server. Therefore we have $t_1 \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$ and $A \rightarrow t_4$ which implies $t_1 \rightarrow t_4$, a contradiction. (iv) Both lookups are local: assume C and C' are the times on the server when it sends to the clients c_1 and c_2 the old and the new values respectively by execution of line 11 of the server code. Clearly C should be before C'. Therefore we have $C' \rightarrow t_4$ and $t_1 \rightarrow C$, and so $t_1 \rightarrow t_4$, a contradiction.

Case (2): Suppose this scenario happens according to Figure 14(b), where client c_1 issues an update with new value and client c_2 issues a lookup which returns the old value. Assume t_1 and t_2 are the beginning and the end of lookup operation on c_2 and t_3 and t_4 are the beginning and the end of the update operation on c_1 and $t_1 > t_4$. Assume that client c_2 returns the lookup value at time A. In this case the invalidation will be received after point A on client c_2. Therefore $t_1 \rightarrow A$ and $A \rightarrow t_4$, and so $t_1 \rightarrow t_4$, a contradiction. □

□

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

$$n_{inv} = \frac{p}{1-p} \cdot (1 - p_{ll}) \tag{37}$$



(a) Case 1 (b) Case 2

Figure 14: Scenarios used to prove the Theorem 4.2

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

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

$$P_{ll} = \sum_{k=0}^{c-1} pmf_{key}(k) \cdot \sum_{i=0}^{\infty} p \cdot pmf_{key}(k) \cdot \Big(p \cdot \big(1 - pmf_{key}(k)\big)\Big)^i \tag{38}$$

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

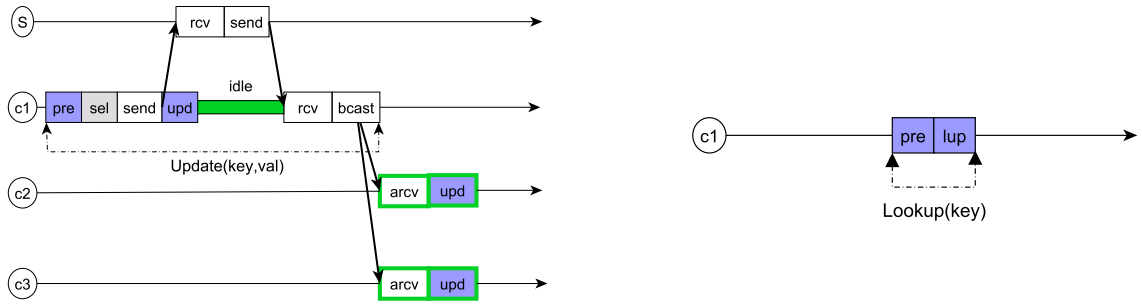
$$P_{ll} = \sum_{i=0}^{\infty} \frac{p}{C} \cdot \big(\frac{p \cdot (C-1)}{C}\big)^i = \frac{p}{p + C \cdot (1-p)} \tag{39}$$

Since the communication patterns for this algorithm includes two idle periods of different duration, we apply Formula 24-25 to compute $\mathscr{O}^c$ with:

$$T_{idle\_1} = o_{lup} + T_{rtt}(send, rcv) - o_{send} - o_{rcv} \tag{40}$$
$$n_{idle\_1} = p \cdot (1 - p_{ll}) \tag{41}$$
$$T_{idle\_2} = max\big(0, T_{rtt}(send, rcv) - o_{send} - o_{rcv} + T_{rtt}(mcast, arcv) - o_{upd}\big) \tag{42}$$
$$n_{idle\_2} = 1 - p \tag{43}$$
$$T_{int} = o_{arcv} + o_{send} + o_{upd} \tag{44}$$
$$n_{int} = (1 - p) \cdot n_{inv} \tag{45}$$

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

The cost of a lookup depends whether there is a cache hit or a cache miss. The cost of a cache hit is the same as a lookup with REP_LOCAL (Formula 56). Otherwise, the cost is given by Figure 12(b). Together



(a) Update (b) Lookup

Figure 15: Sequential consistent replication

we get:

$$T^c_{lup} = p_{ll} \cdot (o_{pre} + o_{lup}) + (1 - p_{ll}) \cdot \big(o_{pre} + 2 \cdot o_{lup} + o_{sel} + T_{rtt}(send, rcv) + o_{upd}\big) \tag{46}$$

$$T^s_{lup} = (1 - p_{ll}) \cdot (o_{rcv} + o_{lup} + o_{send}) \tag{47}$$

The cost of updates is computed based on Figure 12(a). To compute the cost on the server, we do not consider the time it waits for acknowledgments of the invalidation messages as idle time. We assume that the server would always have requests from other clients to handle during this time:

$$\begin{aligned} T^c_{upd} = {} & o_{pre} + o_{sel} + max\big(o_{upd} + o_{send} + o_{rcv}, T_{rtt}(send, rcv) + T_{rtt}(mcast, arcv)\big) \\ & + n_{inv} \cdot (o_{arcv} + o_{send} + o_{upd}) \end{aligned} \tag{48}$$

$$T^s_{upd} = (n_{inv} + 1) \cdot o_{rcv} + o_{mcast} + o_{send} + o_{upd} \tag{49}$$

