ZIMP: Efficient Inter-core Communications on Manycore Machines

Pierre-Louis Aublin, Grenoble University
Sonia Ben Mokhtar, CNRS - LIRIS
Gilles Muller, INRIA
Vivien Quéma, CNRS - LIG

***Abstract*—Modern computers have an increasing number of cores and, as exemplified by the recent Barrelfish operating system, the software they execute increasingly resembles distributed, message-passing systems. To support this evolution, there is a need for very efficient inter-core communication mechanisms. Current operating systems provide various inter-core communication mechanisms, but it is not clear yet how they behave on manycore processors. In this report, we study seven mechanisms, that are considered state-of-the-art. We show that these mechanisms have two main drawbacks that limit their efficiency: they perform costly memory copy operations and they do not provide efficient support for one-to-many communications. We do thus propose ZIMP, a new inter-core communication mechanism that implements zero-copy inter-core message communications and that efficiently handles one-to-many communications. We evaluate ZIMP on three manycore machines, having respectively 8, 16 and 24 cores, using both micro- and macro-benchmarks (a consensus and a checkpointing protocol). Our evaluation shows that ZIMP consistently improves the performance of existing mechanisms, by up to one order of magnitude.**

I. Introduction

Modern computers have an increasing number of cores and as envisioned by the designers of the recent Barrelfish operating system [1], the software they run increasingly resemble distributed, message passing systems. Moreover, it is now admitted that one of the major challenges of the next decade for system researchers, will be to enable modern manycore systems to tolerate both software [2], [3] and hardware failures [4]. To tolerate failures in such message-passing manycore systems, agreement [5] and broadcast protocols [6], [7] can be used for maintaining a consistent state across cores. Furthermore, checkpointing algorithms [8], [9], [10], [11], [12], [13], [14], [15], [16] can provide a mean to resist failures of software distributed across cores. In this area, Yabandeh and Guerraoui have pioneered fault tolerance for manycore architectures with PaxosInside, an adaptation of the Paxos protocol for manycore machines [17].

In order to support efficient execution of such protocols, there is a need for very efficient inter-core communication mechanisms enabling both one-to-one and one-to-many communications. Current operating systems provide various inter-core communication mechanisms. However it is not clear yet how these mechanisms behave on manycore processors, especially when messages are intended to many recipient cores.

In this report, we study seven communication mechanisms, which are considered state-of-the-art. More precisely, we study TCP, UDP and Unix domain sockets, pipes, IPC and POSIX message queues. These mechanisms are supported by traditional operating systems such as Linux. We also study Barrelfish message passing, the inter-process communication mechanism of Barrelfish OS [1]. We show that all these mechanisms have two main drawbacks that limit their efficiency. Firstly, they perform costly memory copy operations. Indeed, to send a message, multiple copies of the latter have to be created in memory. Secondly, they do not provide efficient support for one-to-many communications. Indeed, to send a message to N receivers, N calls to the send primitive of the communication mechanism need to be performed.

We propose ZIMP (Zero-copy Inter-core Message Passing), a new, efficient, inter-core communication mechanism. More particularly, ZIMP provides a zero-copy send primitive by allocating messages directly in a shared area of memory. It also efficiently handles one-to-many communications by allowing a message to be sent once and read multiple times.

We evaluate ZIMP on three manycore machines, having respectively 8, 16 and 24 cores, using both micro- and macro-benchmarks. Our macro-benchmarks consist of two real world applications, namely PaxosInside [17] and a checkpointing protocol [8]. The performance evaluation shows that for PaxosInside, ZIMP improves the throughput of the best state-of-the-art mechanism by up to 473%. It also shows that for the checkpointing protocol, ZIMP reduces the latency of the best state-of-the-art mechanism by up to 58%.

The remaining of the report is organized as follows: In Section II, we describe the existing communication mechanisms that can be used for inter-core commu-

nications. In Section III, we present ZIMP, our efficient communication mechanism devised for manycore machines. We then evaluate the performance of the state-of-the-art mechanisms and of ZIMP in Section IV. We finally discuss related work in Section V, before concluding the report in Section VI.

II. Background

In this section, we start by reviewing seven state-of-the-art inter-core communication mechanisms. We then analyze the drawbacks of all presented mechanisms and discuss the need for a new efficient inter-core communication mechanism.

A. Inter-core communication mechanisms

We review seven inter-core communication mechanisms. Six of them are provided by traditional Linux/Unix operating systems. Among these mechanisms, Unix domain sockets, Pipes, IPC message queues and POSIX message queues have been devised for the communication between processes residing on the same host. We also review TCP and UDP sockets that have been designed for communication over an IP network but that are often used for communication on the same host. We finally present Barrelfish message passing, a communication mechanism that has been specifically devised for manycore machines.

1) Unix domain sockets: Unix Domain sockets is a communication mechanism specifically designed for communications between processes residing on the same host. In order to use Unix domain sockets, the sender and the receiver both create a socket, i.e., a data structure with a pointer to a linked list of datagrams. To send a message, a sender creates a buffer and uses the `sendto()` system call. Then, the kernel copies the buffer from the user space to the kernel space and adds it to the list. To receive messages, a receiver uses the `recvfrom()` system call. Finally, the kernel copies the last entry of the list from the kernel space to the user space.

2) TCP and UDP sockets: TCP and UDP sockets are two mechanisms that have been designed to allow processes residing on distinct machines to communicate over an IP network. Nevertheless, these mechanisms can also be used by processes residing on the same host to communicate by using the loopback interface. From a design point of view, UDP and TCP sockets share similarities with Unix domain sockets: they use the same system calls and messages have to be copied from the user to the kernel space, and vice versa. There are nevertheless some differences: UDP and TCP sockets require large messages to be fragmented into smaller packets (the maximum packet size is 16kB using the loopback interface). Moreover, to send a set of packets TCP and UDP sockets first places them on the sender's socket list. Packets are then pushed by the kernel on the loopback interface. Upon an interruption, the kernel receives the packets and places them on the receiver's socket list. Note that TCP and UDP do not offer the same interface and do not provide the same guarantees. More precisely, TCP is loss-less and stream-oriented while UDP can lose messages (if the receiver socket list is full) and is datagram oriented. Finally, UDP datagrams have a maximum size of 65kB. Hence, to send larger messages using UDP, the application needs to fragment them into chunks of 65kB or less, resulting in additional system calls.

3) Pipes: A pipe is a data structure stored in kernel space and containing a circular list of small buffer entries. Each entry has a size of 4kB, which corresponds to the size of a memory page. To send a message using a pipe, a process uses the `write()` system call. Then, the kernel splits the message in 4kB chunks and copies the chunks from the user space to the pipe's circular list. To receive messages from a pipe, a receiver uses the `read()` system call. This system call copies the content of the pipe to a user space buffer.

4) IPC and POSIX message queues: IPC and POSIX message queues (noted IPC MQ and POSIX MQ in the following, respectively) are queues of messages residing in kernel space. Although the design of both mechanisms is similar, they exhibit some differences. For instance, an IPC MQ can be used between several senders and several receivers, while a POSIX MQ can be used between multiple senders but only one receiver.

To send a message, a sender uses the `msgsnd()` system call[1]. Then, the kernel copies the message from the user space to the kernel space and adds the message to the queue. To receive a message, a receiver uses the `msgrcv()` system call. When a message is present, it is copied from the kernel space to the user space.

5) Barrelfish message passing: Barrelfish message passing (noted Barrelfish MP) is a user-level point-to-point communication mechanism that has been devised for the Barrelfish operating system[2]. For each sender-receiver pair, there are two circular buffers of messages that act as unidirectional communication channels. The

[1]We describe the interface used in IPC message queues. The interface used in POSIX message queues is similar.

[2]We use the publicly available Barrelfish source code release of march 2011, available at *http://www.barrelfish.org*.

size of these channels, S, is defined when they are created. Messages have a fixed size, which is a multiple of the size of a cache line, and they are aligned on a cache line. This alignment prevents the structures to use more cache lines than necessary. As a result the number of memory accesses is reduced. For instance, accessing the totality of a misaligned structure of the size of a cache line requires the processor to fetch two cache lines from the main memory. If this structure is properly aligned, it only requires the processor to fetch one cache line. Each message is composed of a header and of a content. The header contains the sequence number of the message and a notification which informs the receiver that the message can be read. Barrelfish MP requires the receiver to acknowledge periodically the last message it has received. More precisely, the sender can not send more than S messages without having received one acknowledgement. The acknowledgement is a message, sent from the receiver to the sender, which contains the sequence number of the last message read by the receiver.

When a sender wants to send a message (located in a private buffer), it first checks if there is enough room in the channel for a new message. Specifically, it checks if it has sent less than S messages since the reception of the last acknowledgement. If it is not the case, the sender waits for an acknowledgement from the receiver. Otherwise, it copies the message from its private buffer to the communication channel. Then, it writes the sequence number and the notification in the header of the message.

The receiver knows the location of the next entry to be read in the communication channel. To receive a message, it polls the header of the message at that location, waiting for the notification to be written. As soon as the notification is written by the sender, the receiver copies the message from the communication channel to its own buffer. This copy requires several memory accesses, i.e., one memory access for each chunk of the size of a cache line. Finally, the receiver saves the sequence number and, if necessary, sends an acknowledgement to the sender.

B. Discussion: Do we need a new communication mechanism?

Table I summarizes the key points of state-of-the-art mechanisms presented in this section. The first line of the table shows that all presented mechanisms require N message copies to send a message to N receivers. This is not efficient since the memory is accessible by all the cores and a single copy could be shared by the N receivers. The second line of the table shows that for receiving a message, the latter is copied N times by the N receivers (once for each receiver). The third and fourth lines of the table show the number of system calls required by existing mechanisms to send/receive a message to/by N receivers (respectively). We observe that all kernel-level mechanisms require to perform at least N system calls when sending/receiving a message to/by N receivers. More precisely, they all require exactly N system calls apart from UDP, which may require more system calls if the message to be sent/received is greater than 65kB. Indeed, the maximal UDP datagram size is 65kB. Consequently a message greater than this size must be sent/received in several chunks of at most 65kB. Note that system calls incur a considerable overhead due to process context switch. Indeed, using a micro-benchmark, we found that it costs 14 times more cycles to perform a system call than to invoke a standard function[3]. As Barrelfish MP is a user-level mechanism, it does not require any system call for sending and receiving messages.

From the above, we observe that existing communication mechanisms suffer from several drawbacks, which may impact their performance. Summarizing, when sending a message to many receivers, all mechanisms create unnecessary copies of the same message in memory. Furthermore, all kernel-level mechanisms perform a large number of (costly) system calls for sending the same message to a set of receivers. We conclude that none of the existing mechanism is suitable for one-to-many communications. We thus present in the next section ZIMP, a new inter-core communication mechanism optimized for one-to-many communications involving processes residing on the same host.

III. ZIMP

We present in this section ZIMP, our new inter-core communication mechanism. We start by presenting its design in Section III-A and discuss a set of performance optimizations in Section III-B.

A. Design of ZIMP

ZIMP (*Zero-copy Inter-core Message Passing*) is a user-level communication mechanism dedicated to communications between processes residing on the same host. The key feature of this mechanism is that there is no copy when sending a message. Indeed, the message is directly allocated in the communication channel.

[3]This result has been obtained using the same hardware and software configurations described in Section IV.

	Unix domain sockets	TCP sockets	UDP sockets	Pipes	IPC MQ	POSIX MQ	Barrelfish MP	ZIMP
Message copies performed for sending 1 message (N receivers)	N	N	N	N	N	N	N	0
Message copies performed for receiving 1 message (N receivers)	N	N	N	N	N	N	N	N
System calls performed for sending 1 message (N receivers)	N	N	$N*\lceil \frac{M}{65507} \rceil$	N	N	N	0	0
System calls performed for receiving 1 message (N receiver)	N	N	$N*\lceil \frac{M}{65507} \rceil$	N	N	N	0	0

Table I: Summary of the key points of state-of-the-art mechanisms plus ZIMP, when sending a message of size M Bytes to N receivers.

Moreover, it provides an efficient one-to-many communication primitive: there are multiple receive for a single send.

ZIMP allows creating *communication channels*. Each communication channel can be used by several senders and is associated to a set of receivers. This means that each message sent using a channel will be received by all processes belonging to the receivers set associated to this channel. Consequently, for each distinct group of receivers, a separate communication channel must be created.

In ZIMP, each communication channel is implemented using a circular buffer. This buffer stores a set of messages. The number of messages that can be stored in the buffer and the maximal size of each message are specified as a parameter of the communication channel creation primitive. Each channel uses a variable, `next_send_entry`, which indicates to the senders the next entry in the circular buffer where to write a message. This variable is protected by a lock, which allows multiple senders to concurrently access the channel. Moreover, each buffer entry is associated with a bitmap. The bit at position i in a bitmap is set to one when a message is written by a sender and is reset when the message is read by the i^{th} process. Bitmaps are atomically updated by both the senders and the receivers. Finally, each channel uses a `next_read` array to store the index of the next message that will be read by each process in the receivers set. Since the i^{th} entry in the `next_read` array is only updated by the i^{th} process in the receivers set, accesses to the `next_read` array do not need to be synchronized.

