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

Omid Shahmirzadi, Thomas Ropars, and André Schiper

Ecole Polytechnique Fédérale de Lausanne (EPFL),
Lausanne, 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.

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 [22, 13]. 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 [10, 21, 11]. 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 [5]. 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 [9], 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

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 536–547, 2014.

locking, optimistic synchronization and lazy synchronization [9]. In the case of message-passing systems, partitioning and replication are the two main approaches to improve the throughput of concurrent data structures [7]. 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 the data structure in its private memory. Both have been considered in recent work on message-passing manycores [2, 24, 4], 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 [2, 24] to key-value stores [4]. 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 [4]. Because a large majority of operations are usually lookup operations [1], 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. 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 a 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 2 specifies the underlying assumptions and goal of the study. Section 3 introduces the algorithms. Section 4 presents the 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.

2 Assumptions and Goal

The study assumes a fault-free manycore architecture where a large number of threads, each pinned to a single-threaded core during its lifetime, communicate through an on-chip network using the following operations: $send(m, i)$ sends message m to thread i; $bcast(m)$ sends m to all threads; $mcast(m, list)$ sends m to all threads in the $list$; $rcv(m)$ blocks until message m can be received. A thread can be interrupted to deliver a new message m upon its receipt, which is denoted as $arcv(m)$. Communication channels are asynchronous and FIFO.

The study considers the most general consistency criteria, linearizability, and compares the maximum achievable throughput of different linearizable map implementations. A map is a set of items indexed by unique keys that provides *lookup* and *update* operations: $lookup(key)$ returns the value indexed by key ($null$ if no value is associated with key); $update(key, val)$ updates the value indexed by key to val (deleting a key can be done using $update(key, null)$).

3 Algorithms

The two basic techniques to implement scalable concurrent maps on message-passing manycores are partitioning and replication. For each technique, we consider a few algorithms which are representative of the design space. Algorithmic details and correctness proofs can be found in our technical report [20].

3.1 Partitioning

Partitioning a map among a set of server threads can parallelize accesses to different map items. We study two algorithms based on partitioning. In the first algorithm, PART_SIMPLE, a map is partitioned among a set of s servers, *i.e.* item $< key, val >$ is located on server $key \ mod \ s$. A client accesses the corresponding server upon executing a map operation on a key. In the second algorithm, PART_CACHING, recently accessed items are cached on client side. Cached values need to be invalidated if they are updated by other clients. To ensure linearizability, after multicasting an invalidation message, the server waits to receive the acknowledgement from all invalidated clients to finish the update.

3.2 Replication

Replicating a map on each client thread can localize accesses to map items during lookup operations. Unlike in large scale distributed systems, in message-passing manycores locating a replica close to a set of clients is not that beneficial. Even accessing a map replica located in a neighboring core is much more expensive than accessing it locally, since the main access cost is the network cycles used to pack and unpack the payload rather than traversing the hops. In this case replication and partitioning have similar lookup costs, while the former needs expensive updates to ensure consistency. Therefore we only consider the case where each client has its own local map replica.

In replication algorithms, clients deliver updates upon receiving inter-core interrupts. An alternative is to buffer updates and apply them before executing the next map operation. We eliminate this option due to the need for potentially large network buffers, which is not the case in current architectures [21]. To ensure linearizability the following conditions are necessary with respect to each key: (i) updates should be totally ordered; (ii) lookups should be synchronized with updates. We address each condition before describing the algorithms.

Atomic broadcast can be used for total ordering of updates. Among the five classes of atomic broadcast protocols mentioned in [6], we select the one based on a fixed sequencer. In a fixed sequencer algorithm, a sequencer server is in charge of assigning sequence numbers to updates. Three reasons motivate this choice: (a) it needs only one broadcast; (b) updates on different keys can propagate in parallel (by partitioning the sequencing service among multiple sequencer servers, if using a single sequencer can become the bottleneck); (c) replicas can issue requests independently of each other. Other classes lack some of these properties, and so, would provide much lower throughput. Alternatively atomic commitment, *e.g.* two-phase commit, can be used for total ordering of updates. Atomic commit ensures that only one update is applying in the system at a time. This can circumvent the need for dedicating sequencer threads. Therefore we also consider a variant of two-phase commit for total ordering of updates.

