Asynchronous Broadcast on the Intel SCC using Interrupts

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

***Abstract*—This paper focuses on the design of an asynchronous broadcast primitive on the Intel SCC. Our solution is based on OC-Bcast, a state-of-the-art k-ary tree synchronous broadcast algorithm that leverages the parallelism provided by on-chip Remote Memory Accesses to Message Passing Buffers. In the paper, we study the use of parallel inter-core interrupts as a means to implement an efficient asynchronous group communication primitive, and present the userspace library we designed to be able to use interrupts in OC-Bcast and make it work asynchronously. Our experimental evaluation shows that our algorithm allows parallel broadcast operations to efficiently progress concurrently and provides low latency for a single broadcast operation. It highlights that parallel interrupts can help implementing efficient group communication primitives on many-core systems.**

I. Introduction

Recent research in microprocessor design indicates that the most promising way to achieve high performance while lowering power consumption is to integrate many loosely-coupled processors on a single chip [1]. A many-core chip can be viewed as a distributed system, i.e. a set of cores connected through a Network on Chip (NoC). The Intel Single-Chip Cloud Computer (SCC) is a 48-core research prototype of a many-core chip, designed to be operated as a message passing system.

Group communications, such as broadcast, are of major importance in message passing systems, and have been widely studied in different contexts. Considering the low latency and high throughput of a NoC, a many-core chip is very similar to a parallel High Performance Computing (HPC) system. However, results show that porting an HPC communication library to the SCC requires rethinking the design of the communication algorithms [10].

In this paper, we study the implementation of an asynchronous broadcast primitive for the Intel SCC. Our previous work, done in the context of Single Program Multiple Data (SPMD) applications, studied synchronous broadcast operations [10]. It shows that leveraging specific features of the Intel SCC, *i.e.*, Remote Memory Access (RMA) to on-chip Message Passing Buffers (MPB), helps improving the performance of group communications by increasing parallelism in the data dissemination. We adapt the resulting algorithm, called OC-Bcast (On-Chip Broadcast), to work asynchronously in order to be able to use it in a more general execution model. To do so, we propose to use parallel Inter-Processor Interrupts (IPI).

The paper presents the following contributions:

- A study of the global interrupt controller (GIC) on the Intel SCC, and the description of a library to simply manipulate IPIs in userspace (Section IV).
- An asynchronous version of OC-Bcast based on parallel IPIs that allows arbitrary interleaving of concurrent broadcast operations (Section V).
- An evaluation of the proposed algorithm showing that it manages to achieve both low single broadcast latency and high concurrent broadcasts throughput, demonstrating usefulness of parallel IPIs in implementing efficient group communication on many-core chips (Section VI).

Before detailing the contributions, we describe the SCC in Section II and focus on the related work on interrupt-based communication and broadcast on the SCC in Section III.

II. The Intel SCC

The SCC is a general purpose many-core prototype developed by Intel Labs. In this section we briefly describe the SCC architecture and inter-core communication.

a) Architecture: The cores and the NoC of the SCC are depicted in Figure 1. There are 48 Pentium P54C cores, grouped into 24 tiles (2 cores per tile) and connected through a 2D mesh NoC. Tiles are numbered from (0,0) to (5,3). Each tile is connected to a router. The NoC uses high-throughput, low-latency links and deterministic virtual cut-through X-Y routing [5]. Memory components are divided into (i) message passing buffers (MPB), (ii) L1 and L2 caches, as well as (iii) off-chip private memories. Each tile has a small (16KB) on-chip MPB equally divided between the two cores. The MPBs allow on-chip inter-core communication using Remote Memory Access (RMA): Each core is able to read and write in the MPB of all other cores. There is no hardware cache coherence for the L1 and L2 caches. By default, each core has access to a private off-chip memory through one of the four memory controllers, denoted by MC in Figure 1. In addition, an external programmable off-chip component (FPGA) is provided to add new hardware features to the prototype.

b) Inter-core communication: To leverage on-chip RMA, cores can transfer data using the one-sided *put* and *get* primitives provided by the RCCE library [8]. Using *put*, a core (a) reads a certain amount of data from its own MPB or its private off-chip memory and (b) writes it to some MPB. Using *get*, a core (a) reads a certain amount of data from some MPB and (b) writes it to its own MPB or its private off-chip memory. The unit of data transmission is the cache line, equal

