Effective Communication for a System of Cluster-on-a-Chip Processors

Pablo Reble
Chair for High Performance Computing
JARA High Performance Computing
IT Center RWTH Aachen University
Seffenter Weg 23
52074 Aachen, Germany
reble@itc.rwth-aachen.de

Stefan Lankes
Institute for Automation of Complex Power Systems E.ON Energy Research Center
RWTH Aachen University
Mathieustrasse 10
52074 Aachen, Germany
slankes@eonerc.rwth-aachen.de

Fabian Fischer
Faculty of Electrical Engineering and Information Technology
UMIC Research Centre
RWTH Aachen University
52056 Aachen, Germany
fabian.fischer@rwth-aachen.de

Matthias S. Müller
Chair for High Performance Computing
JARA High Performance Computing
IT Center RWTH Aachen University
Seffenter Weg 23
52074 Aachen, Germany
mueller@itc.rwth-aachen.de

ABSTRACT

In this work, we analyze efficient communication methods for a grid of many-core processors in the absence of cache coherence. For this study, we build a multi-chip processor with 240 tightly connected cores and demonstrate its scalability. This processor is based on the Intel SCC, a cluster-on-a-chip research processor with 48 non-coherent memory coupled cores. Our new research system virtually extends the on-chip network of multiple SCC systems and provides new communication functionality for direct on-chip memory access. We analyze access patterns of different communication schemes and apply techniques to hide latency, such as offloading communication and software caching with relaxed consistency.

Keywords

Cluster-on-a-Chip, Low-level Communication, Emulation of on-chip Interconnect, Message Passing

1. INTRODUCTION

Effective Communication has been identified as an important part of creating future supercomputers with a performance of 10^{18} floating-point operations per second (exaFLOPS). In 2008, the first petascale supercomputers were realized with multi-core processor technology as a key driver. For a realization of exascale supercomputers in foreseeable future, new milestones in technology will be required regarding the sheer computing performance in the presence of even higher core counts.

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA
Copyright 2015 ACM 978-1-4503-3404-4/15/02 ...$15.00.
http://dx.doi.org/10.1145/2712386.2712393

In general, a processor design which follows the network-on-chip (NoC) paradigm targets communication in a scalable way. Based on this technology, tiled many-core architectures are an emerging trend in research. Especially focussing on the x86 landscape towards hundreds of cores per die, potential issue concerning the scalability of processor architectures is hardware cache coherence, which manifests in architectural overhead. A loss of cache coherence is a radical change of memory abstraction and will raise challenges to low-level software development, as data invalidation and transfer has to be explicitly controlled.

In this paper, we focus on processor architectures that target the integration of hundreds to thousands cores per die. In order to explore such architectures and to enable research into this direction, we introduce *vSCC*, a new research vehicle to evaluate scalability of the existing software stack for the Intel Single-chip Cloud Computer (SCC). The result is a runtime extension, which can be used to emulate a single *cluster-on-a-chip* processor with 240 cores. This allows to analyze many-core system properties today and thereby guide the development of future systems. Regarding the physical setup of *vSCC* we connect physical distributed SCC devices to a single virtual many-core processor. The hardware of our unique research system consists of a server that is equipped with multiple PCIe expansion cards, as illustrated in Fig. 1.

The basic concept that we have followed is waiving transparent inter-device communication and extending a cluster-on-a-chip architecture by new instructions. For such a communication scenario, this creates the possibility of efficient data movement. This integration is based on the idea of extending the architectural support for message passing of the Intel SCC to a host assisted communication path. Specifically, we extend the functionality of a many-core communication layer for the realization of a more efficient exchange of data with a communication task, which is running on the host and part of the SCC's software stack.

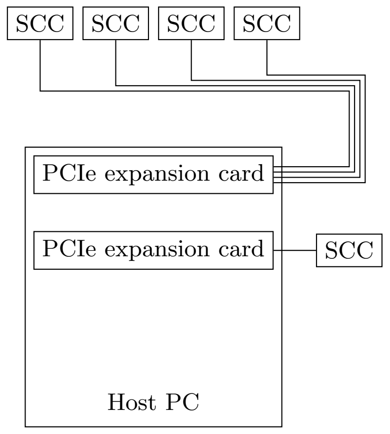


Figure 1: Physical setup of our unique research system, which consists of five Intel SCC devices connected through PCIe expansion cables to a single host

The main contributions of this paper are the following:

- We discuss the design of a system of cluster-on-a-chip processors with 240 cores
- We follow a reverse acceleration approach and effectively offload inter-device communication
- All together, our techniques enable effective hiding of high latency communication path through means of system software support and recover 24 % of on-chip communication performance

The paper its structured as follows. The next section covers related work for the approach of offloading communication and connecting multiple SCC systems. In Section 3, we describe the general concept and implementation of host assisted communication for the communication scenario that is studied in this paper. In Section 4, we analyze the performance by using selected applications and give insights on the communication overhead. We use RCCE[1] and available applications to demonstrate the quality of our implementation. Section 5 summarizes this paper and gives an outlook for future work.