4.3 Sequential consistent map

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

4.3.1 Replicated map

Considering replication, providing a total order of updates is enough to satisfy sequential consistency. Lookups can return immediately with no synchronization, and they can be freely placed in the global history of update operations to create a global history. Therefore replication algorithms which used a fixed sequencer to create a total order of updates, *i.e.* REP_REMOTE and REP_LOCAL, can be weakened to the algorithm depicted in the Figure 16 (We call this algorithm REP_SC for short). In this algorithm, updates are propagated using atomic broadcast (based on fixed sequencer) and lookups return local values immediately. The communication pattern of this algorithm only includes one idle time, as is shown in Figure 15(a). The

Algorithm 10 REP_SC (code for replica c)

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

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

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

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

Algorithm 11 REP_SC (code for server s)

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

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

Figure 16: Sequential consistent replicated map

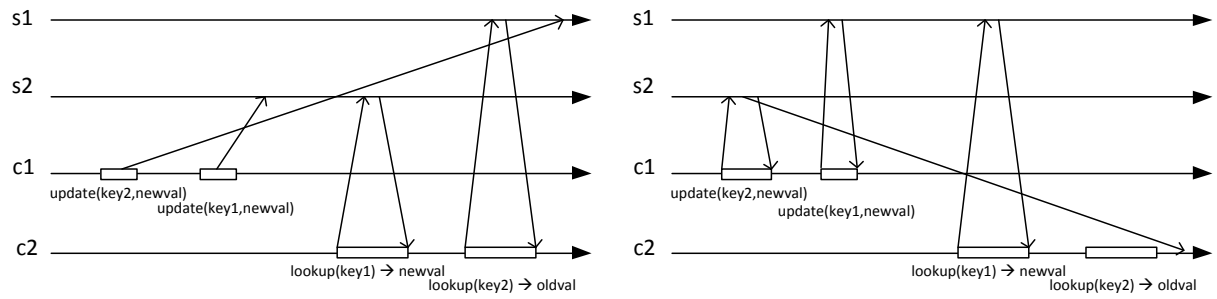
parameters of this algorithm are calculated as follows:

$$T_{idle} = max\big(T_{rtt}(send, rcv) - o_{send} - o_{rcv} - o_{upd}, 0\big) \tag{50}$$
$$n_{idle} = 1 - p \tag{51}$$
$$T_{int} = o_{arcv} + o_{upd} \tag{52}$$
$$n_{int} = (c-1) \cdot (1-p) \tag{53}$$
$$T^s_{upd} = o_{rcv} + o_{send} \tag{54}$$
$$T^c_{upd} = o_{pre} + o_{sel} + max\big(T_{rtt}(send, rcv), o_{send} + o_{rcv} + o_{upd}\big) + o_{bcast} + (c-1) \cdot (o_{arcv} + o_{upd}) \tag{55}$$
$$T^c_{lup} = o_{pre} + o_{lup} \tag{56}$$

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

4.3.2 Partitioned map

To exploit sequential consistency for partitioning solutions, one can think of two optimizations: (1) To make the clients return immediately after sending the update message to the server (which applies to both PART_SIMPLE and PART_CACHING), and (2) to make the server to return immediately after broadcasting invalidation messages to the invalidation set of a key (which only applies to PART_CACHING). In case of having only one server, both optimizations can be applied and resulting algorithms are sequentially consistent. However since sequential consistency is not compositional, having more than one server can break sequential consistency in both cases as they are shown in Figures 17(a) and 17(b). In the first case, consider



(a) Case 1 (b) Case 2

Figure 17: Impossibility of exploiting sequential consistency for partitioning algorithms

the PART_SIMPLE or PART_CACHING in a scenario mentioned in Figure 17(a). The issued update by client c_1 on key_2 arrives to the corresponding server after a long delay. After returning from the first update, it issues another update on key_1 to server s_2. Afterwards client c_2 issues a lookup on key_1, which arrives at s_2 after updating the local value of key_1 to $newVal$, as well as a lookup on key_2 which arrives at s_1 before updating the local value of key_2 to $newVal$. Since the first lookup on key_1 returns $newval$ and the second lookup on key_2 returns $oldval$, it is not possible to create a global history of operations complying with the returned values. In the second case, consider PART_CACHING in a scenario mentioned in Figure 17(b). Assume server s_2 needs to invalidate client c_2 upon receiving an update message on key_2 from c_1. Suppose it takes a long time for this invalidation message to arrive at c_2. Client c_1 issues another update after the first one, which updates the value of key_1 on server s_1 to $newval$. Later client c_2 issues a lookup on key_1 to server s_1, which returns $newval$, while the second lookup on key_2 is done from the local cache, since c_2 has not yet received the invalidation message from server s_2. In this case also it is not possible to create a valid global history of these operations.