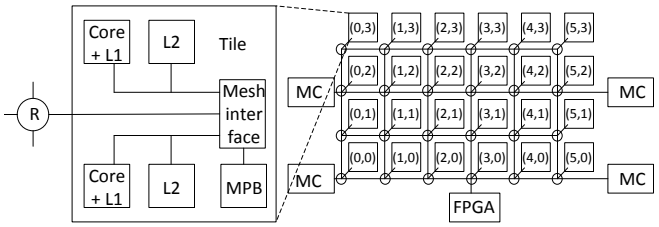


Fig. 1: SCC Architecture

to 32 bytes. If the data is larger than one cache line, it is sequentially transferred in cache-line-sized packets. During a remote read/write operation, each packet traverses the routers on the way from the source to the destination.

Cores are also able to notify each other using inter-process interrupts (IPI), either by writing directly into the receiving core configuration register or by using the Global Interrupt Controller (GIC)[1]. In the latter case, the receiving core is able to obtain additional information about the interrupt through a set of GIC registers. We consider the GIC in this work.

III. Related Work

In the SCC context, there are works on interrupt-based message passing ([6], [7], [9], [11], [12]), as well as on the implementation of collective operations ([4], [2], [8]). However, to the best of our knowledge, there is no work combining the two, that is, leveraging IPIs for collective communication. For this reason, we present the related work in two categories: (i) papers that focus on the collectives, i.e. broadcast and (ii) those that discuss interrupt-based communication.

A. Broadcast Algorithms

Despite several implementations of broadcast on the Intel SCC, the only scenario considered so far is running HPC applications. This assumes the SPMD model, in which each core runs the same program and every core explicitly invokes a routine to participate in a collective operation. As a consequence of this assumption, polling can be used for notification and asynchronous primitives are not necessary.

When it comes to the broadcast algorithms used, RCCE_comm [3], as well as RCKMPI [2] use well-known algorithms based on two-sided communication – binomial tree and scatter-allgather. On the other hand, OC-Bcast [10] applies a tree based algorithm for broadcast directly on top of *put*/*get* primitives, which dramatically improves both latency and throughput by minimizing memory copy operations on the critical path. The algorithm presented in this paper has been directly derived from OC-Bcast, as described in Section IV.

B. Communication Based on Interrupts

The assumption of having only one program running at a time, as well as synchronous communication among cores, which holds for HPC applications, is not valid in general-purpose distributed systems. Therefore, using interrupts for asynchronous communication is a must for porting such systems to the SCC.[2] Examples of SCC software relying upon inter-core interrupts are numerous ([6], [7], [9], [11], [12]). In the context of this paper, most interesting works are those that give specific details on different ways of using interrupts and their cost in terms of performance.

The SCC port of Barrelfish [9] uses IPIs to notify cores about message arrivals. The round-trip message latency reported by the authors was found too high for point-to-point communication in such a system, despite running it on bare metal with the minimum needed software overhead.

Another approach for leveraging interrupts, using the GIC, has been applied in the SCC port of distributed S-NET [12], a declarative coordination language for many-core chips. The port is based on an asynchronous message-passing library: Interrupts are trapped by the Linux kernel and then forwarded to the registered userspace process in the form of a UNIX signal, which is the idea reused in this paper. Using a similar round-trip experiment as in [9], the authors confirm the high latency of inter-processor interrupts. Moreover, the latency they observe is even higher than in [9], mainly because of a necessary context switch before delivering a signal to the registered userspace process. A direct comparison with RCCE, the native SCC message passing library based on polling [8], has shown that IPIs are far less efficient in terms of latency for point to point communication.

Despite being costly for point-to-point message passing, IPIs can be used for asynchronous collective communication with an acceptable cost, as we show in this work.

IV. Broadcast Based on Interrupts

This section describes the design and implementation of our broadcast library based on inter-processor interrupts (IPI). First, we give an overview of the underlying hardware mechanism for sending parallel interrupts. Then we briefly describe OC-Bcast, a polling-based broadcast algorithm for the SCC, and explain how we have adapted it to use interrupts instead.