Executing lookups without synchronization can violate linearizability, as illustrated by Figure 1, and must be avoided. In this scenario, client c issues an operation *update(key,newval)*, which is applied on the map replicas on c' and c'' at time t_1 and t_2 respectively. If lookups can return immediately with no synchronizing, linearizability can be violated: *lookup(key)* on c_1 returns the new value while, at a later time, the same operation on c_2 returns the old value.

We describe three algorithms satisfying the two mentioned conditions: two based on atomic broadcast and one based on atomic commit. For simplicity we describe the algorithms from the perspective of only one key. We partition the sequencer service among s sequencer servers, each responsible for a subset of keys. In the first algorithm, Rep_remote, clients atomically broadcast their updates and return. Lookups need to contact the corresponding sequencer to know the sequence number sn of the last issued update. Lookups can return only when the update with sequence number sn has been delivered. In the second algorithm, Rep_local, lookups do not need any remote communication to synchronize with updates. This makes updates more complex: After atomic broadcast of an update, the source waits until all other clients acknowledge delivery of this update before broadcasting a final acknowledgement and terminating the operation. Lookups issued after delivering an update should wait until the final acknowledgement is delivered in order to return. In the third algorithm, Rep_2pc, a client, before issuing an update, requests from all other clients whether they are applying a conflicting update or not. If no client is applying a conflicting update, it broadcasts the new update and waits to receive an acknowledgement from all to terminate the operation. Otherwise it aborts its own update. Lookups apply a similar technique as in Rep_local to synchronize with updates.

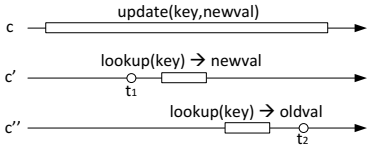


Fig. 1. Non-linearizable execution with a replicated map

4 Performance Modeling

We model the throughput of our map algorithms on message-passing manycores to be able to compare them independently of architecture-specific features and to help manycore programmers to decide about their implementation choice on different platforms. In this section we describe the modeling methodology and we validate it using an existing manycore architecture.

4.1 Methodology

To model the throughput of our algorithms, we assume threads are divided into c *clients*, which issue map operations, and s *servers*, which execute map related code[1]. Client and server threads are located in different cores. Keys are distributed evenly among the servers and are accessed uniformly by the clients.

In manycore architectures with highly efficient NoCs, cores are the main performance bottleneck. We define the following computation parameters to express throughput of our algorithms. We consider a generic map implementation defined by three parameters o_{pre}, o_{lup} and o_{upd}: o_{pre} is the computation time on the client before accessing the map, *e.g.* executing a hash function if the map is implemented using a hash table; o_{lup} and o_{upd} are the computation times corresponding to accessing the underlying data structure during a *lookup* and an *update* respectively. In a configuration with multiple servers, o_{sel} stands for the server selection overhead on the clients. We also associate an overhead with each of the communication primitives introduced in Section 2. Moreover, we introduce the parameter T_{rtt}, to represent round-trip times. More precisely, $T_{rtt}(send\_op, rcv\_op)$ is the round-trip time for the initial message sent with the *send_op* operation (*i.e. send*, *bcast* or *mcast*) and received with the *rcv_op* operation (*i.e. rcv* or *arcv*). The answer is always sent back using *send* and received using *rcv*. If the round trip is initiated with *bcast* or *mcast*, it finishes when the answers from all destinations have been received. We assume that all other computational costs related to the execution of the algorithms, *e.g.* L1 cache accesses, are negligible. Model parameters are summarized in Table 1.

We define T_{lup} and T_{upd}, the number of CPU cycles required to execute a *lookup* and an *update* respectively. For each operation *op*, T_{op} can be divided into the CPU cycles it takes on the client (T^c_{op}) and on the server (T^s_{op}). Considering

[1] Rep_2pc includes no servers.