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

5 Evaluation

In this section, we first model the communication performance of a Tilera TILE-Gx processor. Then we validate the model of our map algorithms on this platform. Finally, using this model, we conduct a detailed study of the performance of the partitioning and replication algorithms in a message-passing manycore. Throughout this section, we consider a map implemented using a hash table. This is representative of most map implementations [9, 18].

5.1 Modeling TILE-Gx8036

We run experiments on a Tilera TILE-Gx8036 processor. We use it as a representative of current message-passing manycore architectures [5]. Experiments are run with version 2.6.40.38-MDE-4.1.0.148119 of

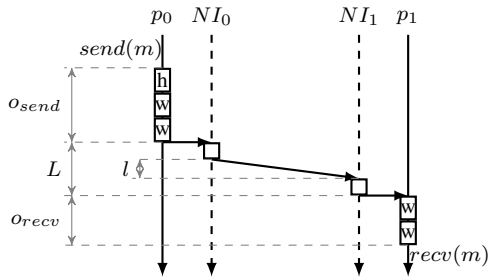


Figure 18: Point-to-point communication on the TILE-Gx for a 2-word message m (NI: network interface)

Tilera's custom Linux kernel. Applications are compiled using GCC 4.4.6 with O3 flag. To implement our algorithms, we use the User Dynamic Network (UDN). In our experiments, we dedicate one queue to *asynchronous* messages: An interrupt in generated each time a new message is available in this queue. Note that the TILE-Gx8036 processor does not provide support for collective operations. Hence, we implement $broadcast$ and $multicast$ as a set of $send$ operations.

Figure 18 describes how we model a point-to-point communication on the TILE-Gx processor. The figure illustrates the case of a 2-word message transmission using $send$ and $recv$. This model is solely based on our evaluations of the communication performance and is only valid for small-sized messages. We do not claim that Figure 18 describes the way communication are actually implemented in the processor.

The overhead o_{send} of a message of n words includes a fix cost of 8 cycles associated with issuing a header packet, plus a variable cost of 1 cycles per word. The overhead o_{recv} is equal to 2 cycles per word. The header packet is not received at the application level. The transmission delay L between the sender and the receiver includes some fix overhead at the network engines on both the sender and the receiver, plus the latency l associated with network traversal. The fix overhead is 10 cycles in total. The latency l depends on the number of routers on the path from the source to the destination: 1 cycle per router. However, on a 36-core mesh the distance between processes has little impact on the performance. Thus, to simplify the study we assume that l is constant and is equal to the average distance between cores, *i.e.*, $l = 6$. Finally, note that there is no *gap* between two consecutive messages sent by the same core.

The first column of Table 2 details the value of the model parameters for the TILE-Gx processor. Our measurements show that the cost of invoking an interrupt handler and restoring the previous context account for 138 cycles. As previously mentioned, we implement $broadcast$ and $multicast$ operations as a sequence of $send$ operations. When the round-trip time is initiated with a collective operation, its duration corresponds to the time required to send all messages plus the time to receive the answer to the last message sent. Finally, we implement the server selection operation using the $modulo$ operation. Its cost o_{sel} varies depending whether the number of server is 2^x (in this case $modulo$ is implemented with a bit-wise AND) or not.