2. RELATED AND PREVIOUS WORK

Tiled many-core processors are an active field of research, for instance the Tilera processor family or the SCORPIO project [2, 5]. Relevant processor prototypes of experimental architectures are commonly assembled to a PCIe device in order to create a flexible research vehicle, whereas the Intel SCC represents another example. Due to its hardware dimensions, the SCC processor is located on a separate mainboard which holds an FPGA that is connected to a PCIe expansion card.

In general, the concept of using PCI Express (PCIe) as a fabric interconnect for computing units is not limited to research projects. A prominent example for High Performance Computing (HPC) is the use of GPUs and CPUs as direct PCIe coupled processing devices. In the field of HPC computing, the Xeon Phi coprocessor is another example of a many-core processor that is assembled to a PCIe device.

Our research shares the basic idea of the reverse-acceleration approach [11]. The approach is named reverse-acceleration, because it targets offloading communication between throughput optimized cores to a latency optimized computing core, instead of offloading compute intensive tasks to throughput optimized cores. The reverse-acceleration approach has been successfully applied as a work-around for the Xeon Phi coprocessor [8]. A hardware limitation of Xeon based host systems avoids a good direct communication performance between two PCIe devices. The solution for MPI based applications is a proxy task that is running on the host and acts as a message broker to the communication library [12]. This work follows a similar approach compared to our work regarding communication offload, where the general goal is to accelerate communication of many-core systems. However, the main contrast is that we target a low-level communication path on data transfer layer and extensions to systems software, instead of a modification of the communication library.

2.1 Intel SCC

The Single-chip Cloud Computer (SCC) experimental processor [15] is a *concept vehicle* created by Intel Labs as a platform for many-core software research.

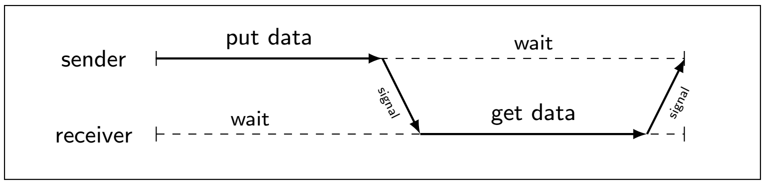
The architecture of the experimental processor has 48 classic in-order P54C based cores without support for hyperthreading, SIMD extensions or cache coherency. Two cores are located on a single tile and share a router, which connects the tile to a two-dimensional mesh interconnect. Rather than being competitive in absolute computing performance compared to commercial coprocessors or accelerators, the SCC implements a new on-chip communication concept for x86 based many-core processors. This concept consists of explicit on-chip data movement and can be seen as hardware support for message passing. In the default configuration, 48 embedded Linux instances, one per core, are running on the coprocessor in private memory regions, which are distributed over four memory controller.

Attributes of the tiled many-core architecture are multiple coherency domains per chip and hardware support for message passing, integrated to the processor architecture. This results in an inherent scalability and flexibility of the many-core architecture itself. From a manufacturability perspective, more transistors per chip can be used to add more tiles to the processor without huge effort in validation and redesign.

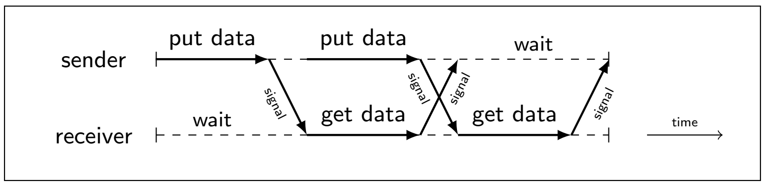
The idea of a virtual expansion of an on-die network across systems to emulate a CPU with up to hundreds of cores has been first mentioned by Gries et al. [6] as an application for high-speed serial I/O connectors.