Figure 1 depicts the steps performed to send and receive a message using ZIMP. To send a message, a sender does not need to allocate memory. It first gets the address of the next available entry in the channel, i.e., `next_send_entry`. It then updates this variable by setting it to the next entry in the buffer in order to allow other potential senders to simultaneously use the channel. Then, the sender waits for the entry to become empty by polling the corresponding bitmap. Specifically, it waits until all the readers have read the previous message at this entry (if any), i.e. it waits for all the bits of the bitmap to become 0. When it is the case, the sender writes the content of the message in the entry. It then "sends" the message, which consists of updating the bitmap associated with the buffer entry by setting each bit to 1. To read a message, a receiver r first gets `next_read[r]`. If the bitmap associated to this entry indicates that there is a message to read (i.e., if r's bit in the bitmap is set to 1), then the receiver performs the three following steps. Firstly, it updates the value of `next_read[r]`. Secondly, it copies the content of the entry. Finally, it updates the bitmap. Otherwise, if there is no message to read, then it waits for a message at this entry by polling the bitmap.

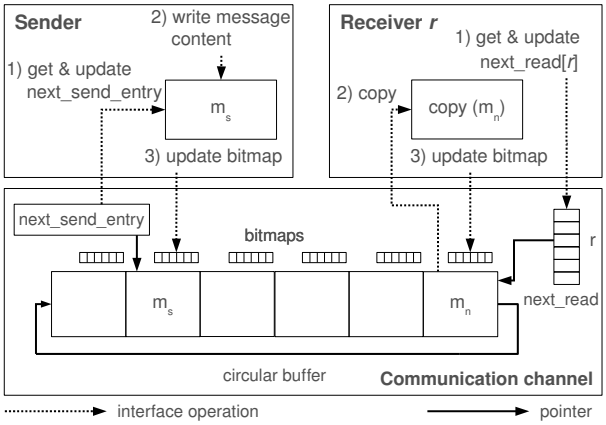

Figure 1: Zero-copy Inter-core Message Passing.

B. Performance optimizations

In order to increase the efficiency of ZIMP, we have implemented a number of cache-related optimizations. Firstly, ZIMP structures are aligned on a cache line, as in the case of Barrelfish MP, which improves the cache usage. Secondly, a full cache line is reserved for each bitmap. This reduces the cost of polling when the

receiver is waiting for a message. More precisely, the first time the receiver accesses the bitmap, it is fetched from the main memory to its cache. Until the cache line is invalidated, which occurs when the bitmap has been modified, subsequent accesses do not require an access to the main memory. Finally, ZIMP structures are padded, i.e., extra bytes are added at the end of the structures, so as the size of each structure becomes a multiple of a cache line size [18]. The advantage of padding is that it prevents false sharing. False sharing occurs when several processes access different unrelated data that fit in the same cache line. Although they do not access the same data, the cache coherency mechanism will invalid the entire line each time a single bit is modified. In ZIMP, structure padding is used to access the `next_read` table efficiently. Indeed, this array contains, for each receiver, the address of the next entry to read, which has a size of 4B. Without structure padding, multiple elements of the array would fit in the same cache line, which would trigger unnecessary cache misses each time one of the elements is modified.

IV. Performance evaluation

In this section, we present a performance analysis of ZIMP. We compare its performance to that achieved by the state-of-the-art mechanisms presented in Section II. We start by a description of the hardware and software settings we used. Then, we present an evaluation based on a micro-benchmark. Finally, we evaluate the mechanisms using two macro-benchmarks: a consensus and a checkpointing protocols.

A. Hardware and software settings

We ran our experiments on three different manycore machines, each one running Linux kernel version 2.6.38 and designed with 64B cache lines. Details regarding the three machines are provided below:

- 8-core machine: This is a Dell Precision WorkStation T7400 that hosts two quad-core Intel Xeon E5410 processors clocked at 2.33GHz and 8GB of RAM. Each core has a private L1 cache of 32kB. Moreover, each pair of cores share a 6MB L2 cache.
- 16-core machine: This is a Dell PowerEdge R905 machine that hosts four quad-core AMD Opteron 8380 processors and 32GB of RAM. Each core is clocked at 2.5 GHz, has private L1 and L2 caches of 64 and 512 kB, respectively, and shares a 6MB L3 cache with the three other cores hosted on the same processor.
- 24-core machine: This is a HP Proliant DL165 G7 machine that hosts two AMD Opteron 6164HE processors clocked at 1.7GHz and 48GB of RAM. Each processor contains two sets of six cores that share a L3 cache of 6MB. Moreover, each core has private L1 and L2 caches of 64kB and 512kB, respectively.

B. Micro-benchmark

We have developed a simple micro-benchmark that works as follows: a sender sends messages to a set of receivers, each one running on a dedicated core. Each experiment lasts for two minutes. We measure the throughput at which receivers deliver messages. We present the results obtained on the 24-core machine: results on other machines are consistent. Each value in the presented graph is an average computed over three runs for which the standard deviation was very low. We use the micro-benchmark to study (1) the impact of message size on performance, (2) the impact of the number of receivers on performance, and (3) the impact of the hardware on performance.

1) Impact of the message size: we compare in this experiment the throughput of the various communication mechanisms obtained with the following message sizes: 1B, 64B, 128B, 512B, 1kB, 4kB, 10kB, 100kB and 1MB. We present results obtained with two extreme configurations: 1 receiver (Figure 2) and 23 receivers (Figure 3). Results for other numbers of receivers are consistent and reported in Appendix A.

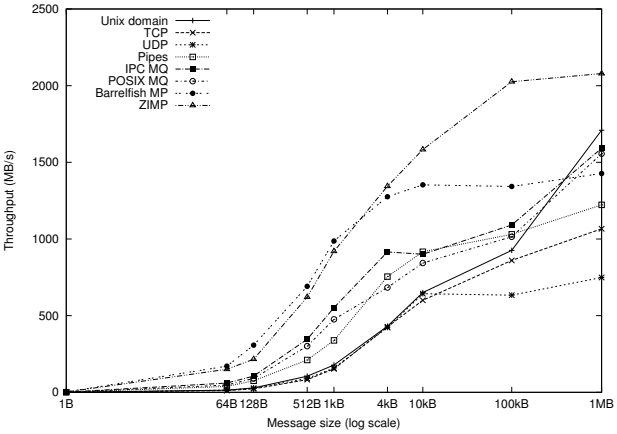

Figure 2: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (1 receiver).

The first observation we can make is that in all configurations depicted from Figure 2 to Figure 3, ZIMP and Barrelfish MP outperform other mechanisms (except with 1MB messages, where Barrelfish MP is

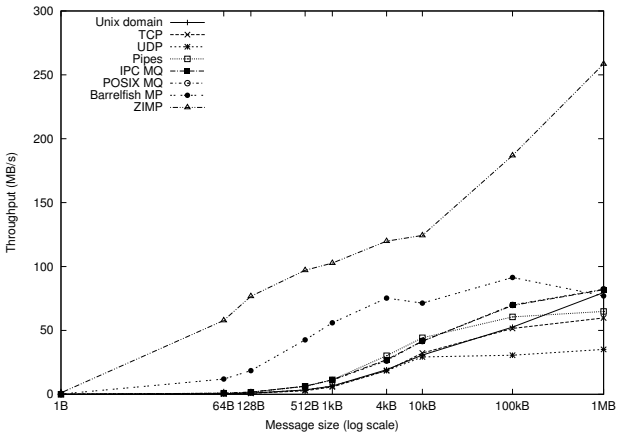

Figure 3: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (23 receivers).

outperformed by other mechanisms). Regarding the relative performance of ZIMP and Barrelfish MP, Figure 2 shows that with one receiver, Barrelfish achieves slightly better performance than ZIMP (+15% on average) for message sizes below 1kB. For larger messages, ZIMP achieves better performance and the performance difference increases with the message size (+5% with 4kB messages and +46% with 1MB messages). Figure 3 shows that with 23 receivers, ZIMP consistently and drastically outperforms Barrelfish MP (+515% with 64B messages and +236% with 1MB messages).

We study and explain below the performance achieved by the various mechanisms.

• **TCP, UDP and Unix domain sockets:** they achieve lower performance than other mechanisms. Overall, we observe that Unix domain sockets achieve slightly better performance than the two other mechanisms. Additionally, we observe that, regardless the number of receivers, TCP sockets are between 3% and 22% less efficient than UDP sockets for messages below 1kB. We attribute this behavior to the overhead induced by the various mechanisms used in TCP to ensure reliable, FIFO delivery of messages (acknowledgements, flow control, etc). With larger messages (i.e. above 65kB), we observe that TCP sockets start outperforming UDP sockets (from 36% to 64%). This is due to UDP which requires large messages to be fragmented into smaller messages by the applications (recall that the maximum datagram size is 65kB). This induces additional system calls.

• **Pipes, IPC and POSIX MQs:** they exhibit comparable performance in most configurations. For example, the performance difference between these three mechanisms with more than 2 receivers and messages less than 1MB ranges between 0.2% and 20%, for a mean of 6%. Nevertheless we can observe some differences. For instance, when there is only 1 receiver, IPC MQ achieves better performance than POSIX MQ (between 2% and 34%). This is due to the fact that POSIX MQ performs a file lookup to find the POSIX queue corresponding to a given identifier, which induces some overhead.

• **Barrelfish MP:** among state-of-the-art mechanisms, this is the most efficient one. The gap between Barrelfish MP and other mechanisms is larger for small messages than for large messages, regardless the number of receivers. For instance, with 1 receiver, Barrelfish MP outperforms TCP sockets by 1400% for 1B messages and by 32% for 1MB messages. Specifically, we noticed that Barrelfish MP reaches its limit with messages of 4kB. This performance difference between small and large messages is due to the fact that Barrelfish MP is limited by the cache size. Indeed, there are 66 times more L2 cache misses per bytes for messages of 1MB compared to messages of 64B. This results in a lower number of instructions per cycles: Barrelfish MP spends more time waiting for data.

• **ZIMP:** it outperforms all state-of-the-art mechanisms in almost all configurations. The only exception is for message sizes below 1kB when there is only one receiver. In that case, Barrelfish MP slightly outperforms ZIMP. ZIMP is less efficient in this configuration because its design induces a small overhead that is not necessary for configurations involving only one sender and one receiver. Specifically, the sender needs to acquire a lock and the bitmap is modified atomically. Note that although it is not clearly visible in Figure 3, the difference between the throughput of ZIMP and other protocols is very large even for messages of 1B as further discussed in the following section.

2) Impact of the number of receivers: the goal of this experiment is to study the impact of the number of receivers on the performance of the various communication mechanisms presented in this report. We measure the throughput improvement of ZIMP over other mechanisms when increasing the number of receivers from 1 to 23. We study two message sizes that correspond to the extreme sizes used in the previous experiment: 1B and

1MB. We depict performance results in Figure 4 (1B messages) and in Figure 5 (1MB messages). Note that we use a logarithmic scale for the Y axis. Results for intermediary message sizes are consistent and presented in Appendix B.

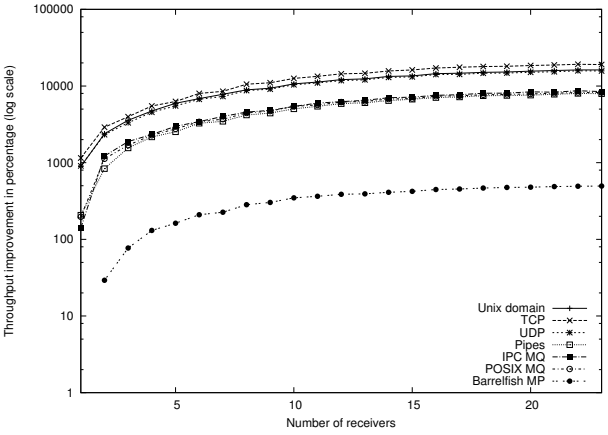

Figure 4: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (1B messages).

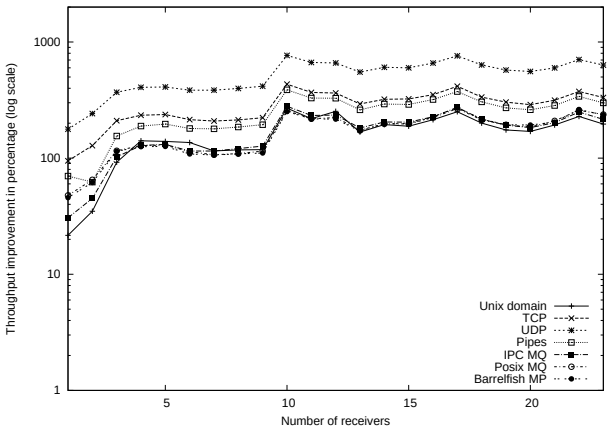

Figure 5: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (1MB messages).