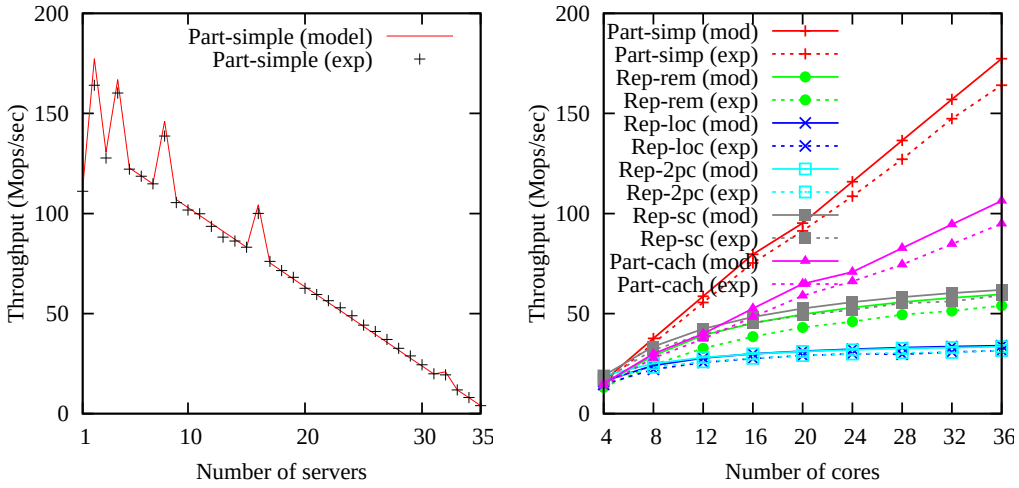
5.2 Model validation

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

Threads are pinned to cores in ascending order: thread t_i is pinned to core i. Note that the size of the messages depends on the algorithms specification. For instance, in PART_SIMPLE, update requests sent to a

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

Table 2: Parameters value in cycles (A "-" means that the value is the same as on TILE-Gx)



(a) Part-simple on 36 cores

(b) Algorithms performance

Figure 19: Model validation on Tilera TILE-Gx processor (90% of lookup operations)

server include 4 words: the id of the sender, the operation id, the hash-key, and the value. The answer is a one-word message containing simply the acknowledgment. The messages size is taken into account for the modeling.

The results presented in Figure 19 assume a load of 90% of lookups ($p = 0.9$). Each point is the average throughput of 6 runs, where in each run every core issues 10000 operations repeatedly on the map. Client threads randomly choose the next operation to execute with a uniform distribution. Keys are distributed among the servers uniformly. Similarly, clients randomly select the key for the next operation with a uniform distribution, *i.e.*, $pmf_{key}(k) = 1/c$. Figure 19(a) shows the variation of the throughput with PART_SIMPLE when the total number of threads is 36 and when the number of server threads varies from 1 to 35. It compares the performance obtained through experiments (dots) and predicted by the model (line). It first shows that the model manages to precisely estimate the performance of the algorithm. The *hiccups* that can be observed are due to the cost of the $modulo$ function used for server selection, and correspond to cases where the number of servers is 2^x. Both the experiments and the model show that the optimal configuration in this case is with 2 servers.

Figure 19(b) presents the maximum throughput of the different algorithms when varying the total num-

ber of threads. To obtain this graph, for each case we run the same test as described by Figure 19(a), and we take the best configuration. This figure shows that we manage to correctly model the performance trends of the algorithms executing on the TILE-Gx processor. Also, it shows that the throughput obtained with the model is always higher than the experimental one. This is expected since the model ignores some computational costs (*e.g.*, operations on private variables) related to the implementation of the algorithms. Additionally, the model considers the maximum *overlapping* $\mathscr{O}^c$ between idle periods and interrupts handling, which is most probably less during experiments. Hence, the model provides an upper bound on the performance of the algorithms, which is at the same time not far from the actual performance. PART_CACHING is the algorithm for which the difference between the model and the experiments is the highest. But even in this case, the difference is at most 12%. Finally, note that in this experiment PART_SIMPLE always outperforms the other solutions. This might be due to the high cost of interrupt handling as well as non-efficient broadcast service which penalizes the other algorithms. Hence these results could not be generalized.