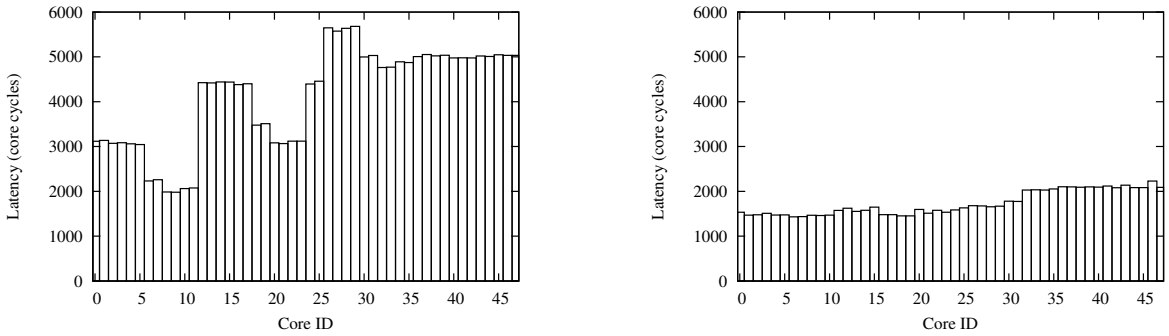
A. Interrupt Hardware on the SCC

Using the basic IPI mechanism on the SCC, a core can send an interrupt to another core by writing a special value to the configuration register of that core. This generates a packet which is sent through the on-chip network to the destination core. Although this mechanism is simple and straightforward, it lacks some essential features. For example, the identity of the notifier is unknown and it is possible to send only one interrupt at a time.

Fortunately, the SCC has an off-chip FPGA, which allows for adding new hardware features. An extension to the basic IPI mechanism has been provided by Intel, which comprises a set of registers for managing IPI (request, status, reset and mask). As a consequence, a core can send an interrupt to up to 32 other cores in just one instruction, by writing an

[1]The GIC is available starting with sccKit 1.4.0 and is located on the FPGA.

[2]Strictly speaking, it is possible to communicate asynchronously using a dedicated polling thread, but this solution wastes CPU cycles and energy.



(a) With access to GIC registers

(b) Without access to GIC registers

Fig. 2: Latency of broadcasting an interrupt at the kernel level

appropriate bit mask to its request register[3]. The work of generating interrupt packets is completely delegated to the FPGA interrupt controller.

To test whether the FPGA interrupt controller actually delivers multiple interrupts in parallel, we have performed the following experiment: A core sends an interrupt to all cores (including itself), by issuing two instructions which write a mask of "1"-s to its request register on the FPGA. Then, the core measures the time until it receives its own interrupt. The results, given in Figure 2a, indicate a significant difference in latency observed by different cores, ranging from about 2000 to almost 6000 core cycles (cf. VI-A for setup details). Further experiments have confirmed that this difference grows as a function of the number of cores that the interrupt is sent to – it is barely noticeable for less than 20 cores, but then starts to increase rapidly.

The experiment presented above could lead us to the conclusion that parallel notification using interrupts scales poorly, but further investigation explains this result. Namely, upon receiving an interrupt, there is a fixed set of steps a core should perform. This includes reading from the status register, to determine the sender, and resetting the interrupt by writing to the reset register. Since all the registers related to interrupt handling are on the FPGA, access to them is handled sequentially. When an interrupt is sent to many cores at once, they all try to access their interrupt status register at the same time, but their requests contend and are handled one after another, which explains the observed performance loss. We believe that a proper on-chip implementation of interrupt registers would eliminate this problem, since they could be accessed in parallel. To confirm that the reason for bad scaling of the interrupt mechanism is contention on the FPGA, we have repeated the same experiment, but this time deliberately avoiding the FPGA registers, except on the sending core. In Figure 2b we see that the times measured across the cores are very similar and close to 2000 core cycles. Slight differences in latency are easy to explain. Namely, the FPGA is connected to the mesh via the router between tiles (2,0) and (3,0) (cf. Figure 1), so the round-trip time to the FPGA is shorter for cores closer to this router. Next, it takes slightly more time for cores 32 to 47 to receive their interrupt. This is because, as already described, it is possible to send at most 32 interrupts by issuing a single instruction. Therefore, when broadcasting an interrupt, a core first broadcasts to cores 0 to 31 in the first instruction, and then to the other cores, which results in slightly higher latency.