We observe that for both 1B and 1MB messages, the throughput improvement of ZIMP over other mechanisms is impacted by the number of receivers: the higher the number of receivers, the larger the performance improvement. For instance, with 1B messages, the improvement of ZIMP over TCP sockets spans from 1150% (with 1 receiver) to 19000% (with 23 receivers). Similarly, with 1MB messages, the improvement of ZIMP over TCP sockets spans from 94% (with 1 receiver) to 333% (with 23 receivers). ZIMP has been devised specifically for efficient one-to-many communication on a manycore machine. Specifically, a message is sent once and received multiple times and the cache is used efficiently. This is not the case for state-of-the-art mechanisms, which are heavily impacted by the number of receivers. Consequently, the improvement of ZIMP increases as there are more receivers.

3) Impact of the hardware on performance: In this section we are interested on the impact of the hardware on performance. Specifically, we present the throughput improvement of ZIMP over Barrelfish MP when increasing the number of receivers. Results are presented on the three hardware described in Section IV-A. Figure 6 shows the case when increasing the number of receivers from 1 to 23, for messages of 128B. Results with other message sizes are shown in Appendix C. Results show that the percentage of improvement of

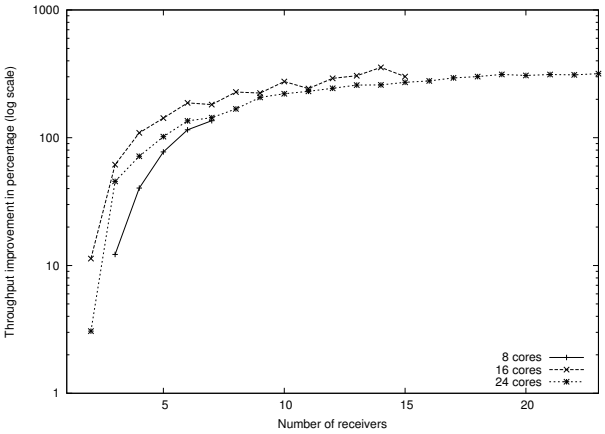

Figure 6: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 128B.

ZIMP over Barrelfish MP is consistent on different machines. In particular, above the configuration with one receiver, the average standard deviation between the three curves is less than 9% (with a minimum standard deviation of 2% observed for 11 receivers and a maximum of 22% observed for 7 receivers). For one receiver we observe larger differences in the performance of ZIMP compared to Barrelfish MP on different hardware. Indeed, the improvement appears to be negative (-7.7%) with 1 receiver on the 16 cores, while it is greater than 4.9% on the two other machines.

C. Macro-benchmarks

We have seen in the previous section that ZIMP and Barrelfish MP clearly outperform all other communication mechanisms. In this section, we compare the performance of these two mechanisms using two macro-benchmarks: the PaxosInside agreement protocol [17] and a checkpointing protocol [8].

1) Agreement protocol: Guerraoui and Yabandeh have recently proposed PaxosInside [17], an adaptation of the Paxos protocol [5] for manycore machines. In short, similarly to Paxos, the PaxosInside protocol distinguishes three roles for nodes taking part in the protocol: proposer, acceptor and learner. An important difference with Paxos is that PaxosInside relies on a single active acceptor, which is replaced by a backup acceptor when a failure occurs. For every consensus it performs, PaxosInside performs several rounds of one-to-one and one-to-many communications. More precisely, assuming that there are l learners in the system, every consensus requires $2 + l$ one-to-one message exchanges and 1 one-to-many message exchange.

In the presented experiments, we deploy PaxosInside with one proposer, one acceptor, and three learners (which allows tolerating the failure of exactly one learner). The proposer issues 100,000 requests for consensus. We measure the throughput at which consensus are performed as a function of the request size. We have experimented different node placements: the leader and the acceptor share the same cache, the acceptor and the learners share the same cache, etc. We did not observe noticeable differences between these placements. Therefore, we only report results for one placement.

Figure 7 presents the throughput improvement brought by ZIMP over Barrelfish MP, on the three machines presented in Section IV-A. Presented results are an average over three runs. The observed standard deviation was very low; we did thus not depict it. We observe that ZIMP systematically outperforms Barrelfish MP. We also observe that the improvement is quite important: on the 24-core machine, it ranges from 13% with 1B requests to 473% with 10kB requests. Finally, we observe that the throughput improvement starts to decrease for requests greater than 10kB to reach 83% with requests of 1MB.

We attribute this drop to an increase of the memory accesses, due to the limited size of the cache. Indeed, on the 3 machines, we observed an important augmentation of the number of cache misses (L2 for the 8-core machine, L3 for the 16 and 24-core machines) per byte for

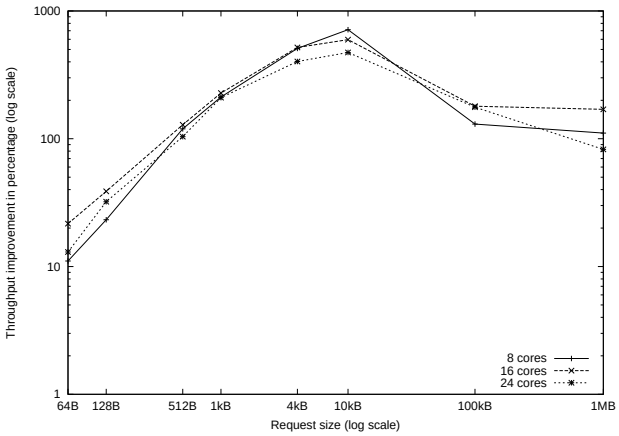

Figure 7: Agreement protocol: throughput improvement of ZIMP over Barrelfish MP as a function of the request size.

both mechanisms. We noticed that the number of cache misses is principally caused by the copy operation on messages. The larger the messages, the less the number of messages that can fit in the cache. Consequently, there are more accesses to the main memory. As an access to the main memory is more costly than an access to the cache, it limits the throughput that can be achieved by both mechanisms. Moreover, Barrelfish MP requires several memory copies, compared to ZIMP which is a zero-copy mechanism. Hence the number of cache misses is more important for Barrelfish MP than for ZIMP. Even if the number of cache misses increases for both mechanisms, ZIMP still offers a higher throughput than Barrelfish. The difference between the two mechansisms only becomes less important. Consequently, the improvement falls. Finally, note that the improvement is roughly equivalent accross the 3 machines.

2) Checkpointing protocol: Manivannan and Singhal have proposed a checkpointing protocol [8] that has the particularity to not rely on vector timestamps. In this report, we focus on a sub-protocol of this protocol, responsible for gathering snapshots. This sub-protocol works as follows: to gather a snapshot, a node sends a message to every other nodes in the system. Upon reception of such a message, nodes send back their latest checkpoint to the initiator of the snapshot. The snapshot is finished once the initiator received a checkpoint for every node in the system. This sub-protocol involves 1 one-to-many and n one-to-one exchanges of messages, where n is the number of nodes taking part in the protocol.

In the presented experiments, there is one node that

issues 100,000 requests for snapshots in a closed-loop, meaning that it only issues a new snapshot request when the previous one completed. The size of a snapshot request is always 128B. We vary the size of the checkpoints that are sent to the snapshot initiator (from 128B to 1MB) and the number of nodes participating in the system (from 2 to 24). We measure the time required to complete a snapshot. Results are all the average over three runs. The observed deviation was very low and is thus not depicted on the graphs. Results for configurations not presented in this section appear in Appendix D.

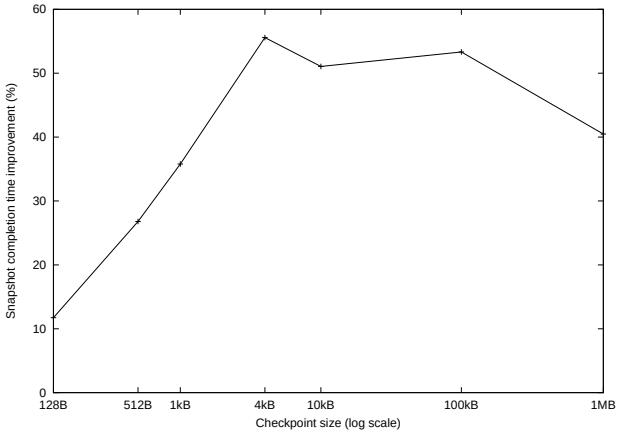

Figure 8: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (24 nodes).

Figure 8 depicts the snapshot completion time improvement brought by ZIMP over Barrelfish MP as a function of the checkpoint size, in a system with 24 nodes. On the 24-core machine, we observe that the improvement increases from +11.7% for checkpoints of 128B up to +55.6% for checkpoints of 4kB. Similarly to PaxosInside, it then decreases down to +40.5% for messages of 1MB. As for the agreement protocol, this is due to the limited size of the cache.

Figure 9 depicts the snapshot completion time improvement brought by ZIMP over Barrelfish MP as a function of the number of nodes in the system, for a checkpoint size of 4kB, on the three machines presented in Section IV-A. We observe that ZIMP improves the snapshot completion time over Barrelfish MP between +48% and +58% whatever the number of nodes in the system.

V. Related work

In 2003, Immich *et al.* have presented a study of five inter-process communication mechanisms across

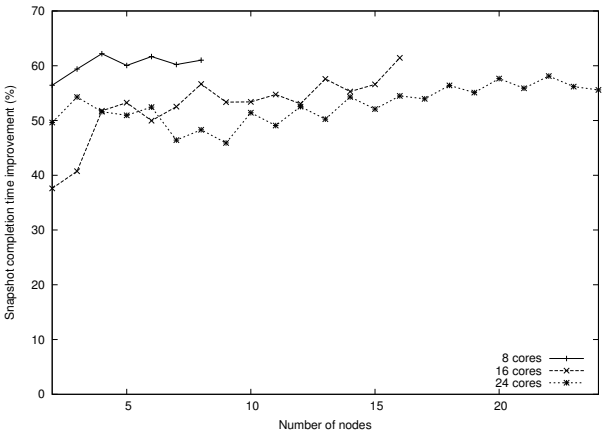

Figure 9: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the number of nodes (4kB checkpoints).

Unix and Linux operating systems [19]. More precisely, they have evaluated pipes, named pipes, IPC message queue, shared memory with semaphores and Unix domain sockets on four versions of Linux and two of FreeBSD. They considered a micro-benchmark, with a producer/consumer scheme, for messages ranging from 64B to 4608B. The main focus of this work was to study the impact of the operating system on the performance of existing inter-process communication mechanisms. K. Wright *et al.* [20] conducted a performance analysis of pipes, Unix Domain sockets and TCP sockets on different manycore platforms. The authors conducted their experiments on different hardware but focused on micro-benchmarks with only one sender and one receiver and large messages (ranging from 1MB to 100MB).

Our performance analysis of existing inter-process communication mechanisms is complementary to both the above studies. Indeed, neither we studied the impact of the operating system on performance (as performed in [19]) nor we studied the impact of very large messages on performance (as performed in [20]). Instead, none of the above works analyzes how existing inter-process communication mechanisms behave on manycore machines when they are used for one-to-many communications, which has been the focus of our study. Our results are further assessed on both micro- and macro-benchmarks. On the other hand, our work does not only compare existing mechanisms the ones against the others as performed in [19] and [20]. It further proposes a new inter-process communication mechanism that outperforms existing ones by up to an order of magnitude.

High Performance Computing (HPC) requires efficient intra-nodes message passing mechanisms. KNEM [21] is a Linux kernel module enabling HPC applications to efficiently send and receive large messages. Specifically, KNEM provides both one-to-one (i.e., unicast) and one-to-all (i.e., broadcast) communication primitives. The one-to-all communication primitive allows one core to send large amounts of data to all other cores in order to perform parallel processing. The difference with our mechanism is that ZIMP allows a process to communicate with a set of other processes whatever their number and their location. If a developer wants to use KNEM to send a message to a subset of processes it should either use the broadcast primitive and generate unnecessary traffic in the machine or use the unicast primitive many times, which is inefficient as shown for other mechanisms in this report.

Another approach to improve performance resides in the modification of the hardware. Lee *et al.* have recently presented HAQu [22], a piece of hardware that accelerates operations on software queues. Using a simulator and a micro-benchmark the authors reached a speed-up of 6.5 compared to a state-of-the-art software queue. ZIMP consists of an efficient alternative for those machines that do not provide hardware facilities for efficient inter-process communication.

VI. Conclusion

In this report we have seen that existing communication mechanisms are not efficient for intra-node communication in a manycore machine. We have presented ZIMP, a new communication mechanism which benefits from a minimal number of copies of messages and from an efficient one-to-many communication primitive. We have implemented a micro-benchmark and two distributed algorithms: PaxosInside, a consensus algorithm for manycore machines, and a checkpointing algorithm. The evaluation shows that ZIMP is more efficient than state-of-the-art mechanisms.

Acknowledgments

We are very grateful to Fabien Gaud, Baptiste Lepers, Alessio Pace and Nicolas Palix for their helpful feedback on this work

Some experiments presented in this report were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 29–44.