We have presented the realization of this approach, without any modifications to the SCC hardware in [13]. In this previous work we reached the goal of a transparent connection of two SCC devices by routing on-chip packets through the host system. For this small virtual *cluster-on-a-chip* processors, we have identified the default communication scheme of RCCE as a major performance bottleneck. To hide communication latency of a transparent inter-device connection, the *local put* communication scheme of RCCE was replaced.

[1]a low-level communication library for the Intel SCC, available at: `github.com/Intel-SCC/RCCE`



(a) RCCE `send` and `receive`



(b) iRCCE `send` and `receive` (with pipelining)

Figure 2: Timely behavior of basic blocking communication protocols

This exchange of the communication scheme to *remote put* required fundamental changes to the communication protocol, which is described in detail in our previous work. We have finally developed a new communication protocol, which effectively uses the hardware support for message passing also for inter-device communication. As a result, a RCCE session with moderate communication traffic can be scaled up to 96 cores on the virtual extended on-chip interconnect. Only minor modifications to the hardware abstraction layer were necessary, such as a mapping of remote on-chip memory and extending the RCCE ranks in a linear way up to 96.

Due to hardware limitations of the SCC processor, the existing prototype prevents a stable use for rising inter-device communication, which is a result of three or more tightly coupled SCC devices. To further explore the scalability of the SCC processor architecture and related software solutions, we present in this paper the vSCC architecture with more than 200 cores. To this end, we develop a cooperation between device and host, which in fact waives full transparency and extend the SCC architecture by new functionality. This functionality is added to the communication task and enables the application of classic software optimization techniques, such as caching or write combining.

Our research shares similarities to a control of dedicated cores, called copy cores, which have been used in related work to accelerate a memory copy operation for the SCC [16].

2.2 RCCE family

RCCE has been developed by Intel Labs as a light-weight communication environment for the SCC research processor [9]. It provides a low-level interface to abstract hardware details of on-chip communication. Because of an intended bare-metal use without operating system support, it purely relies on busy waiting and flag based synchronization.

Rather than explicitly flush shared data and selectively invalidate caches, a programmer can follow the message passing programming paradigm to develop applications by using a two-sided communication interface, called *non-gory*. Nevertheless, it is a low-level API with a limited functionality, compared to a fully featured communication environment, such as MPI.

The reference implementation of RCCE has been implemented as a layered approach. This includes a basic one-sided interface, called *gory*, which can be seen as a hardware abstraction layer. Applications can also use this low-level API instead of the *non-gory* interface for direct on-chip memory access or a combination of both. Such programming is complex, because it is specific for each many-core architecture and hard to debug. A direct use of the gory interface mainly targets applications where a high predictability is essential, for instance real-time applications in a bare-metal framework.

In its nature of a layered approach, the communication protocol invokes *gory* functions for the realization of send and receive functionality, similar to the described applications. Blocking communication through `RCCE_send()` and `RCCE_recv()` works as illustrated in Fig. 2a.

First, the sender puts the message into its local communication buffer. Second, for the indication of this event, the sender toggles a flag at the receiver's side. Finally, the sender waits for the indication that the receiver has copied the message to its private memory. If a message does not fit into the MPB, the message is internally split into smaller messages and transferred consecutively. In general, blocking communication means that the send function returns, if the receive function has been completed.

The default communication protocol implements a *local-put* and *remote-get* communication scheme. A strength of this communication scheme is that each core exclusively writes to its local communication buffer, which simplifies the synchronization model.

iRCCE has been developed at RWTH Aachen University as a non-blocking extension for RCCE [4]. Moreover, our communication extension supports sophisticated communication protocols and introduces optimizations [3]. Figure 2 compares the timely behavior of a blocking communication, with and without pipelining. The diagram illustrates the communication progress and indicates a previous completion of the pipelined protocol. The pipelined protocol of iRCCE introduces additional overhead by using a finer synchronization granularity, but provides the advantage of interleaving put and get operations. Consequently, this protocol can accelerate point-to-point communication, if the internal packet