5.3 Analysis of the map algorithms

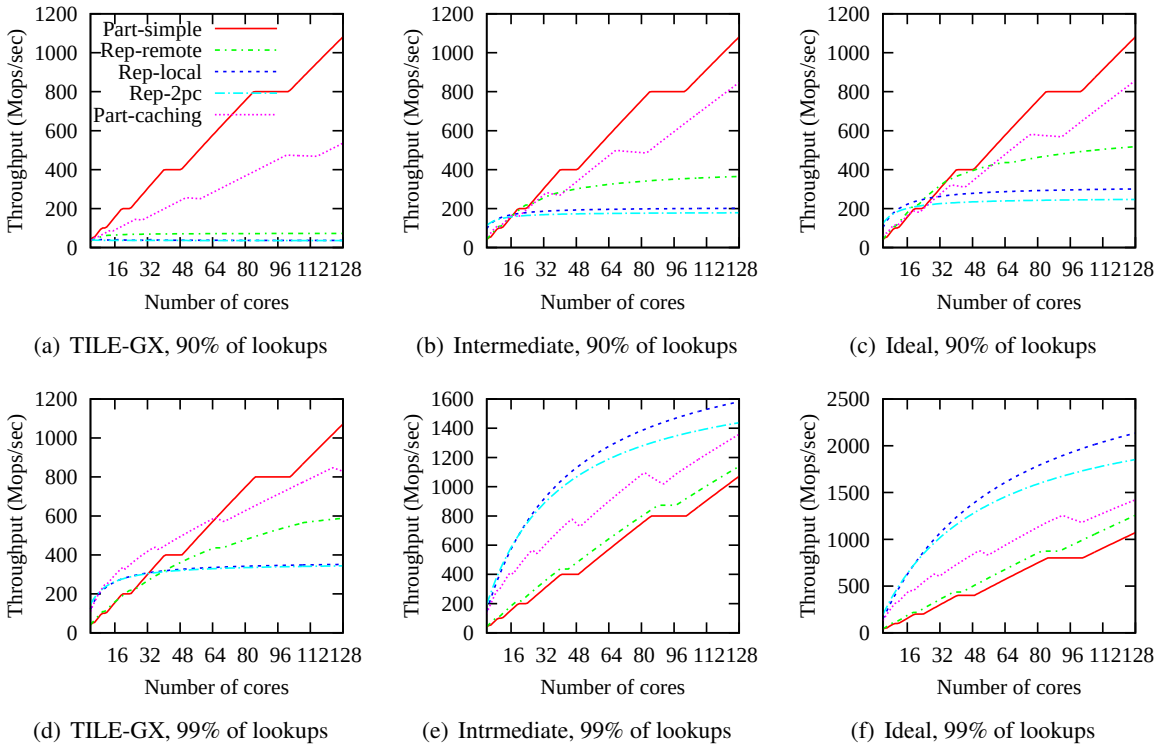
Analytical modeling helps us to do the comparative study of different algorithms under different settings and loads, *e.g.* where the target platform has different architectural features or the load distributions are not uniform. Moreover it helps us to concretely understand the performance bottlenecks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

The first feature is non-efficient broadcast service on Tile-Gx. Due to the lack of a hardware-based broadcast service on this platform, broadcasting to n participants consumes cpu time of n sends and n receives. Note that even the most efficient software implementation of broadcast on top of send and receive primitives, leads to the consumption of the mentioned amount of cpu time [5]. Some recent architectures implement the broadcast service in hardware, *e.g.* Kalray MPPA [2], Adapteva Epiphany [1] and Picochip DSP [4]. To model this feature on these platforms, we assume that the overhead of $broadcast$ and $multicast$ is the same as the overhead of a $send$, which would be the ideal case. Second, even if interrupt handling on the TILE-Gx is rather efficient, its overhead remains high compared to other cpu costs. Solutions have been proposed to save and restore an execution context very efficiently using different architectural and compilation techniques [23, 30, 14, 25]: More specifically in [23], a solution with a constant 4 cycles cost is presented. Hence the second feature we consider is efficient interrupt handling with a cost of 4 cycles.

In order to assess the affect of the mentioned features on the comparative performance of different algorithms, we incrementally define two platforms which do not suffer from them. First we define an *intermediate* platform that has the same characteristics as the TILE-Gx processor but provides efficient asynchronous receives (see Table 2, *intermediate* platform). Second we define an *ideal* platform that has the same characteristics as the *intermediate* platform but also provides hardware-based broadcast service (see Table 2, *ideal* platform).

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

[5] When broadcast is implemented using asynchronous communication, the throughput of the system is independent from the broadcast algorithm [22].



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

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

Figure 20: Performance on the three platforms ($o_{pre} = 12$, $o_{op} = 11$)

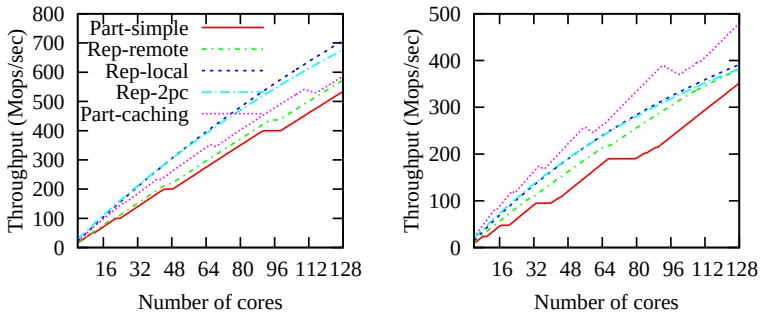
Fifth, we assess the effects of collocating clients and servers on the same core on the performance of the algorithms. Finally we calculate how non-uniform load distribution on the servers can impair the maximum obtainable throughput.