[2] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability(ASID '06)*. ACM, 2006, pp. 25–33.

[3] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: ten years later," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11. New York, NY, USA: ACM, 2011, pp. 305–318.

[4] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer pcs," in *Proceedings of Eurosys 2011*. ACM, 2011.

[5] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.

[6] F. P. Junqueira and B. C. Reed, "Brief announcement zab: a practical totally ordered broadcast protocol," in *Proceedings of the 23rd international conference on Distributed computing*, ser. DISC'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 362–363.

[7] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, pp. 5:1–5:32, July 2010.

[8] D. Manivannan and M. Singhal, "Asynchronous recovery without using vector timestamps," *J. Parallel Distrib. Comput.*, vol. 62, pp. 1695–1728, December 2002.

[9] R. Prakash and M. Singhal, "Low-cost checkpointing and failure recovery in mobile computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 1035–1048, October 1996.

[10] X. Ren, R. Eigenmann, and S. Bagchi, "Failure-aware checkpointing in fine-grained cycle sharing systems," in *Proceedings of the 16th international symposium on High performance distributed computing*, ser. HPDC '07. New York, NY, USA: ACM, 2007, pp. 33–42.

[11] I. C. Garcia and L. E. Buzato, "An efficient checkpointing protocol for the minimal characterization of operational rollback-dependency trackability," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, ser. SRDS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 126–135.

[12] L. Ziarek, P. Schatz, and S. Jagannathan, "Stabilizers: a modular checkpointing abstraction for concurrent functional programs," in *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ser. ICFP '06. New York, NY, USA: ACM, 2006, pp. 136–147.

[13] K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, "Micro-checkpointing: Checkpointing for multithreaded applications," in *Proceedings of the 6th IEEE International On-Line Testing Workshop (IOLTW)*, ser. IOLTW '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 3–.

[14] C. Karlsson and Z. Chen, "Highly scalable checkpointing for exascale computing," in *IPDPS Workshops*, 2010, pp. 1–4.

[15] R. Barbosa and J. Karlsson, "On the integrity of lightweight checkpoints," in *Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 125–134.

[16] I. Majzik, A. Pataricza, M. D. Cin, W. Hohl, J. Hönig, and V. Sieh, "Hierarchical checking of multiprocessors using watchdog processors," in *Proceedings of the First European Dependable Computing Conference on Dependable Computing*, ser. EDCC-1. London, UK: Springer-Verlag, 1994, pp. 386–403.

[17] R. Guerraoui and M. Yabandeh, "PaxosInside," EPFL, Tech. Rep. EPFL-REPORT-153309, 2010.

[18] T. Liu and E. D. Berger, "Sheriff: Detecting and eliminating false sharing," University of Massachusetts, Amherst, Tech. Rep. UM-CS-2010-047, 2010.

[19] P. K. Immich, R. S. Bhagavatula, and R. Pendse, "Performance analysis of five interprocess communication mechanisms across unix operating systems," *J. Syst. Softw.*, vol. 68, pp. 27–43, October 2003.

[20] K.-L. Wright and K. Gopalan, "Performance analysis of inter-process communication mechanisms," Binghamton University, Tech. Rep. TR-20070820, 2007.

[21] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis," in *Proceedings of the 2009 International Conference on Parallel Processing*, ser. ICPP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 462–469.

[22] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck, "HAQu: Hardware accelerated queueing for fine-grained threading on a chip multiprocessor," in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA-17, February 2011.

APPENDIX

A. Microbenchmark: impact of the message size

In this section we present the impact of the message size on the performance of the various mechanisms. The experiments have been performed on the 24-core machine. We present results from 2 (Figure 10) to 22 receivers (Figure 30)).

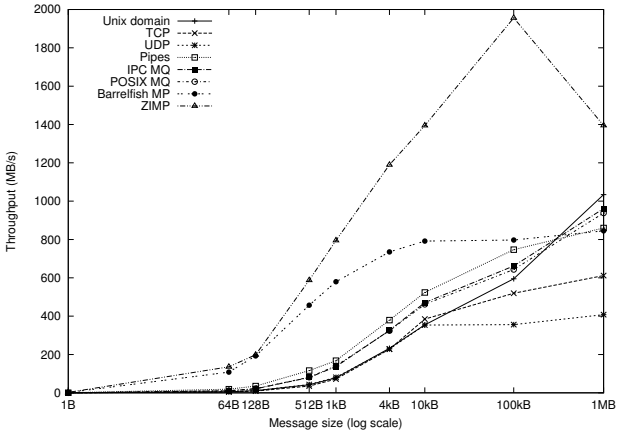

Figure 10: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (2 receivers).

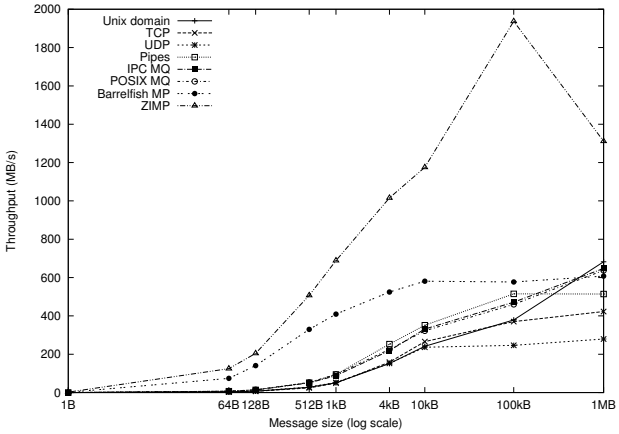

Figure 11: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (3 receivers).

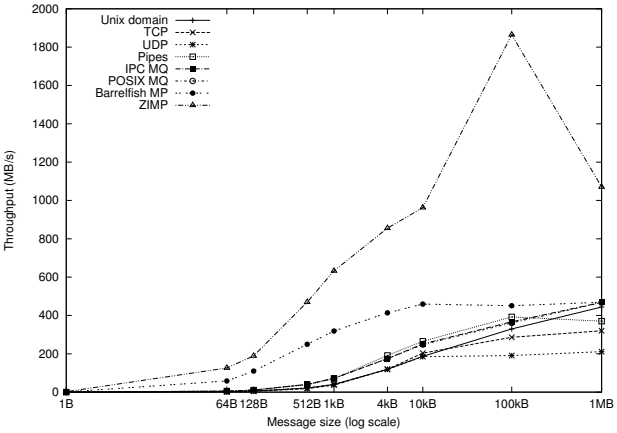

Figure 12: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (4 receivers).

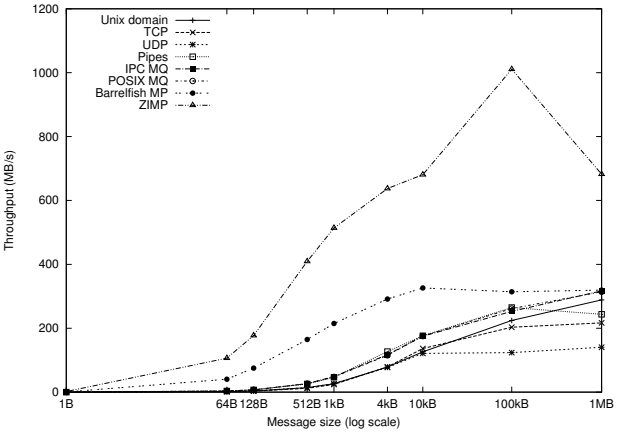

Figure 14: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (6 receivers).

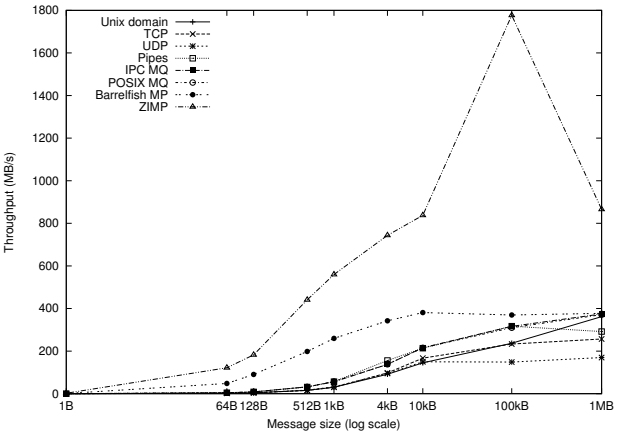

Figure 13: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (5 receivers).

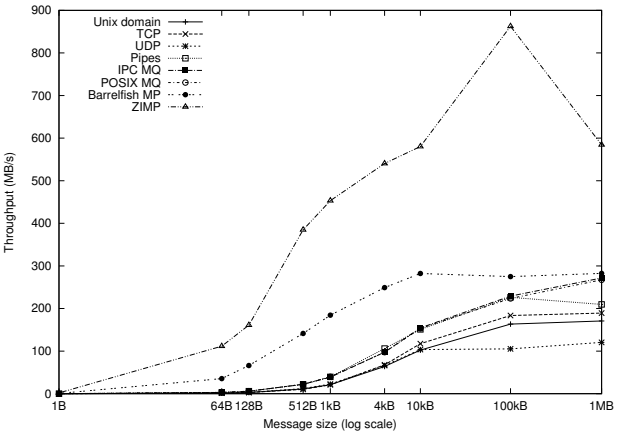

Figure 15: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (7 receivers).

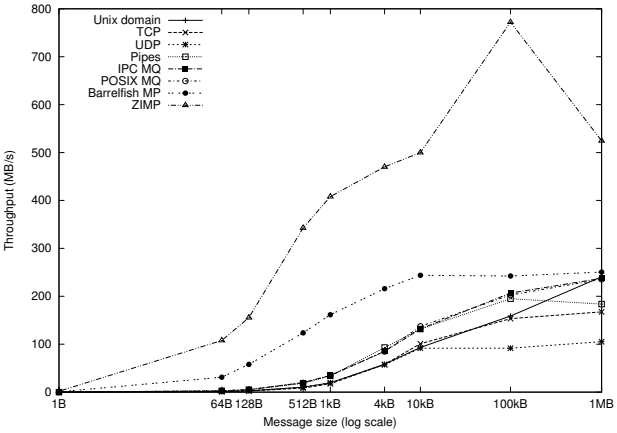

Figure 16: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (8 receivers).

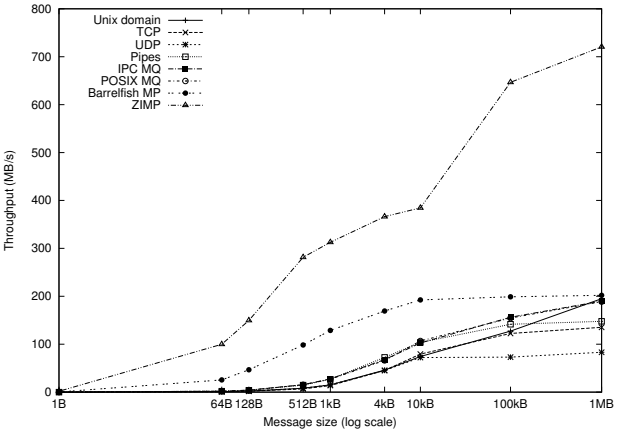

Figure 18: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (10 receivers).

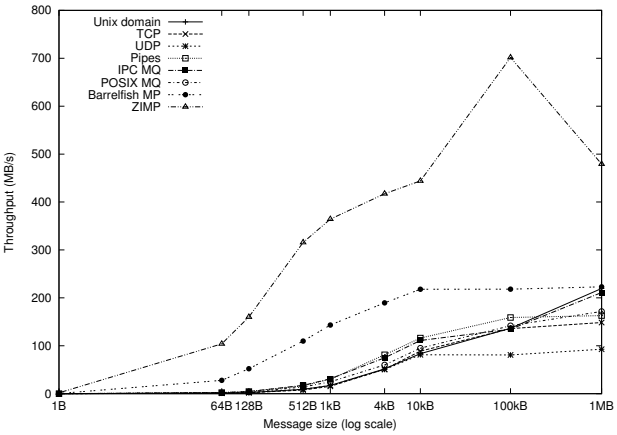

Figure 17: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (9 receivers).

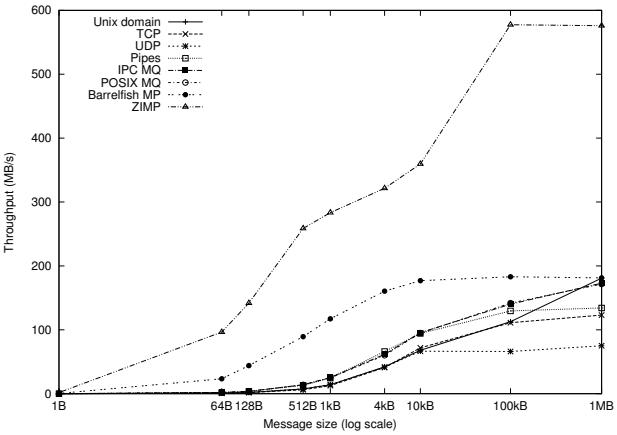

Figure 19: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (11 receivers).