Another set of experiments, as well as comparisons with results of other authors [12], confirmed that the latencies presented in Figure 2b are practically indistinguishable from the latency of sending point-to-point interrupts (about 2000 cycles). This implies that the cost of notification using interrupts is practically constant with respect to the number of cores notified. However, as we have described, sequential access to the off-chip registers for interrupt handling slows down the whole process in the current implementation on the SCC. Still, from Figure 2a we can see that even with this effect, broadcasting an interrupt to the 48 cores is only about 3 times more expensive than sending a point-to-point interrupt, making this mechanism interesting for use in group communication.

B. OC-Bcast Based on Interrupts

Now we describe how the SCC interrupt hardware presented above can be used to perform asynchronous broadcast. As the base, we used OC-bcast [10], an optimized on-chip broadcast algorithm built on top of one-sided put and get primitives. The principle of OC-bcast is the following: a broadcast k-ary tree is formed, with the sender as its root. The sender puts the message in its MPB and notifies its k children, which then copy the message to their own MPBs in parallel and notify the parent that it can free its MPB. The children repeat this process with their children, until all the cores have got the message. The value of k is configurable. Obviously, higher values of k offer more parallelism, but they can lead to contention on the MPB, which can cancel out the gain obtained by the increase in parallelism. This is not a problem for the SCC itself (OC-Bcast with $k = 47$ even gives the lowest latency for some message sizes), but can be an issue at large scale.

However, in its original flavor, OC-Bcast uses MPB polling for notification. Each child has a flag in its MPB that it polls when waiting for a message. This means that the children

[3]The upper limit of 32 is merely a consequence of the 32-bit memory word on the P54C

cannot be notified in parallel about the existence of a message, since the parent can write only one flag at a time, which was mitigated to some extent by using a special notification tree. This problem can be addressed by parallel interrupts. The modified algorithm can be summarized as follows:

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

C. Implementation

To implement the modified OC-Bcast, we have developed a userspace library for interrupt handling, following the idea given in [12]. Namely, a userspace process can register itself with a special kernel module. Every time an interrupt from another core is received, the kernel module sends a real-time UNIX signal to the registered process, which triggers a user-provided handler. We have opted for real-time signals because they can be queued if there is more than one signal pending. This way, we ensure that every interrupt is converted to a signal and the algorithm can be written entirely in userspace.

A drawback of this approach is a performance loss already observed in [12], since it increases the end-to-end delay of sending interrupts. Namely, the numbers presented in Figure 2b show only the latency until the receiver's kernel handles the interrupt. To propagate it to a userspace process in the form of a UNIX signal, a context switch is necessary, which significantly increases the cost. Nevertheless, we have adopted this approach for two reasons. Firstly, such an implementation changes only absolute numbers and does not prevent us from observing changes in performance resulting from design-level decisions. The same algorithm could be implemented in the Linux kernel or directly on bare metal, which completely avoids UNIX signals and context switching. Secondly, our library is easy to integrate with RCCE and the accompanying tools, which makes it convenient for other researchers willing to use inter-processor interrupts without significant effort.

V. Managing Concurrent Broadcast

OC-Bcast was initially designed in the context of SPMD applications, where a core has to explicitly call the broadcast function to participate in the collective operation. As a consequence, a core is involved in only one collective operation at a time. Using interrupts in OC-Bcast allows us to move to a more general model where broadcast operations can arbitrarily interleave at one core. In this section, we study how to efficiently manage this aspect.

The algorithm described in Section IV-B has to be modified to allow asynchronous broadcast operations issued by different cores. Indeed, without modifications the algorithm would be prone to deadlocks. A simple scenario can be used to illustrate a deadlock situation. Consider two cores c and c' that try to broadcast a message concurrently, with c' being a child of c in the tree where c is the root and the opposite in the tree where c' is the root. Core c' cannot copy the message that c is trying to broadcast in its MPB because it is busy with its own message. Core c' will be able to free its MPB when it knows that all its children have copied the message. However c cannot get the message from c' either, because it is in exactly the same situation as c'. There is a deadlock.