5.3.1 Comparison of the three platforms

Figure 20 shows the performance of different linearizable algorithms as a function of the total number of cores when the percentage of lookups is 90% and 99%, representative loads of many map use-cases [16, 6]. The assumptions made in this evaluation are: i) keys are integers and a simple shift-add hash function is used, *i.e.*, $o_{pre} = 12$; ii) the hash table is small enough to fit into the L2 cache of one core, *i.e.*, we assume that accesses to the hash table cost one L2 access ($o_{op} = 11$)[6]; iii) clients randomly select the key for the next operation with a uniform distribution, *i.e.*, $pmf_{key}(k) = 1/c$. Note that the uniform distribution can be considered as a worst case for PART_CACHING since it implies that the probability that one core issues many lookups on the same key is low. Later we see that a non-uniform key access distribution can improve the performance of PART_CACHING. The two first assumptions are representative of the use of maps in an operating system [18].

Three conclusions that can be drawn from Figure 20. First, if the ratio of lookups is not very high, then partitioning approaches outperforms replication at scale on all platforms (see Figures 20(a) to 20(c)). On the *ideal* platform, REP_LOCAL provides the best performance for 128 cores with 99% of lookups, but the minimum ratio of lookups for REP_LOCAL to be the most efficient in this case is actually 98%. However its throughput reaches a plateau if the total number of cores increases indefinitely. Second, on the TILE-Gx processor, partitioning outperforms replication even if the ratio of lookups is very high (see Figures 20(a)

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



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

Figure 21: Impact of the computational costs (*ideal* platform, 99% of lookups)

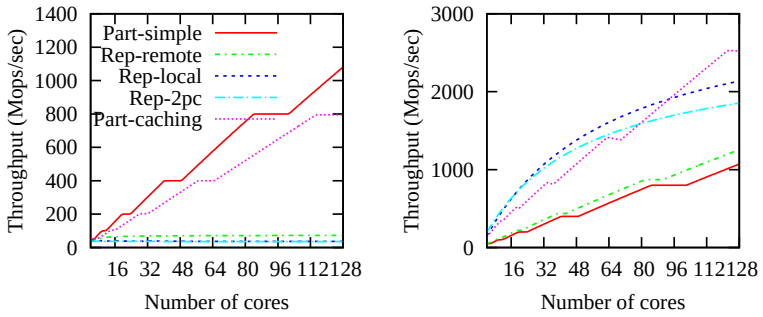
and 20(d)). Replication can outperform partitioning on TILE-Gx only if the lookups are less than 0.1% of the total number of operations. Third, the affect of having broadcast in hardware in comparative performance of different algorithms is much less than providing efficient asynchronous receives. Additional experiments (not presented here) show that these three results remain valid for other values of o_{pre} and o_{op}.

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

5.3.2 Impact of the computational costs

One might wonder if the results displayed in Figure 20 depend on the assumptions made on the map. Figure 21 shows the performance of the linearizable algorithms for other values of o_{pre} and o_{op}. To better assess the impact of these changes, we consider the ideal platform because the relative cost of these parameters is then higher compared to the communication costs. Additionally, we assume a load with 99% of lookups.

Figure 21(a) presents the performance when the hash function cost is 156 cycles, which is a typical cost for a hash function operating on strings. A comparison with Figure 20(f) shows that the maximum throughput of all algorithms decreases but that their relative performance does not change. Figure 21(b) presents the performance when the cost of the operations on the hash table is also increased to 88 cycles. It corresponds to the cost of an access to the main memory. This setting is representative of an in-memory key-value store [3]. In this case, the algorithms based on replication are mainly impacted because the cost of updating the hash table is higher. As a result, compared to Figure 20(f) where REP_LOCAL was providing the best results, PART_CACHING is now the most efficient algorithm. Note that we do not present results for a configuration with a low hash function cost and a high operation cost because we could not find any corresponding use case.



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

Figure 22: Impact of the access pattern ($o_{pre} = 12$, $o_{op} = 11$)

5.3.3 Performance of PART_CACHING with non-uniform client key access

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

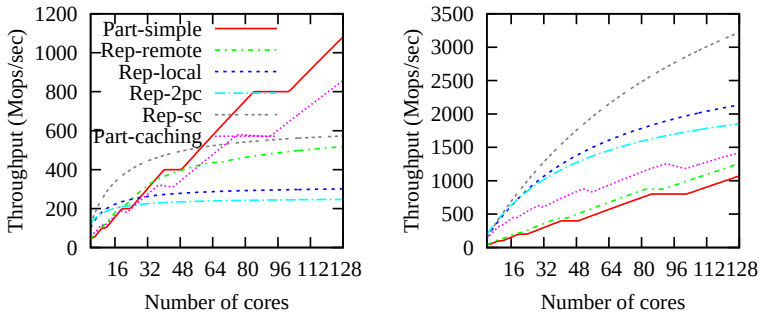
Figure 22 shows the performance with $c_{key} = 4$ and $r = 80$. It considers TILE-Gx with 90% of lookups and the *ideal* platform with 99% of lookups. In both cases, the performance of PART_CACHING is greatly improved. In Figure 22(b), PART_CACHING even outperforms REP_LOCAL.