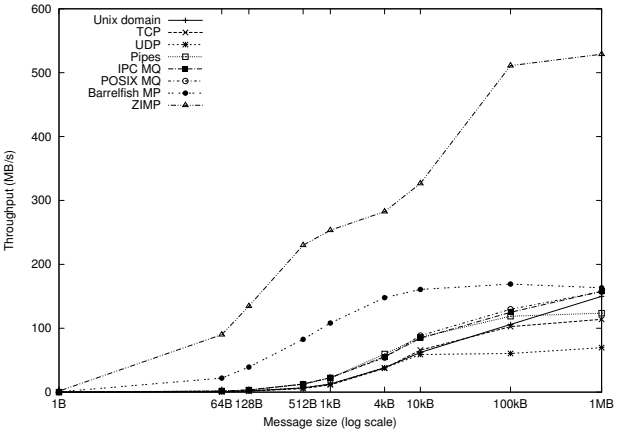

Figure 20: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (12 receivers).

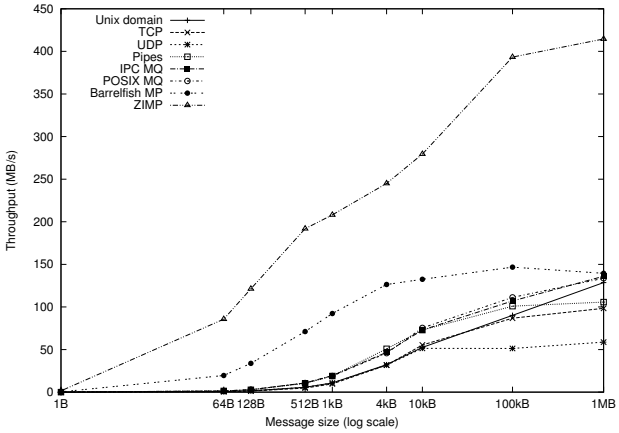

Figure 22: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (14 receivers).

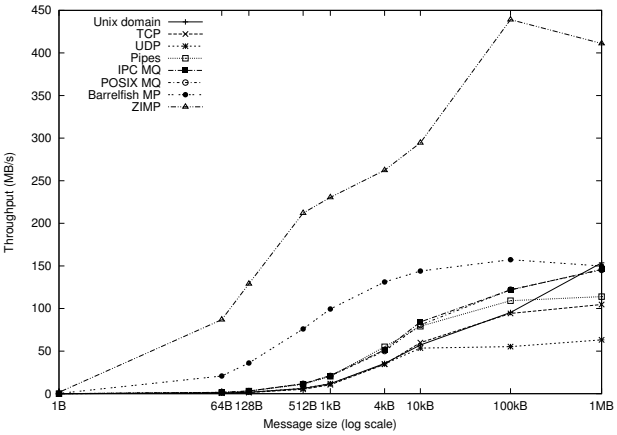

Figure 21: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (13 receivers).

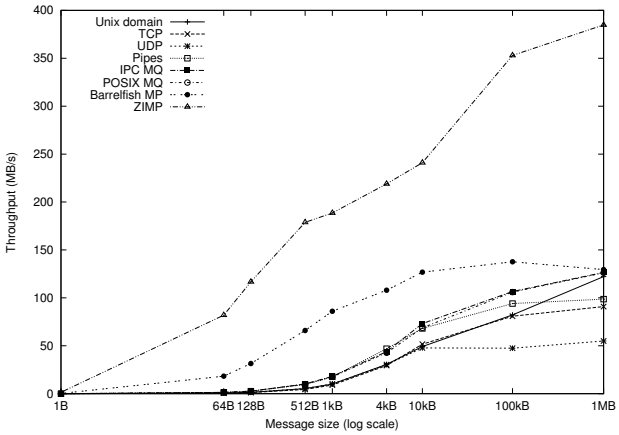

Figure 23: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (15 receivers).

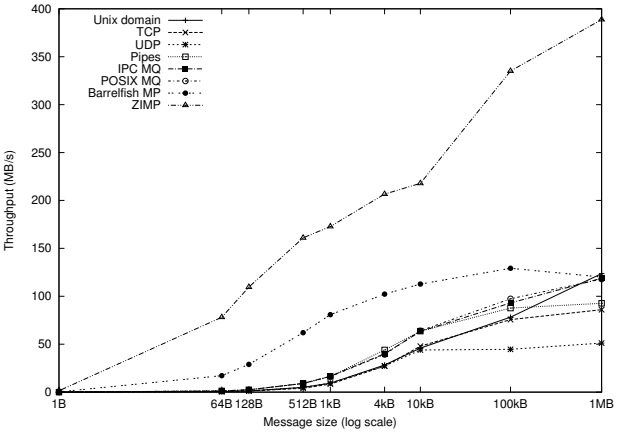

Figure 24: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (16 receivers).

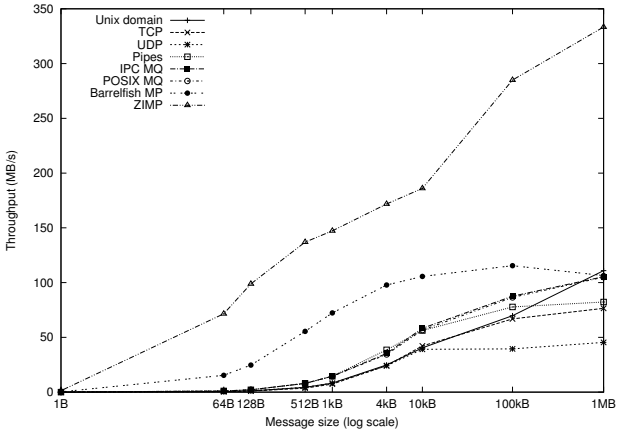

Figure 26: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (18 receivers).

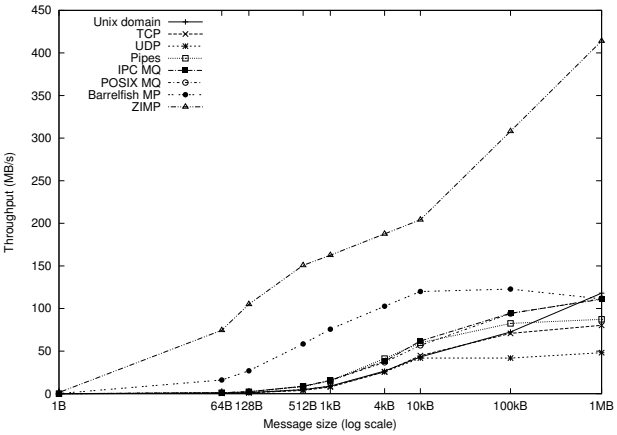

Figure 25: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (17 receivers).

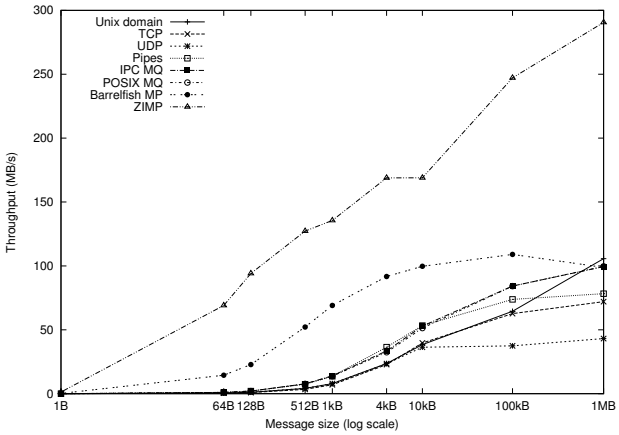

Figure 27: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (19 receivers).

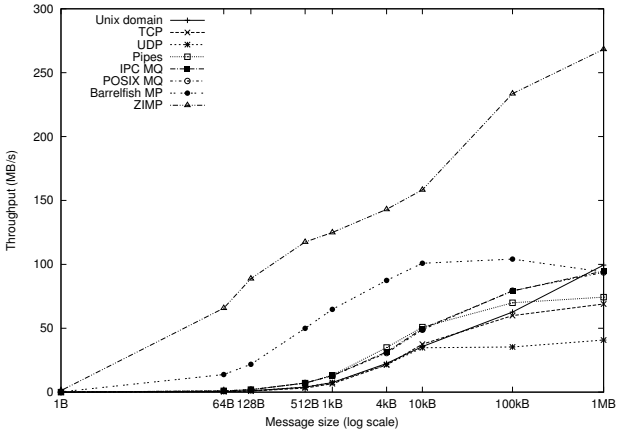

Figure 28: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (20 receivers).

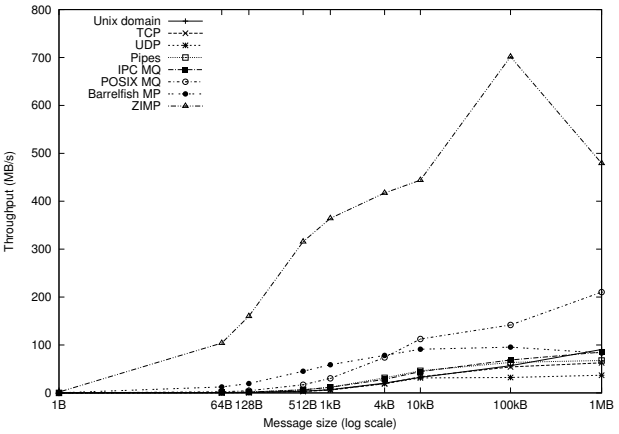

Figure 30: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (22 receivers).

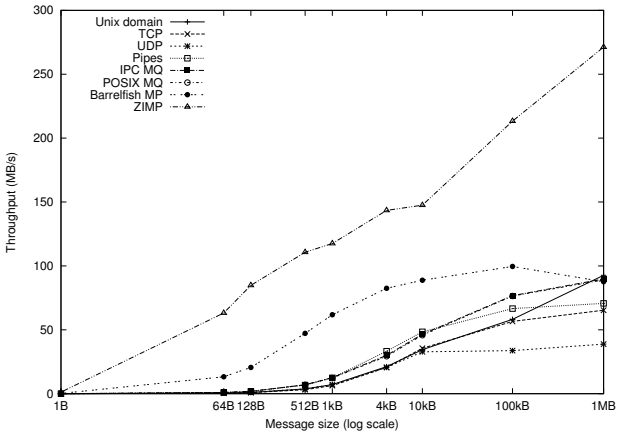

Figure 29: Micro-benchmark: throughput of the different communication mechanisms as a function of the message size (21 receivers).

B. Microbenchmark: impact of the number of receivers

In this section we present the impact of the number of receivers on the performance of ZIMP compared to the state-of-the-art communication mechanisms. The experiments have been performed on the 24-core machine. We present results for messages of 64B, 128B, 512B, 1kB, 4kB, 10kB and 100kB, from Figure 31 to Figure 37.

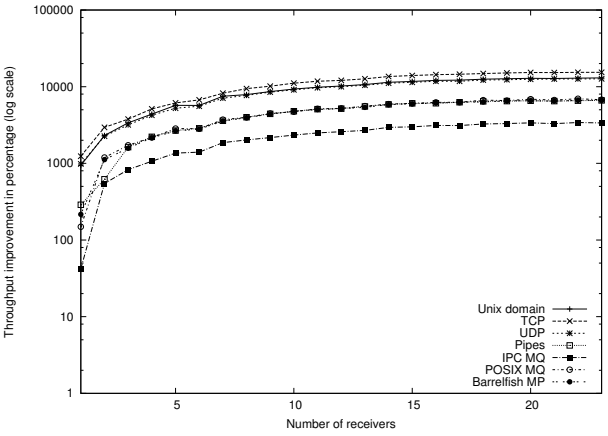

Figure 31: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (64B messages).

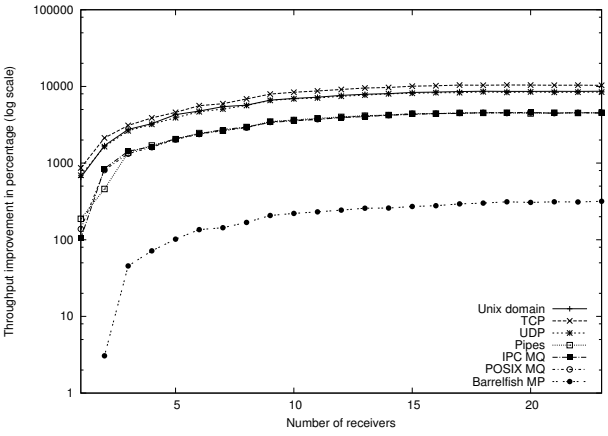

Figure 32: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (128B messages).