To deal with this problem, a simple solution would be to use a global shared lock to prevent multiple broadcast operations from being executed concurrently. In this case, the problem becomes equivalent to broadcast in the SPMD model and no further modifications to OC-Bcast are necessary. However, this would limit the level of parallelism and prevent us from fully using the chip resources.

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

Algorithm 1 presents the pseudo-code of this solution for a core c. In the presented algorithm, we do not put any requirements on the tree structure. We only assume that a predefined deterministic algorithm is used to compute the broadcast trees. Thus, during the initialization, each core is able to compute the tree that will be used by each source (line 7). Furthermore, if a message is larger than the available MPB, it is divided into multiple chunks.

For the sake of simplicity, the pseudo-code is not fully detailed. It only illustrates the important modifications that are made to avoid deadlocks. We define three functions as an interface to the algorithm described in Section IV: $OCBcast\_send\_chunk(chunk, Tree)$ initiates the sending of the chunk $chunk$ in the tree $Tree$; $OCBcast\_receive\_chunk(chunk, buf, src)$ allows to get $chunk$ from the MPB of core src in buf, buf being either the MPB of the caller or a memory region in its off-chip private memory; $OCBcast\_forward\_chunk(chunk, Tree)$ is used to forward a $chunk$ in the tree $Tree$. Contrary to $OCBcast\_send\_chunk()$, $OCBcast\_forward\_chunk()$ assumes that the chunk is already in the MPB of the sender. In the pseudo-code, a chunk includes not only payload, but also some meta-data, *i.e.*, the id of the core that broadcasts the message ($chunk.root$) and the id of the message the chunk is part of ($chunk.msgID$).

As mentioned before, we allow a core to receive chunks directly in its off-chip private memory when its MPB is busy with another chunk that is being sent (line 17). Thus, the

Algorithm 1 Asynchronous broadcast algorithm (code for core c)

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

 5: initialization:
 6:     define deliver_chunk() as the IPI handler
 7:     for coreID ∈ 0...n do compute Tree_coreID

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

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

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

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

sender can free its MPB. The chunks that the core is supposed to forward to other cores, are stored in a queue (lines 25-28), that is flushed when the MPB becomes available (line 15 and line 23). Note that to ensure fairness, if the MPB is free at the time the core receives an interrupt but some chunks are already queued to be forwarded (line 17), the chunk is received in the private memory and added to the queue. Thus, a chunk cannot overtake another chunk that has been in the queue already for some time. However, if no chunk is in the queue and the MPB is available, the chunk is first copied in the MPB to limit the number of data movements between the MPB and the private memory that could harm the performance of the broadcast operation [10].

VI. Evaluation

In this section we evaluate our broadcast algorithm. After describing the system parameters used for our experiments, we measure the latency of the presented broadcast algorithm and compare it with that of OC-Bcast. Then we show how the algorithm behaves with different values of k and with more cores broadcasting at the same time.

Message Size (Number of cache lines)	1	32	64	128
OC-Bcast	44.0 μs	76.1 μs	112.6 μs	189.8 μs
Asynchronous broadcast	40.2 μs	75.5 μs	118 μs	196.7 μs

TABLE I: Comparing the latency of synchronous broadcast (OC-Bcast) and asynchronous broadcast for different message sizes.

A. Setup

We have performed the experiments under the default SCC settings: 533 MHz tile frequency, 800 MHz mesh and DRAM frequency and standard LUT entries. We use sccKit 1.4.1.3, running a custom version of sccLinux, based on Linux 2.6.32.24-generic. The kernel of every core runs the special kernel module for converting interrupts to UNIX signals, described in Section IV.

B. Experiments

The first experiment measures the latency when messages of different sizes are broadcast from one core (core 0 in this case). We fix the value of k to 47 (see Section IV), which enables us to obtain the highest level of parallelism when sending the interrupts and reading from the MPB. Due to space constraints, we do not consider other values of k in this experiment.

Table I compares the obtained latency with that of OC-Bcast[4]. The two algorithms have very similar latencies with these settings. This confirms that the interrupt hardware on the SCC is useful for designing asynchronous collective operations, even though its latency is high for point-to-point communication, as pointed out in other studies [9], [12].