5.3.4 Impact of weakening consistency criteria to sequential consistency

As we discussed earlier, unlike partitioning solutions, replication solutions are able to exploit sequential consistency. As we saw earlier in comparing linearizable solutions, partitioning is the best approach unless three conditions are met: (i) the percentage of lookups are extremely high; (ii) the cost of asynchronous receives are extremely low; and (iii) the map is located in the cache system of the cores. Provided that these conditions are met, replication can outperform partitioning. In order to understand up to which extent a weaker consistency criteria could be in favor of replication, we compare the performance of REP_SC with other linearizable solutions on the *ideal* platform, where $o_{pre} = 12$, $o_{op} = 11$, for both 90 and 99 percent of lookup workload. As you see in Figure 23(a), with 90 percent of lookups operations REP_SC still cannot beat partitioning solutions at scale, although it outperforms other replication solutions as expected. However as you see in 23(b), with 99 percent of lookups REP_SC outperforms all other solutions significantly. The threshold for percentage of lookup operations in which after that REP_SC outperforms all other algorithm at all scales, is around 95%. This threshold for REP_LOCAL is around 98%, which was mentioned earlier too. Therefore weakening consistency criteria although improves the performance of replication, but still the three conditions are necessary for replication to outperform partitioning, even though partitioning solutions are not able to exploit sequential consistency in their favor.

5.3.5 Collocating clients and servers on the same core

Our evaluations are based on the assumption that clients and servers are located on different cores. One can argue that placing clients and servers on the same core might lead to a better maximum throughput. In this case a core, while playing the role of a server, can receive the requests asynchronously. This strategy does



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

Figure 23: Impact of weakening consistency criteria ($o_{pre} = 12$, $o_{op} = 11$)

not make sense on the TILE-Gx architecture since relative high cost of asynchronous receive is added to the critical path of all operations. However considering the ideal platform, where the cost of asynchronous and synchronous receives are in the same order, it is not clear how this strategy can affect the maximum throughput. Therefore we use our model to obtain the maximum throughput in this case. For the sake of simplicity, we consider the simple partitioning algorithm, PART_SIMPLE. To obtain the maximum throughput of this algorithm, one can consider a total number of C cores partitioned into two sets: the first set S_1, with the size of $C - S$, are those who are purely clients and the second set S_2, with the size of S, are those who collocate clients and servers. To compute the maximum throughput, we obtain the maximum throughput of each set and sum them up. The maximum throughput of S_1 is calculated in the same way as before:

$$\mathscr{T}^{S_1} = min\left(\frac{C - S}{p \cdot T^c_{lup} + (1-p) \cdot T^c_{upd}}, \frac{S}{p \cdot T^s_{lup} + (1-p) \cdot T^s_{upd}}\right) \tag{57}$$

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

$$\mathscr{T}^{S_2*} = \frac{S}{p \cdot (T^c_{lup} + T^s_{lup}) + (1-p) \cdot (T^c_{upd}, T^s_{upd}) - \mathscr{O}^c} \tag{58}$$

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

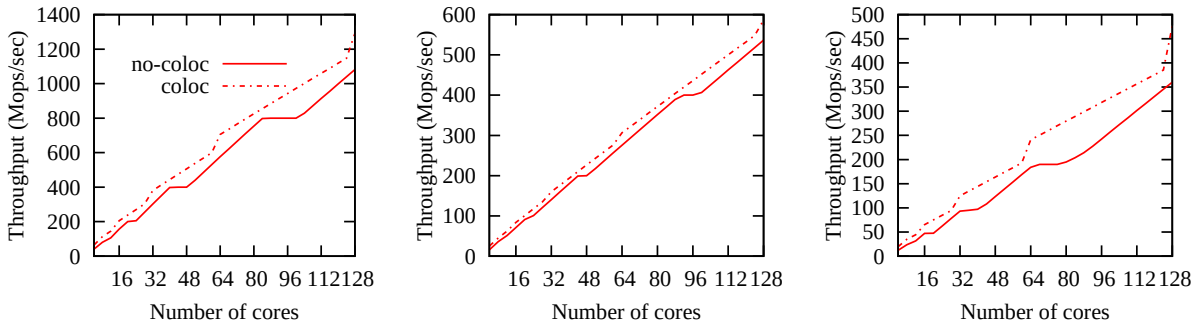
$$\mathscr{L} = \frac{\mathscr{T}^{S_1} \cdot \left(p \cdot T^s_{lup} + (1-p) \cdot T^s_{upd}\right)}{S} \tag{59}$$

Considering the above formula, we obtained the maximum achievable throughput of PART_SIMPLE with collocating clients and server in Figure 24. Considering all three use cases, the performance improvement is at most 20 percent. Analysis of other algorithms show that their performance improvement by collocating clients and servers does not exceed 20 percent.