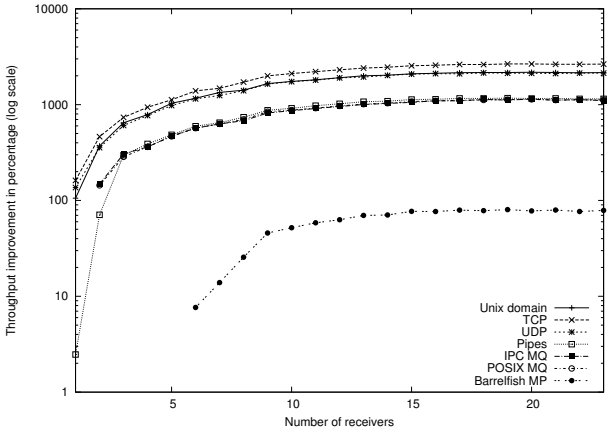

Figure 33: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (512B messages).

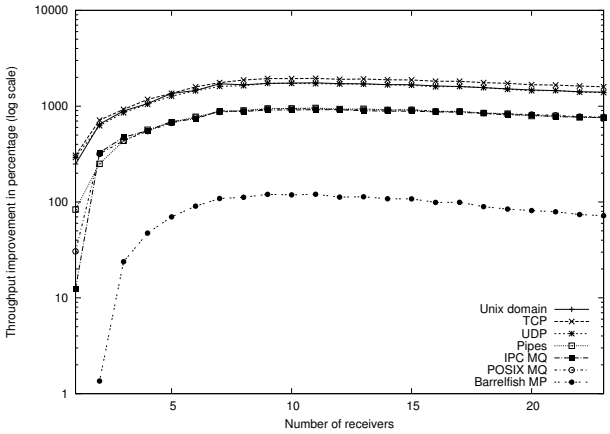

Figure 34: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (1kB messages).

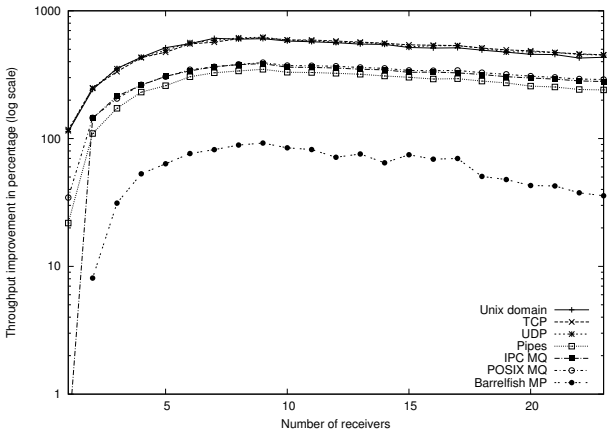

Figure 35: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (4kB messages).

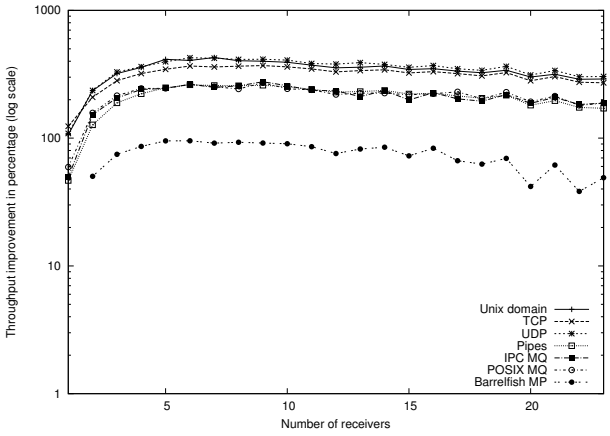

Figure 36: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (10kB messages).

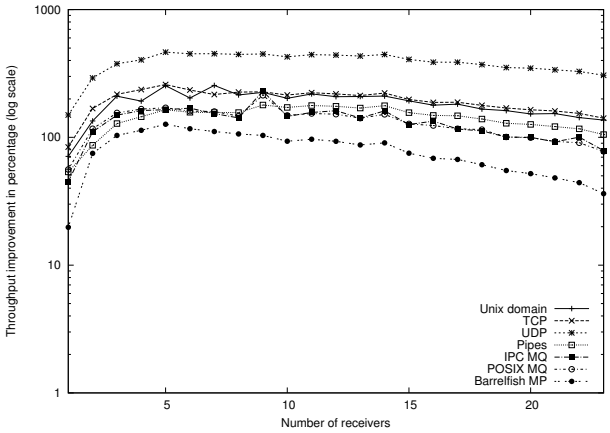

Figure 37: Micro-benchmark: throughput improvement of ZIMP over state-of-the-art mechanisms as a function of the number of receivers (100kB messages).

C. Microbenchmark: impact of the hardware

In this section we present the impact of the hardware on ZIMP throughput improvement over Barrelfish MP, for different message sizes. More specifically, we present results for messages of 1B, 64B, 512B, 1kB, 4kB, 10kB, 100kB and 1MB, from Figure 38 to Figure 45.

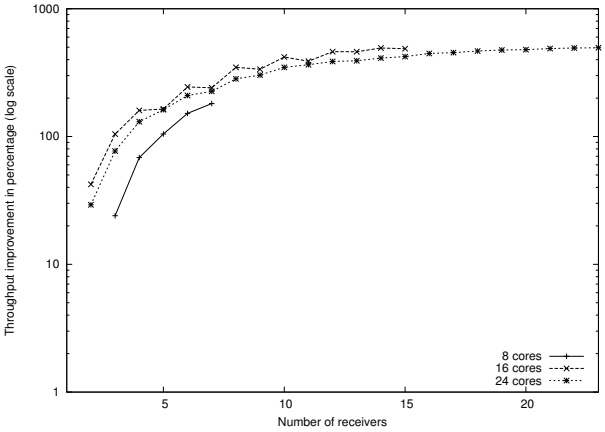

Figure 38: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 1B.

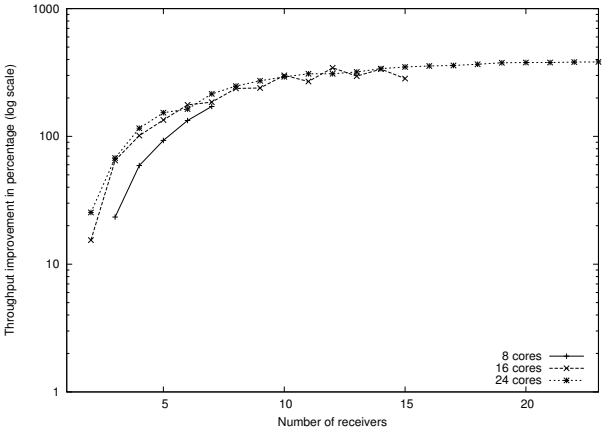

Figure 39: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 64B.

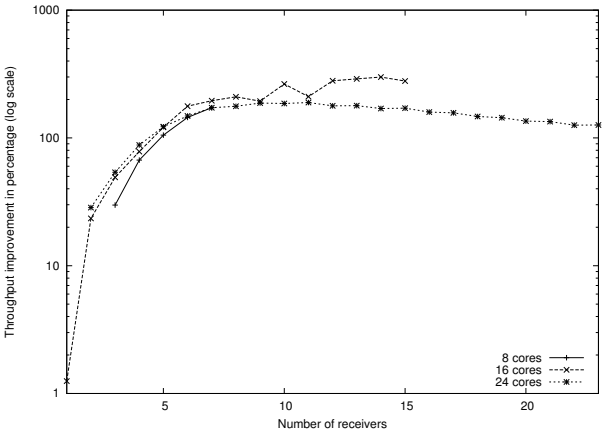

Figure 40: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 512B.

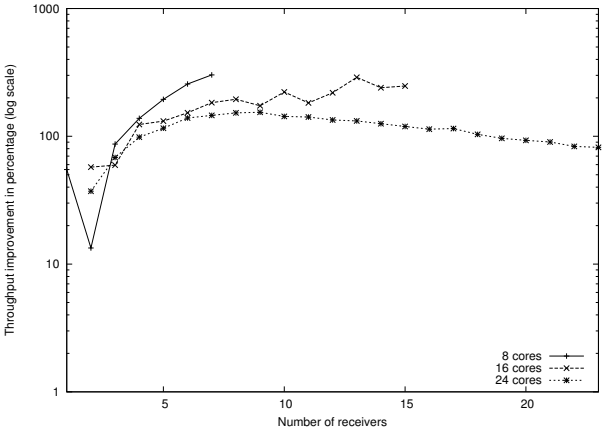

Figure 41: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 1kB.

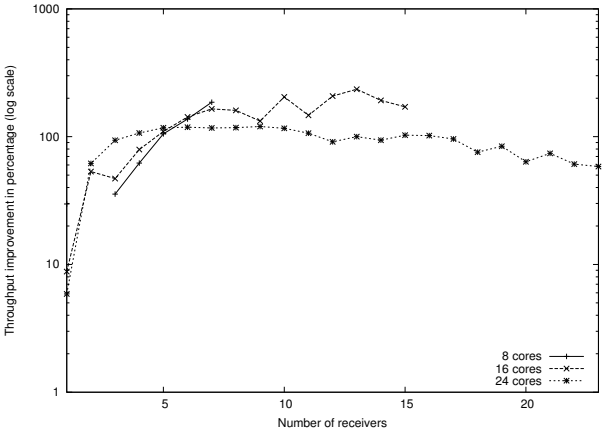

Figure 42: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 4kB.

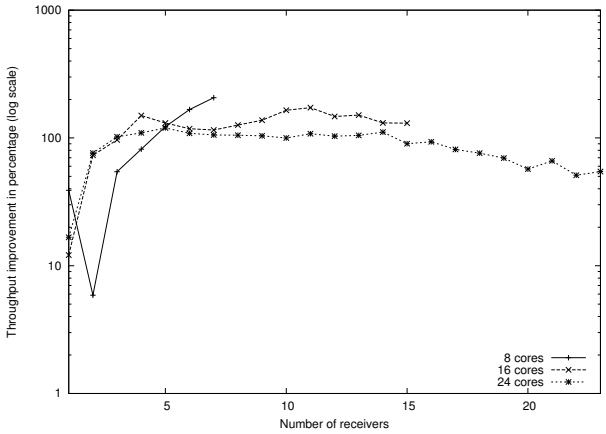

Figure 43: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 10kB.

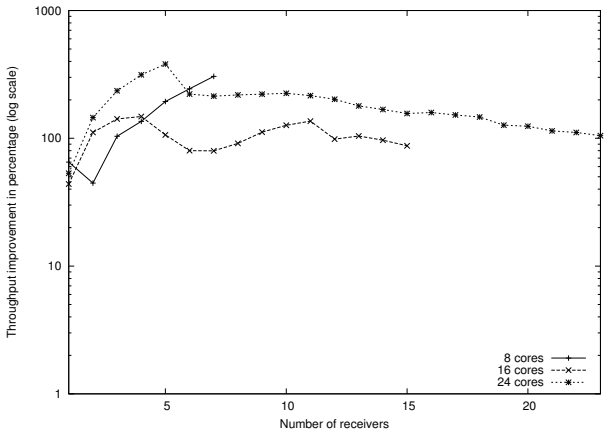

Figure 44: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 100kB.

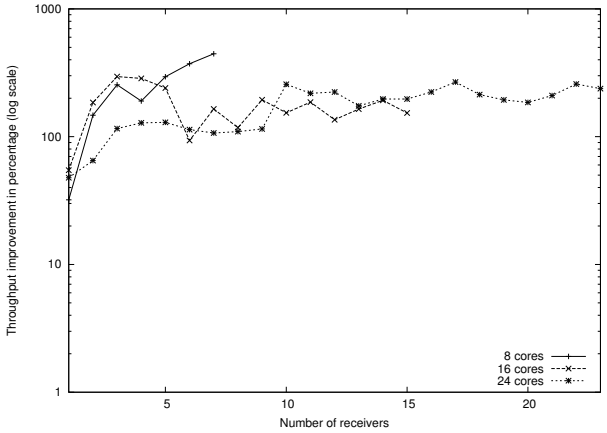

Figure 45: ZIMP throughput improvement over Barrelfish MP on different hardware. Messages of 1MB.

D. Checkpointing protocol: snapshot completion time improvement

In this section we present the snapshot completion time improvement of ZIMP over Barrelfish MP for different checkpoint sizes and different number of nodes.

Specifically, we first present the snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the number of nodes for checkpoint sizes of 128B, 512B, 1kB, 10kB and 100kB, from Figure 46 to Figure 51.

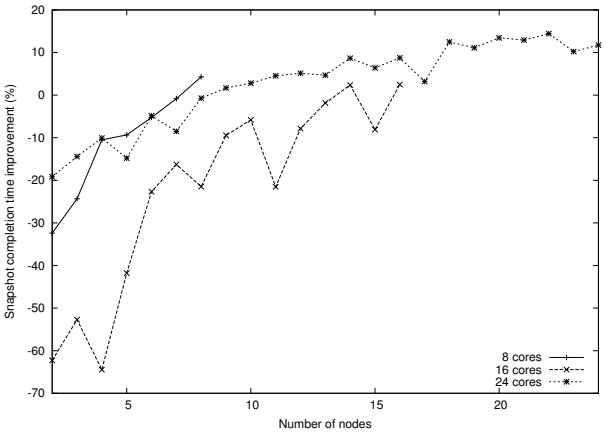

Figure 46: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the number of nodes (128B checkpoints).