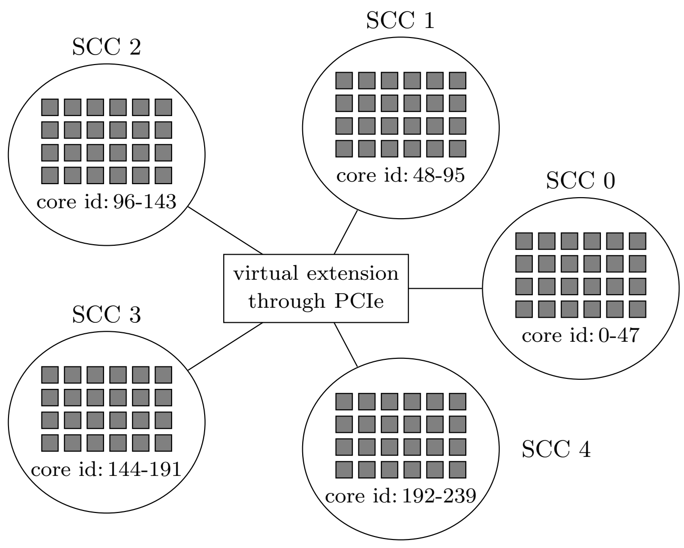


Figure 3: Topology of vSCC consisting of five devices with 240 physical core ids

size is chosen appropriately. Resulting communication performance in terms of throughput is detailed in Section 4.

We have summarized the communication protocol of the RCCE family here in detail, because we present in the next section a new communication concept for a grid of cluster-on-a-chip processors, that relies on this basic kind of communication. Furthermore, we compare the achieved results for inter-device communication to on-chip communication.

2.3 Limitations of existing Prototype

An extension of the SCC host system driver, which tunnels a proprietary network-on-chip protocol through a PCIe connection, has been presented in a previous paper by some of the authors [13]. This previous prototype implements a simple routing, which leads to a transparent extension of the on-chip interconnect. In this scenario the communication task which is running on the host can be seen as a proxy for the proprietary on-chip protocol of the SCC. Beside this essential functionality to forward fixed size messages, no further optimizations have been added to the communication task.

Nevertheless, remote traffic had to be analyzed by a so called communication task, which is running on the host. The feature to distinguish between different types of communication was already supported by the original SCC driver provided by Intel Labs, but the communication task was restricted to handle inter process communication on the host.

For the realization of a transparent inter-device communication path, RCCE had to be modified. In [13], we have developed a new communication protocol and integrated to iRCCE, which implements a remote put communication scheme. Experiments with the previous prototype were limited to this scheme, because the good performance of the *remote put* scheme relies on the option of generating automatic write acknowledges for requests that target off-chip memory. This option has known stability issues, which prevents a tight coupling of more than two SCC devices and works only for applications with a moderate inter-device communication.

In the following, we will show that additional functionality in system software, such as a virtual DMA controller, can waive the described restrictions and enable a further analysis of the SCC hardware and software.

3. COMMUNICATION CONCEPT

The physical communication topology of the Intel SCC architecture represents a two dimensional mesh. From a programmers perspective, a message passing application uses ranks to identify specific processes. Running a RCCE application on the SCC, the processes are numbered in a linear way and mapped to physical cores that are sorted in a descending order according to their id. As a result, a neighboring communication rank does not guarantee a small communication distance. This mapping of physical core ids to ranks is a common abstraction for the programmer, but it does not necessarily represent an ideal communication scheme.

For the vSCC architecture this mapping represents a challenge, because a third dimension is added to the communication topology, whereas only a single physical link at (x, y) coordinate $(3, 0)$ exist. To describe the coordinates of a vSCC core the triple (x, y, z) is used, as illustrated in Fig. 3. As we use the device number as z coordinate, the inter-device communication with a higher latency ($\sim 10^4$ core cycles) is in z direction. A communication path in x or y direction has a relatively low latency (~ 100 core cycles), due to the on-chip interconnect [14]. Nevertheless, the communication path for vSCC in z direction corresponds to a routing functionality on the data transfer layer.

The challenge is, that the virtual extension does not necessarily change the mapping of RCCE ranks to physical cores. First all cores of the first device are assigned to RCCE ranks in a linear way, which is continued to a second device starting with id 48. In Fig. 3 the topology of vSCC is illustrated, whereas each rectangle represents a tile and circles represent devices, that are connected through PCIe to a tightly coupled system.

The basic communication layer of RCCE, also called *gory*, abstracts the hardware details of communication, such as explicit access to on-chip memory and flushing of caches, by providing a simple one-sided interface through `put` and `get` functions. This RDMA functionality is further provided by the vSCC architecture. In the next paragraph, we summarize the hardware support for message passing of the SCC architecture.

3.1 Hardware support for message passing

RCCE developers introduced the term MPB, which is commonly used in synonym for the software controlled on-chip memory for the Intel SCC. Within the scope of this work, we will use different terms to describe different regions of the software controlled on-chip memory. We use the term local memory buffer (LMB) to describe the software controlled on-chip memory per tile, MPB for the message passing buffer, and SF for *synchronization flag* region. The resulting relation is that SF and MPB share the LMB.