Table 1. Model parameters and their values ("-" : the same as on TILE-Gx)

parameter	description	TILE-Gx	*int.*	*ideal*
c	number of clients			
s	number of servers			
p	probability of a *lookup* operation			
o_{pre}	computation before a map access			
o_{lup}	access to the map for a *lookup*			
o_{upd}	access to the map for an *update*			
o_{sel}	server selection overhead	if $s = 2^x$ 17,else 90	-	-
o_{send}	overhead of $send(m)$	$8 + \|m\|$	-	-
o_{bcast}	overhead of $broadcast(m)$	$c \cdot o_{send}$	-	o_{send}
o_{mcast}	overhead of $multicast(m, list)$	$\|list\| \cdot o_{send}$	-	o_{send}
o_{rcv}	overhead of a *synchronous* receive	$2 \cdot \|m\|$	-	-
o_{arcv}	overhead of an *asynchronous* receive	$138 + o_{rcv}$	$4 + o_{rcv}$	$4 + o_{rcv}$
L	average communication latency	16	-	-
$T_{rtt}(send, rcv)$	round-trip time with *send* and *rcv*	$2 \cdot (o_{send} + o_{rcv} + L)$	-	-
$T_{rtt}(send, arcv)$	round-trip time with *send* and *arcv*	$2 \cdot (o_{send} + o_{arcv} + L)$	-	-
$T_{rtt}(bcast, arcv)$	round-trip time with *bcast* and *arcv*	$o_{bcast} + o_{arcv} + o_{send} + o_{rcv} + 2 \cdot L$	-	-
$T_{rtt}(mcast, arcv)$	round-trip time with *mcast* and *arcv*	$o_{mcast} + o_{arcv} + o_{send} + o_{rcv} + 2 \cdot L$	-	-

a load where the probability of having a *lookup* operation is p, the maximum throughput $\mathscr{T}^c$ achievable by clients (and equivalently by the servers) is:

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

Hence, the maximum throughput $\mathscr{T}$ of a map is:

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

As an example we model the throughput of the PART_SIMPLE algorithm. The communication pattern is described in Figure 2. It is similar for a *lookup* and an *update* operation. The only difference is that during an update operation, applying the update on the map can be removed from the critical path of the client. Computing T^s_{op} (where *op* is *upd* or *lup*), T^c_{lup} and T^c_{upd} is as follows:

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

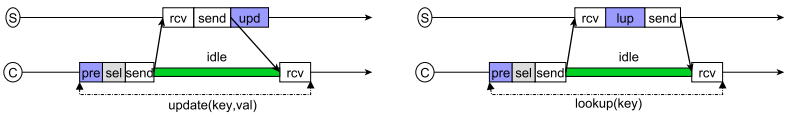
$$T^c_{lup} = o_{pre} + o_{sel} + T_{rtt}(send, rcv) + o_{lup} \tag{4}$$

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

Unlike PART_SIMPLE, modeling the throughput of other algorithms involves some complexity. In replication algorithms, a client can deliver asynchronous messages *for free* during idle periods. This increases the throughput of the clients and alters the general Formula 1. Moreover in PART_CACHING, the probability of hitting the local cache as well as the number of clients which need to be invalidated should be computed. The detailed performance model of the other algorithms can be found in [20].

4.2 Validation

We use a Tilera TILE-Gx8036 processor [21] as a representative of current message-passing manycore architectures to validate our model. It consists of 36



(a) Update (b) Lookup

Fig. 2. PART_SIMPLE algorithm

cores communicating through a mesh interconnect. Cores and mesh operate at the same frequency, 1.2 Ghz. Each core is provided with a 32 KB L1 instruction cache, a 32 KB L1 data cache, a 256 KB L2 cache and four independent FIFO receive buffers where each can contain up to 118 64-bit words. Threads located on different cores can communicate using *send* and *rcv* primitives with no operating system intervention. A *send* puts the data in one of the four receive buffers of the destination and a *rcv* blocks until this data is available. Upon receipt of a message in any of the four buffers, an interrupt can be raised to perform an asynchronous receive. There is no hardware support for collective operations.