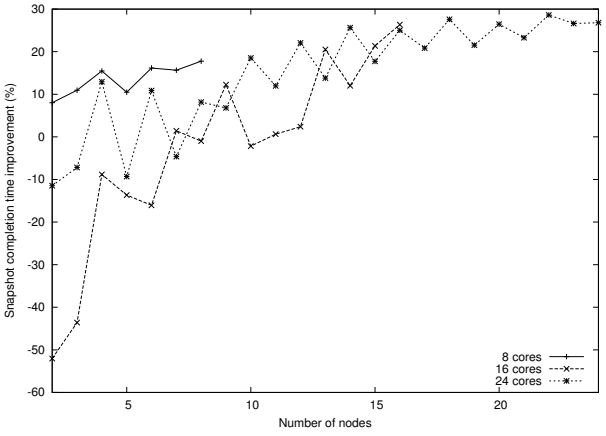

Figure 47: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the number of nodes (512B checkpoints).

Second, we present the snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint sizes, from 2 (Figure 52) to 23 (Figure 74) nodes.

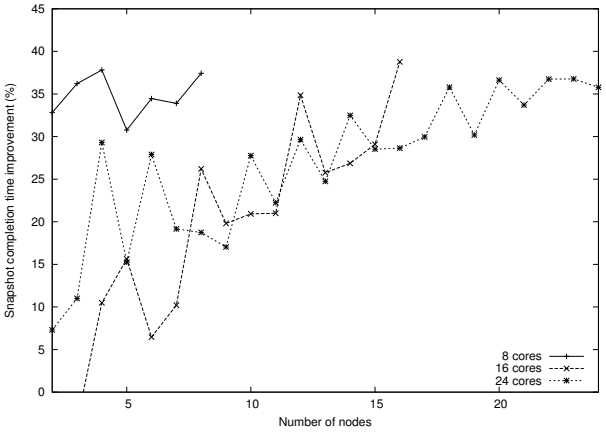

Figure 48: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the number of nodes (1kB checkpoints).

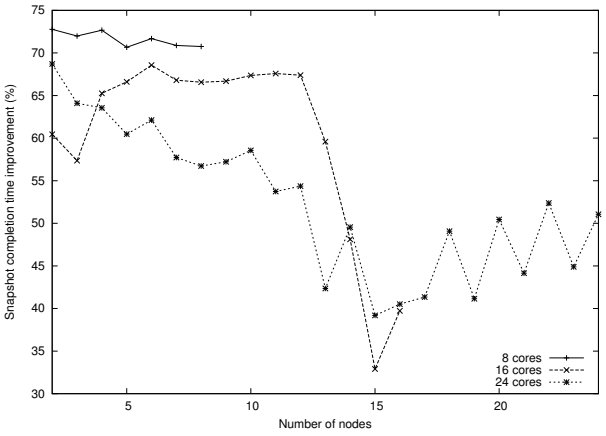

Figure 49: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the number of nodes (10kB checkpoints).

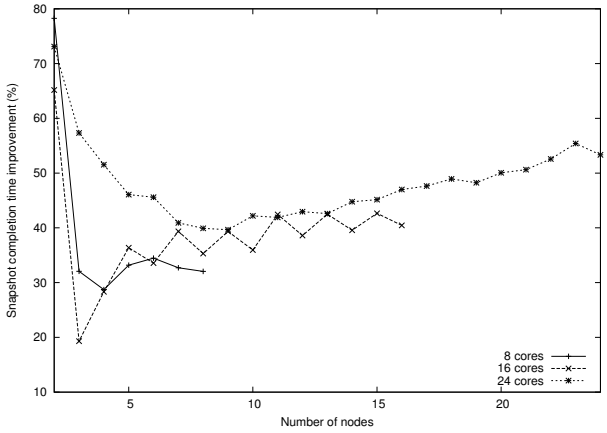

Figure 50: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the number of nodes (100kB checkpoints).

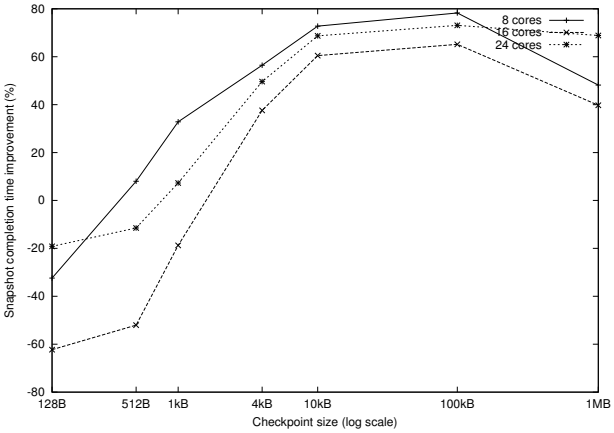

Figure 52: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (2 nodes).

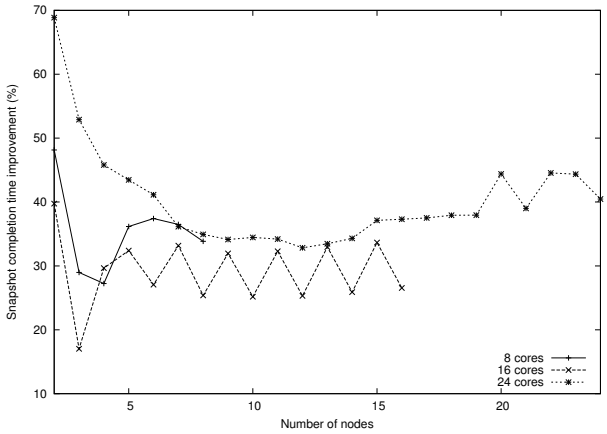

Figure 51: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the number of nodes (1MB checkpoints).

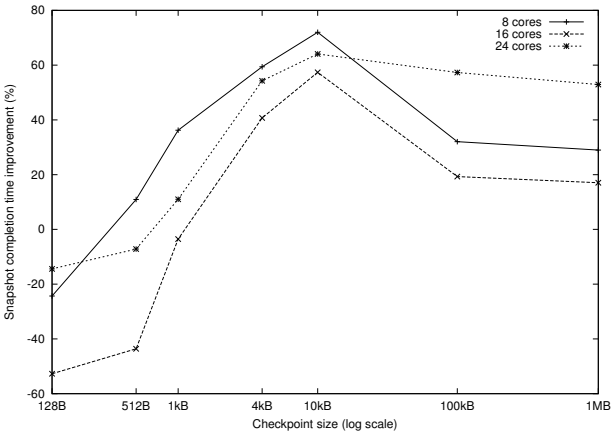

Figure 53: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (3 nodes).

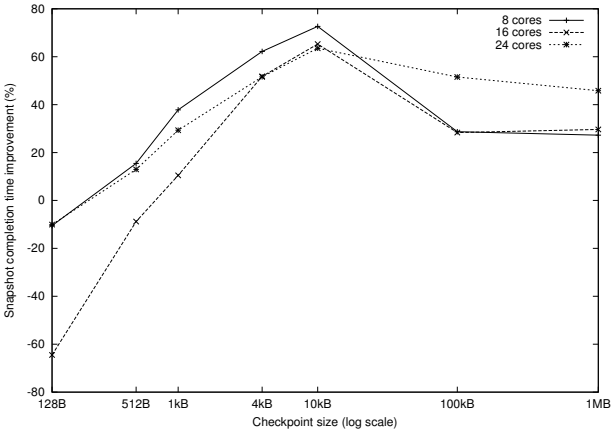

Figure 54: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (4 nodes).

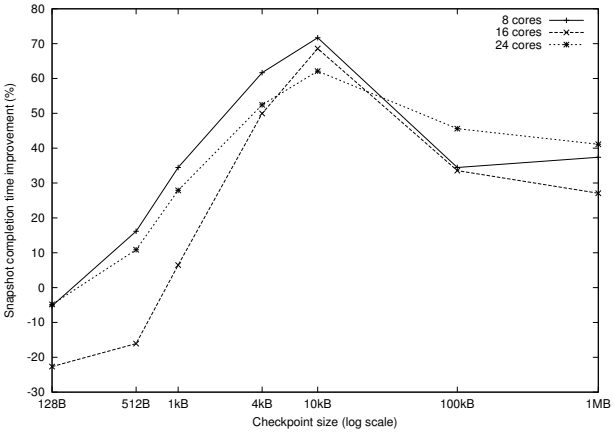

Figure 56: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (6 nodes).

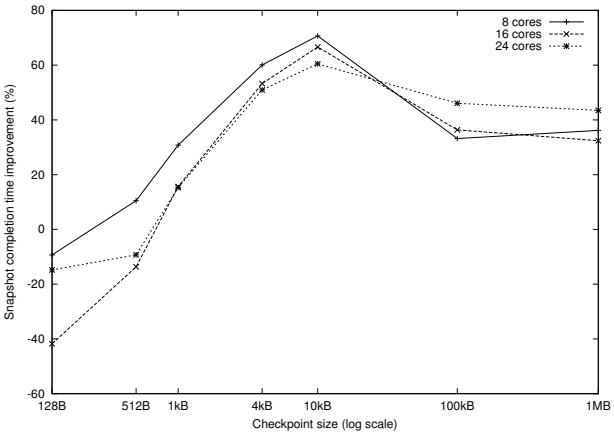

Figure 55: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (5 nodes).

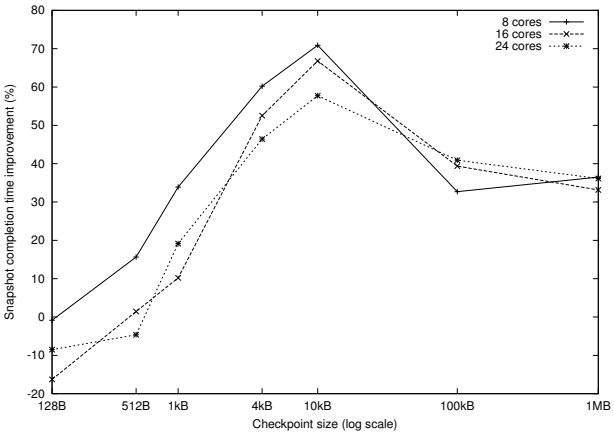

Figure 57: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (7 nodes).

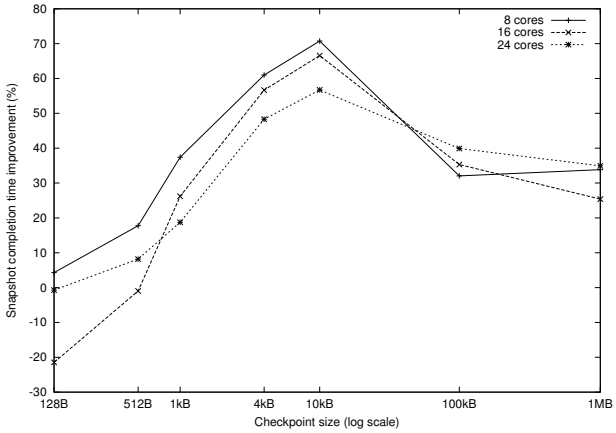

Figure 58: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (8 nodes).

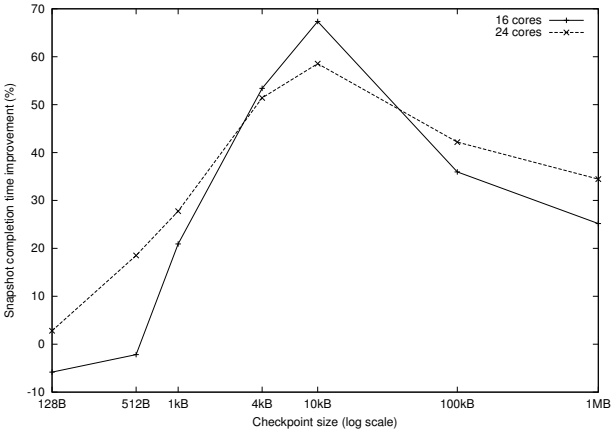

Figure 60: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (10 nodes).

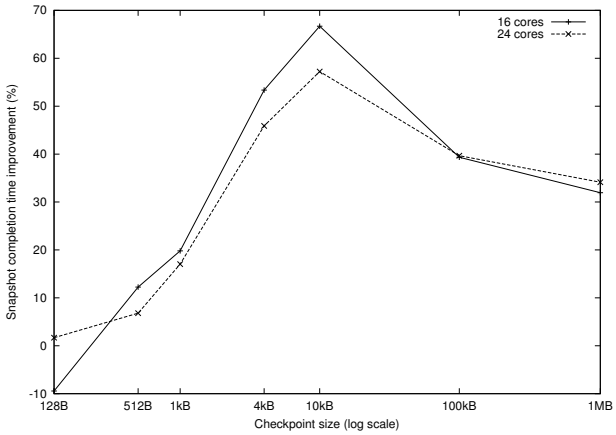

Figure 59: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (9 nodes).