It is interesting to notice that the latency of the asynchronous broadcast algorithm increases faster as a function of the message size. This is because of a higher level of MPB contention. More specifically, it is pointed out in [10] that too much parallelism in accessing the MPB can impair performance. In OC-Bcast, notifications are propagated using a binary tree, which results in less overlapping accesses to the MPB of the sender than when a parallel interrupt is sent. This shows that extremely high values of k might be inappropriate at large scale because of the contention effect.

In the second experiment, we change the output degree of the broadcast tree (k) and the number of sources, that is, the number of cores broadcasting in parallel. Each source repeatedly broadcasts a 4 KB (128 cache lines) message from its private memory, without waiting for the other cores to receive the message, thus creating a message pipeline. This way we observe the throughput of the system, that is, the amount of data broadcast in a unit of time.

The result of this experiment is given in Figure 3. With a single source, the throughput decreases as k increases. The reason is the cost of polling flags (there are at most k flags to poll). To wait for an acknowledgment from its children, each parent has to poll k flags in its MPB and reset them afterwards. The variations in the performance can be explained by the fact

[4]The version of OC-Bcast considered here is slightly optimized with respect to the original paper which presents it [10].

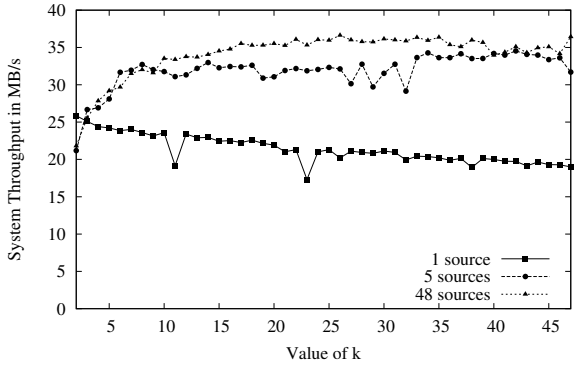


Fig. 3: Throughput of the asynchronous broadcast algorithm for different values of k and different number of concurrent sources

that a core does not control when it will be signaled. In fact, when a core is about to forward a received message to the children, it can get interrupted to receive another message. If this happens, the children have to wait, which introduces sporadic performance drops.

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

We can also see that the difference in throughput when broadcasting from 5 and 48 sources is not significant. This is because the system gets saturated. Based on the model presented in [10], the maximum bandwidth when copying data from a core's MPB to the off-chip memory is about 55 MB/s (assuming cache line prefetching implemented in software as in iRCCE [4]). Our algorithm achieves 68% of this maximum bandwidth.

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

C. Discussion

The presented experiments show two important properties of our asynchronous broadcast algorithm. First, in spite of being built on more general assumptions, its latency is comparable with that of the most efficient synchronous broadcast algorithm currently available for the SCC. Second, the algorithm manages concurrent broadcasts efficiently, even when all cores are broadcasting at the same time.

VII. Conclusion

In this work we have presented a novel asynchronous broadcast algorithm for the Intel SCC, which is based on RMA and parallel IPI. Our algorithm is derived from OC-Bcast, an optimized synchronous broadcast algorithm for the SCC. The evaluation of our asynchronous broadcast primitive demonstrates that the algorithm manages to efficiently deal with concurrent broadcast operations to achieve low latency and high system throughput. Comparisons with existing synchronous broadcast primitives also show that parallel IPI are of general interest to implement efficient group communications on many-core chips.

As future work, we plan to study the use of IPI and on-chip RMA operations for other group communication primitives on the Intel SCC. Especially, we will focus on group communication primitives that provide ordering properties, to implement replicated data structures.

References

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

[2] I. C. Urena, M. Riepen, and M. Konow. RCKMPI–Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). *Recent Advances in the Message Passing Interface*, pages 208–217, 2011.

[3] E. Chan. RCCE_comm: a collective communication library for the Intel Single-chip Cloud Computer. http://communities.intel.com/docs/DOC-5663, 2010.

[4] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525 –532, july 2011.

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

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

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

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

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

[10] Darko Petrović, Omid Shahmirzadi, Thomas Ropars, and André Schiper. High-Performance RMA-Based Broadcast on the Intel SCC. In *24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburg, PA, USA, June 2012. to appear.

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

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