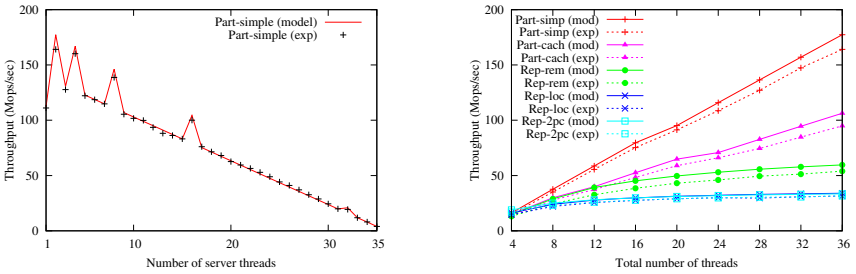
We obtain the model parameters for the TILE-Gx processor through microbenchmarks. Each *send* has a fixed overhead of 8 cycles plus 1 cycle per word. Due to the lack of collective operations, we implement *bcast* and *mcast* as a set of consecutive *send* operations, so their cost is a linear function of o_{send}[2]. Each *rcv* needs 2 cycles to deliver each word from the receive buffers. Each *arcv*, in addition to the cycles needed for receiving messages from the buffers, requires an overhead of 138 cycles to save and retrieve the execution context. We compute the round trip times, as the length of the critical path on the source thread from the first send operation to the last receive operation. Round-trip times take into account the average communication latency L, which involves a fixed packing and unpacking overhead of 10 network cycles plus an average traversal cost of 6 network cycles (1 cycle per hop). Finally o_{sel} is 17 cycles if the number of servers is a power of two, otherwise 90 cycles (*mod* function is implemented using bitwise operations). Table 1 summarizes the TILE-Gx parameters.

To validate our model, we pin each thread to a different core. Clients issue map operations with 90% probability a lookup ($p = 0.9$). Keys are evenly distributed among the servers and are accessed uniformly by the clients. We consider a map implemented using a hash table which fits into the L1 cache of the cores: $o_{lup} = o_{upd} = 0$ cycles[3]. We use the DJB hash function to generate 4 bytes long keys from 36 bytes long strings: $o_{pre} = 156$ cycles. We consider a collision-free scenario. Experiments are run with version 2.6.40.38-MDE-4.1.0.148119 of Tilera's custom Linux kernel, compiled using GCC 4.4.6 with O3 flag.

Figure 3(b) presents the maximum throughput of the five algorithms, obtained through experiments and model, for different total number of threads. Each point in the experimental results is the average throughput of 6 runs, where in each

[2] Effects of such an implementation will be later removed by considering a platform with a hardware-based broadcast service.

[3] We consider L1 to better evaluate the accuracy of our communication model.



(a) PART_SIMPLE with 36 threads (b) Performance of algorithms

Fig. 3. Model validation on TILE-Gx processor (p=0.90)

run every client issues 10000 operations on the map. Keys are distributed among the servers uniformly and clients randomly select the key for the next operation with a uniform distribution. For a given number of threads, the corresponding throughput for each algorithm represents the throughput obtainable from the best possible configuration of clients and servers. For example, the throughput of the simple partitioning algorithm with 36 threads in Figure 3(b) is obtained through the graph shown in Figure 3(a), where the configuration with 2 servers and 34 clients leads to the best throughput (hiccups are due to o_{sel}). These figures show that we manage to model the throughput of the algorithms with good approximation (a maximum of 12% deviation in the case of PART_CACHING). However the throughput obtained through the model is slightly higher than through experiments. This is mainly because in practice other computational costs are involved (*e.g.* operations on the cached variables).

5 Evaluation

Studying algorithms only on the TILE-Gx leads to architecture-specific results. Two limitations of this processor can decrease the performance: (i) asynchronous receives, although relatively efficient compared to existing architectures, are still much more costly than synchronous ones; (ii) there is no efficient broadcast service[4]. These limitation could impair the performance of all replication algorithms and of PART_CACHING and so, could be the reason for the higher throughput of PART_SIMPLE observed in Figure 3(b). Using our model, we define two platforms based on TILE-Gx to avoid the harmful effects of the mentioned limitations. We define an *intermediate* platform where synchronous and asynchronous receives have similar costs. We also define an *ideal* platform, which enhances the *intermediate* platform with broadcast operation in hardware. In this case the cost of *bcast* is equal to the cost of a single *send*. The *ideal* platform provides the best setting for the replication algorithms. These assumptions are realistic. In [18], a

[4] When broadcast is implemented using asynchronous communication, the throughput of the system is independent from the broadcast algorithm [16].

solution with a constant 4 cycles cost of saving and restoring an execution context is presented. Moreover some existing manycores, *e.g.* Kalray [11], provide hardware-based broadcast. Table 1 summarizes parameter values for these platforms. In this section we compare the algorithms performance on these platforms. We also discuss how different consistency, configuration and load assumptions can alter the results (see [20] for detailed discussions).