The Intel SCC architecture handles private and shared data differently, thereby that for shared data a new memory type has been introduced (MPBT). Memory that uses this new memory type can be combined with *write through* or *uncached* memory configuration. For this shared data in *write through* configuration, only the first level cache is activated and all other caches are bypassed. Hereby, the tagged cache-lines in first level cache can be invalidated by a single instruction (`clinvd`). Additionally in both cache memory configurations a transparent write combining buffer is activated to accelerate write bursts to shared memory.

Our approach is to extend this concept to the implementation of a cache that is maintained by the communication task. For the observed communication scenario of RCCE, data movement that targets on-chip memory can be divided into two categories: *synchronization* and *communication*. Specifically, access to flags, which are for example used to coordinate data transfer, and buffer access for point-to-point communication are handled differently.

Similar to the differentiation between shared and private data of the SCC memory model, our approach to accelerate inter-device communication is to handle remote data access differently. In order to realize this differentiation of inter-device data transfer, the memory regions have to be known to the communication task. Specifically, each rank has to register start address and length of the communication buffer to the communication task. As a result, the task can classify incoming requests and handle them in a different way. Flag access[2] bypasses all transparent buffers of the communication task and is forwarded to another device. Even if remote read operations to synchronization flags are forwarded without caching, write requests can be directly acknowledged immediately, whereas the acknowledgement of read requests has to be delayed until the requested data is available.

Another important attribute for remote memory access is the fact that continuous read operations are used by RCCE family to transfer data with a predictable access pattern. For our communication scenario this attribute generates the possibility of prefetching data with a high accuracy. As a result, the communication task holds a complete or partly copy of the remote MPB to handle following remote read requests with a lower latency. In order to implement relaxed memory coherence in an efficient manner, we use information on the communication protocol to avoid data inconsistency. Such that the sender that writes to a local MPB explicitly invalidates the outdated part of the host copy explicitly.

[2]RCCE uses read operations exclusively to local flags

3.2 Communication Task

Our approach to overcome the described limitations is adding new functionality to the inter-device communication path of vSCC. This includes new instructions to control the consistency of an intermediate buffer with the goal to hide communication latency.

For our prototype, the communication task has been implemented as an extension of a background process, also called *daemon*, of the device driver that is part of the SCC's software stack. Because the host is connected to multiple devices, our communication task consists of multiple threads on kernel level. This multithreaded *daemon* directly invokes read and write operations to the system interface of an SCC device. Similar to the original version of the SCC driver, a physical DMA controller on the host is invoked for communication through PCIe to the device.

The main difference to our previous work is that the communication task is no more restricted to transparent routing of memory requests. Moreover, we added memory mapped register[3] to emulate instructions, that can be used to control the support of communication. As a result, the communication task can cache and prefetch data from another device, whereas the process which is running on the SCC can control the data consistency.

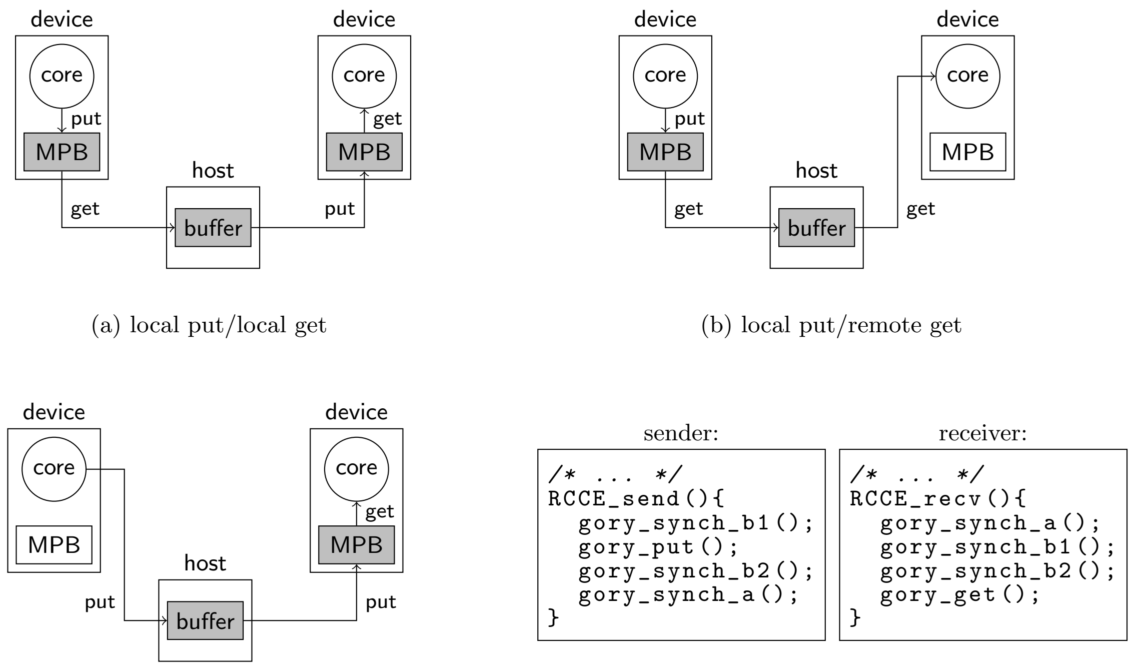
This kind of control has the advantage, that information of the communication protocol can be used for optimizations. For example, a message of 1 kB is transferred from one device to another device by invoking cooperative `send` and `recv` functions. The offloaded communication task needs information on data location, size, and communication scheme of the message passing protocol to perform a copy operation. To accelerate different communication schemes, classic software techniques, such as prefetching and caching of remote data can be used.