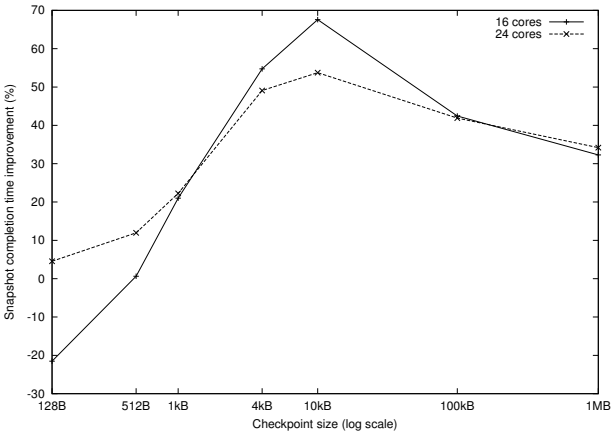

Figure 61: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (11 nodes).

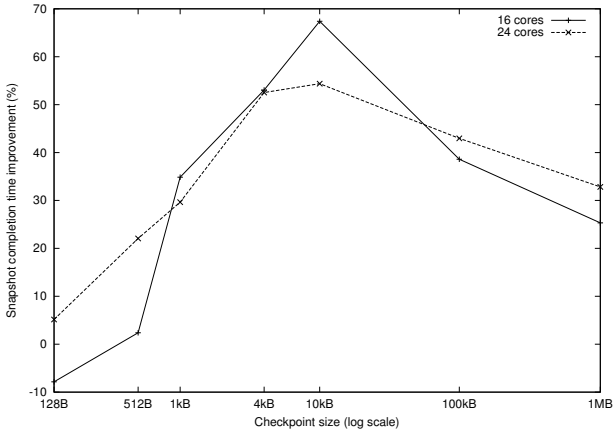

Figure 62: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (12 nodes).

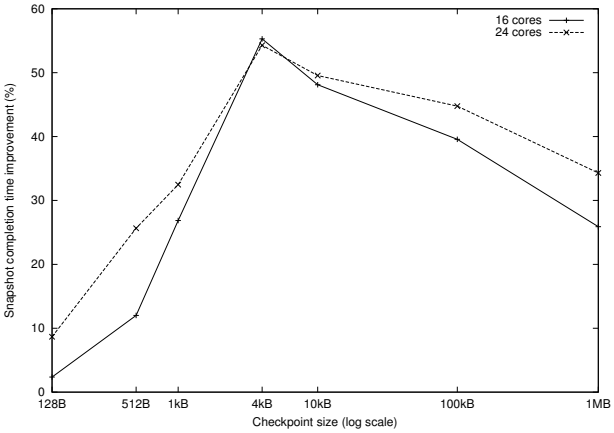

Figure 64: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (14 nodes).

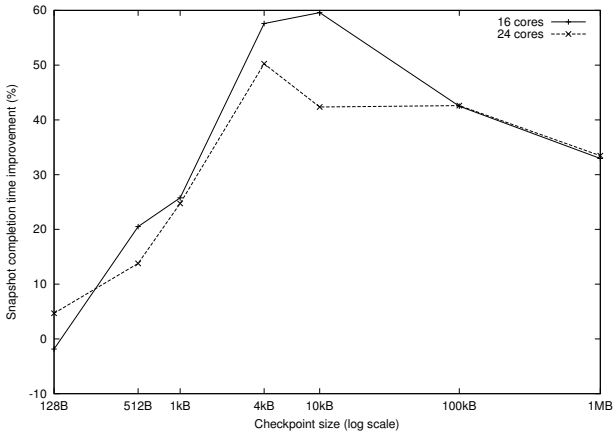

Figure 63: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (13 nodes).

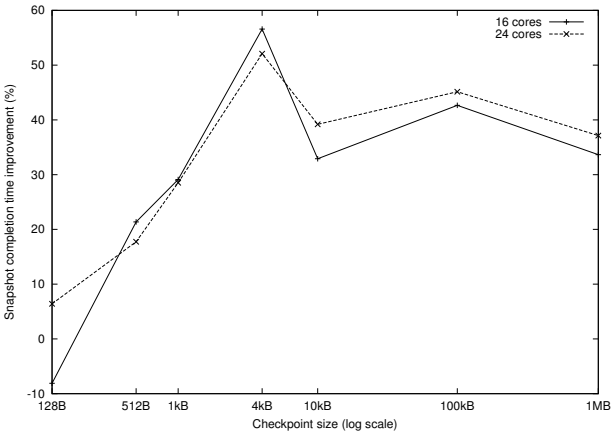

Figure 65: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (15 nodes).

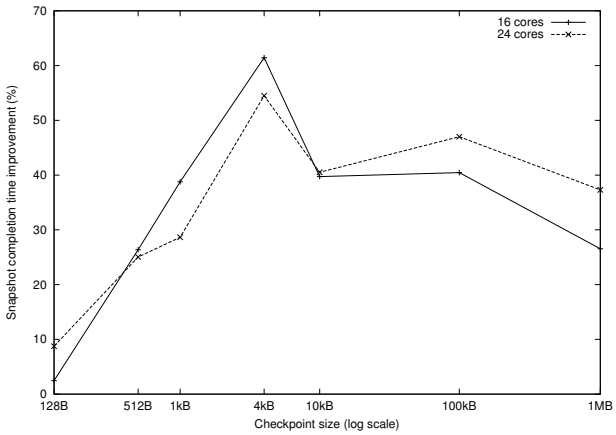

Figure 66: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (16 nodes).

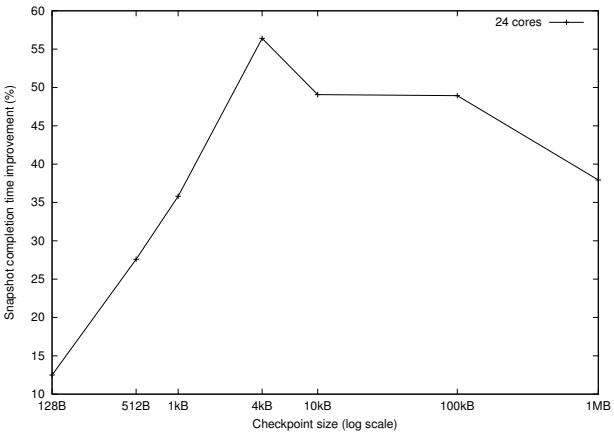

Figure 68: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (18 nodes).

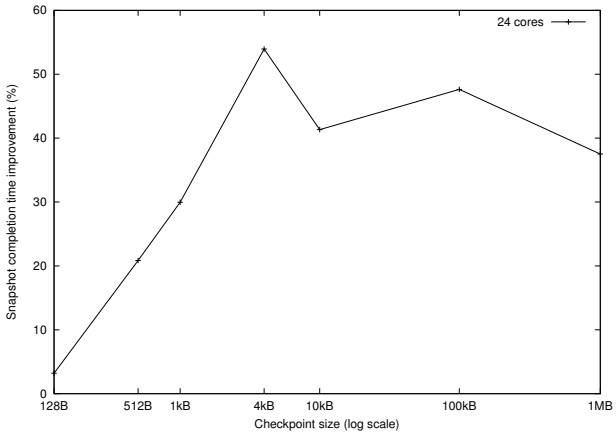

Figure 67: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (17 nodes).

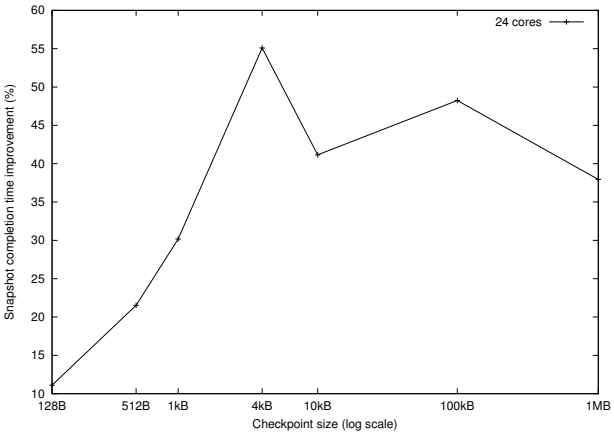

Figure 69: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (19 nodes).

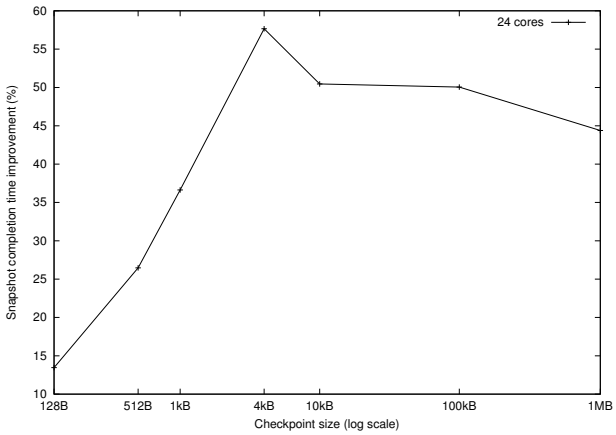

Figure 70: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (20 nodes).

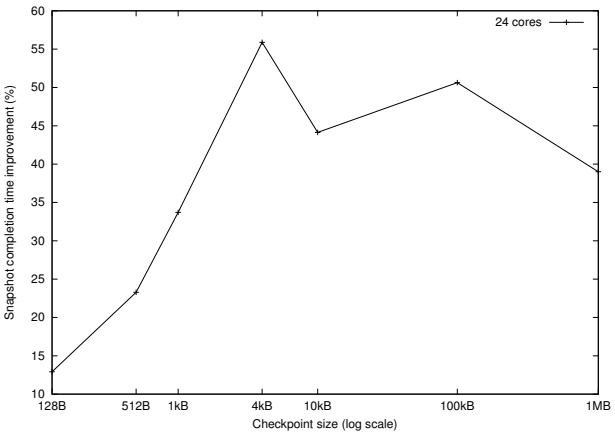

Figure 72: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (21 nodes).

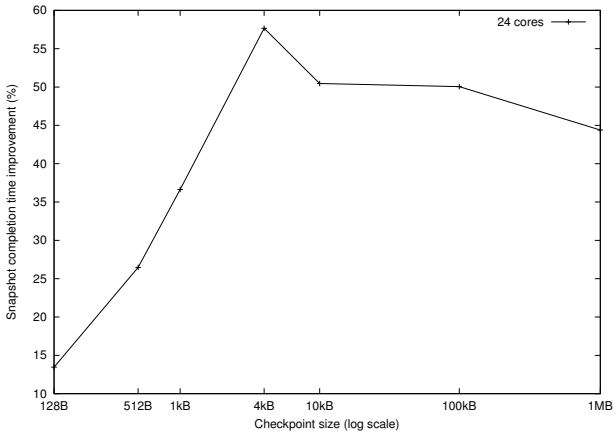

Figure 71: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (20 nodes).

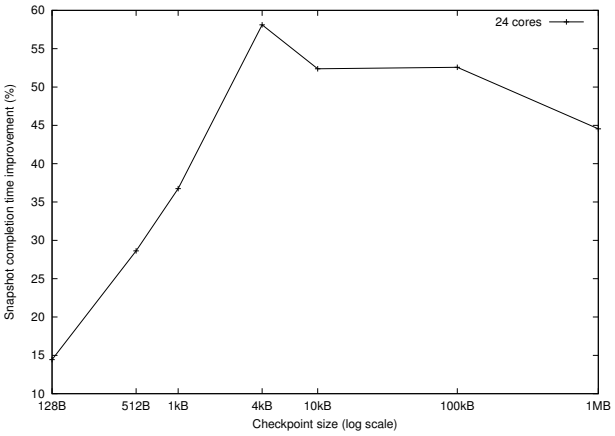

Figure 73: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (22 nodes).

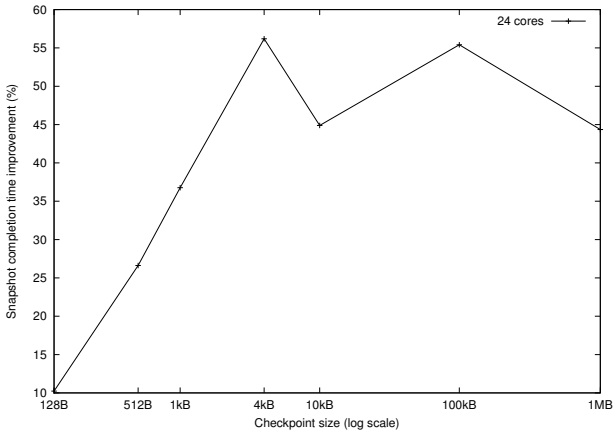

Figure 74: Checkpointing protocol: snapshot completion time improvement of ZIMP over Barrelfish MP as a function of the checkpoint size (23 nodes).