5.1 Comparison on Different Platforms

To compare the algorithms on different platforms, we apply our analytical model. We consider a map implemented using a hash table as the most popular map implementation. To avoid orthogonal issues, we consider a collision free scenario where the keys are evenly distributed among servers and are accessed uniformly by the clients. To assess different computational costs, we identify three use cases with different hash function costs (depending on its input type) and hash table sizes (small enough to fit in the L2 cache or otherwise in memory). Namely we consider (i) a small hash table with an integer hash function ($o_{pre} = 12$, $o_{op} = 11$); (ii) a small hash table with a string hash function ($o_{pre} = 156$, $o_{op} = 11$); (iii) a big hash table with a string hash function ($o_{pre} = 156$, $o_{op} = 88$). The first two are representative use cases in operating systems [12] while the latter is a representative use case in key-value stores [14][5].

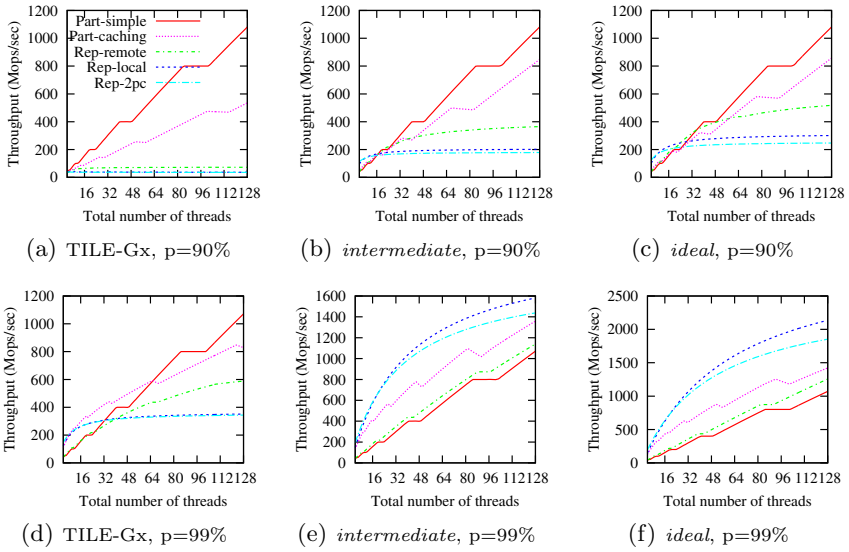
Considering the first use case, we compare the performance of the algorithms on different platforms with 90% and 99% of lookups (see Figure 4). We apply the same methodology as in model validation to obtain throughput graphs. Three main conclusions can be taken from the results. First, with 90% of lookups PART_SIMPLE outperforms other algorithms on all platforms at almost all scales. Second, with 99% of lookups REP_LOCAL outperforms the partitioning algorithms only if asynchronous receives are handled efficiently. Actually on the *ideal* platform the minimum ratio of lookups for replication to outperform partitioning is 98%. Third, having broadcast in hardware does not change the relative performance of the algorithms dramatically (compare Figures 4(e) and 4(f)).

Considering other use cases, the mentioned conclusions remain valid. The only exception is the scenario where the hash table is located in the main memory. In this case even with 99% of lookups, PART_CACHING shows best performance on all platforms. This is due to the fact that replicated maps are not able to leverage the locality if map replicas are not cached.

5.2 Discussion

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

[5] We did not find any use cases for a big hash table applying a cheap hash function.



(a) TILE-Gx, p=90% (b) *intermediate*, p=90% (c) *ideal*, p=90%

(d) TILE-Gx, p=99% (e) *intermediate*, p=99% (f) *ideal*, p=99%

Fig. 4. Analytical performance on the three platforms ($o_{pre} = 12$, $o_{op} = 11$)

to outperform partitioning. Study of even weaker consistency criteria [23], 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 phenomenon 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.

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 [10]. The main reason is that inter-core communication in these architectures involves some synchronization costs [17] which are not included in our model.

6 Related Work

This paper compares different map algorithms using performance modeling. A few recent studies have proposed performance models for other manycore architectures [17, 19]. Our approach is similar to the one used in these papers. The main difference is that the underlying communication system considered in these studies are different from our paper: [17] models RMA-based communication and targets the Intel SCC processor; [19] models point-to-point communication on top of cache-coherent shared memory and targets the Intel Xeon Phi processor.