5.3.6 Non-uniform load distribution on the servers

In calculating throughput of all algorithms, we assumed that the clients uniformly access the servers. However non-uniform distribution of the keys among servers can affect the maximum obtainable throughput.

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



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

Figure 24: Impact of collocating clients and servers on ideal platform (PART_SIMPLE)

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

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

$$\mathscr{T}^s = \sum_{i=1}^{s} k_i \cdot \frac{1}{p \cdot T^s_{lup} + (1-p) \cdot T^s_{upd}} \tag{60}$$

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

5.4 Discussion

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

Althought the map algorithms designed for shared memory architectures mostly ensure linearizability [16], to assess the effects of weakening the consistency criteria, we also study the case of sequential consistency. Replicated maps are able to exploit sequential consistency by removing the synchronization between lookups and updates. On the contrary partitioned maps are not able to exploit sequential consistency, mainly because sequential consistency is not compositional. Evaluations show that replication still needs the same conditions as with the case of linearizability to outperform partitioning. Study of even weaker consistency criteria [28], using a similar methodology, can complement this study.

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

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

We considered the TILE-Gx, a general purpose message-passing manycore, as the baseline for our evaluations. We believe that our conclusions remain valid on similar architectures since: (i) TILE-Gx provides efficient inter-core communication; (ii) using our model we could consider cases where broadcast operations and asynchronous receives are very efficient. Still, using our model, one can directly do a comparison on other architectures. One exception is the architectures with one-sided communication primitives, *e.g.* Intel SCC [17]. The main reason is that inter-core communication in these architectures involves some synchronization costs [21] which are not included in our model.

6 Related Work

This chapter uses performance modeling to compare different algorithms. A few recent studies have proposed performance models for other manycore architectures [21, 24]. Our approach is similar to the one used in these papers. They all cover the same communication scenarios as the LogP model [11] (or its extensions) that is commonly used in message-passing systems. The main difference is that the underlying communication system considered in these studies are different from the one of this chapter: [21] models RMA-based communication and targets the Intel SCC processor; [24] models point-to-point communication on top of cache-coherent shared memory and targets the Intel Xeon Phi processor.

The implementation of scalable data structure in message-passing manycore is an important research topic for message-passing-based operating systems[7, 29, 15]. Barrelfish operating system [7] applies a model, where they structure the operating system as a distributed system of cores, communicating with each other using message passing. They view the state as replicated instead of shared. Hence any potentially shared data structure is considered as if it is a local replica. Consistency among the replicas is maintained by exchanging explicit messages. Their claim to improve scalability by applying replication is based on reducing the traffic on the interconnect, memory contention, synchronization overhead and access latencies. On the other hand, use of client-server approach to provide shared state on chip level, is on the rise. Fos [29] operating system applies a model, where the operating system is factored into function specific services, where each service is provided by a set of cores, so called fleets. Cores communicate with fleets using only messages. Fleets behave similar to Internet servers, which allowed them to scale up to millions of machines, but instead of web pages they provide traditional kernel operations and data structures. Fleets can internally apply different techniques, *e.g.* partitioning, to improve their performance. As an interesting use-case, the implementation of a naming service for the FOS operating system has been studied in [8]. The naming service is based on a hash map which is made scalable using replication. The replication algorithm used is this study is similar to REP_2PC but is not compared to other approaches. Partitioning and replication were both originally proposed as a mean to scale the operating system in the Tornado project [15]. The Tornado

project targets NUMA machines where remote memory accesses are an order of magnitude more costly than local accesses. Since Tornado was designed for shared-memory processors, message-passing was emulated in software with a high cost for software-based multicast operations. We compared partitioning and replication in the context of modern message-passing manycore chips, which provide completely different trade-offs regarding communication performance compared to [15].

Optimization of in-memory key-value stores for manycore is an area where our results could be applied [9, 20]. The authors of [9] and [20] both propose a partitioning approach similar to the PART_SIMPLE algorithm. The solution proposed in [20] is based on message-passing emulated on top of shared memory whereas [9] takes advantage of hardware message-passing provided by Tilera. This chapter complements these studies by providing a comparison between partitioning and replication.

7 Conclusion

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

References

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

[2] Kalray. www.kalray.eu.

[3] Memcached. www.memcached.org.

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

[5] Tilera. www.tilera.com.

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

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, 2009.

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

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

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

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

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

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

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

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

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

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

[18] Chuck Lever. Linux kernel hash table behavior: analysis and improvements. Technical Report TR 00-1, University of Michigan, 2000.

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

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

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

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

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

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

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

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

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

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

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

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