Implementation of scalable data structures is an important research topic for message-passing-based operating systems [2, 24, 8]. Partitioning and replication were both originally proposed as a mean to scale the operating systems in the Tornado project [8]. 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 [8]. As an interesting use-case, a naming service for the FOS operating system is implemented using a weakly-consistent replicated hash map [3]. The replication algorithm used in this study is a variant of Rep_2pc, but is not compared to other alternatives.

Optimization of in-memory key-value stores for manycores is another area where our results could be used [4, 15]. The authors of [4] and [15] both propose a partitioning approach similar to the Part_simple algorithm. The solution proposed in [15] is based on message-passing emulated on top of shared memory whereas [4] takes advantage of hardware message-passing provided by Tilera. Our paper complements these studies by comparing replication and partitioning.

7 Conclusion

The 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] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE, pp. 53–64 (2012)

[2] Baumann, A., Barham, P., Dagand, P., et al.: The multikernel: a new OS architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 29–44 (2009)

[3] Beckmann, N.: Distributed naming in a factored operating system. Master's thesis, Massachusetts Institute of Technology (2010)

[4] Berezecki, M., Frachtenberg, E., Paleczny, M., Steele, K.: Many-core key-value store. In: Proceedings of the 2011 International Green Computing Conference and Workshops, pp. 1–8 (2011)
[5] Calciu, I., Dice, D., Lev, Y., Luchangco, V., Marathe, V.J., Shavit, N.: NUMA-aware reader-writer locks. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2013)
[6] Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys 36(4), 372–421 (2004)
[7] Devlin, B., Gray, J., Laing, B., Spix, G.: Scalability terminology: Farms, clones, partitions, and packs: Racs and raps. Technical Report MS-TR-99-85, Microsoft Research (1999)
[8] Gamsa, B., Krieger, O., Appavoo, J., Stumm, M.: Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In: The Third Symposium on Operating Systems Design and Implementation, pp. 87–100 (1999)
[9] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2012)
[10] Howard, J., Dighe, S., Hoskote, Y., et al.: A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In: International IEEE Solid-State Circuits Conference Digest of Technical Papers (ISSCC), pp. 108–109 (2010)
[11] Kalray, `http://www.kalray.eu`
[12] Lever, C.: Linux kernel hash table behavior: analysis and improvements. Technical Report TR 00-1, University of Michigan (2000)
[13] Martin, M.M.K., Hill, M.D., Sorin, D.J.: Why on-chip cache coherence is here to stay. Communications of the ACM 55(7), 78–89 (2012)
[14] Memcached, `http://www.memcached.org`
[15] Metreveli, Z., Zeldovich, N., Kaashoek, M.F.: Cphash: A cache-partitioned hash table. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 319–320 (2012)
[16] Petrović, D., Shahmirzadi, O., Ropars, T., Schiper, A.: Asynchronous broadcast on the Intel SCC using interrupts. In: Proceedings of the 6th Many-core Applications Research Community Symposium, pp. 24–29 (2012)
[17] Petrović, D., Shahmirzadi, O., Ropars, T., Schiper, A.: High-performance RMA-based broadcast on the Intel SCC. In: Proceedinbgs of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 121–130 (2012)
[18] Rafla, N., Gauba, D.: Hardware implementation of context switching for hard real-time operating systems. In: 54th IEEE International Midwest Symposium on Circuits and Systems (2011)
[19] Ramos, S., Hoefler, T.: 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, pp. 97–108 (2013)
[20] Shahmirzadi, O., Ropars, T., Schiper, A.: High-throughput maps for message-passing manycore architectures: partitioning versus replication. Technical Report 196582, EPFL (2014)
[21] Tilera, `http://www.tilera.com`
[22] Torrellas, J.: Architectures for Extreme-Scale Computing. IEEE Computer 42(11), 28–35 (2009)
[23] Vogels, W.: Eventually consistent. Communications of the ACM 52(1), 40–44 (2009)
[24] Wentzlaff, D., Agarwal, A.: Factored operating systems (FOS): the case for a scalable operating system for multicores. ACM SIGOPS Operating Systems Review 43(2), 76–85 (2009)