If we assume a *local put/remote get* scheme, the sender copies the message from private to shared buffer. Next, a synchronization flag is set at the receiver, that the copy operation has been finished. We loop through this flag to inform the communication task, that the data on sender side is available. Additionally, the sender has to inform the communication task on the location of the message, which is handled through a write operation to a memory mapped register. As a result, the communication task can begin to copy the message to an intermediate buffer and after a warm-up phase answer remote memory requests of the receiver in parallel. This method represents a prefetching technique, which has been successfully applied to the described communication scenario.

3.3 Host accelerated Communication

Figure 4 compares the data flow of different communication schemes with host invocation. Additionally, Fig. 4d holds a code snippet, which depicts the synchronization points for the alternative communication schemes. For the default *local put* scheme of RCCE, synchronization points `a` and for the *remote put* scheme synchronization points `b1` and `b2` are active.

[3]A memory mapped register executes a specified operation, e. g. a frequency change, and is controlled through read and write access to a specific address



(a) local put/local get

(b) local put/remote get

(c) remote put/local get

```
/* ... */
RCCE_send(){
  gory_synch_b1();
  gory_put();
  gory_synch_b2();
  gory_synch_a();
}
```

```
/* ... */
RCCE_recv(){
  gory_synch_a();
  gory_synch_b1();
  gory_synch_b2();
  gory_get();
}
```

(d) source code

Figure 4: Comparison of inter-device communication schemes for message passing (left sender, right receiver)

The basic approach of accelerating inter-device communication is offloading data movement to the host. This introduces overhead, which is another copy operation that extends the two way copy scheme from RCCE to a three way copy scheme, as illustrated in the graphic. Instead of copying data from private to shared buffer on sender side and back to private buffer on receiver side, a communication task is embedded performing an intermediate copy operation.

The communication task holds a buffer, that can be either used as a cache (Fig. 4b), as a write combining buffer (Fig. 4c), or as an internal buffer (Fig. 4a). For all 3 configurations, we will further detail the communication scheme.

Next, the inter-device communication progress is described for a protocol that supports a *remote put* scheme. First the sender, which is located on the left in the graphic, writes the message directly to the host located intermediate buffer. Next, the communication task copies the data in a certain granularity from its intermediate buffer to the MPB of the remote device. This behavior is equivalent to a write combining buffer.

For the *local put/remote get* scheme, the offloaded communication task handles the data transfer between device and host. First, the sender writes the message to its local MPB and triggers the communication task to copy the data to the intermediate buffer. Next, the receiver reads the message from the intermediate buffer on the host, which corresponds to a software cache, as illustrated in Fig. 4b.

Both communication schemes are possible in a kind of transparent mode, whereas the intermediate buffer is mapped as the remote on-chip buffer and memory requests are directly performed. Since the intermediate buffer represents a non-coherent copy of the message passing buffer, update and invalidation of data have to be controlled in an explicit way. Update and invalidation operations represent extensions of the instruction set of the SCC, which we have realized by a new set of memory mapped register.

As a result of these additional operations, different memory models are possible for optimization. In contrast to the described variants of host assisted communication, which can be realized in a transparent or non-transparent way, *local put/local get* can not be realized in a non-transparent way. A main advantage of this new local-access communication scheme is the fact that sender and receiver copy data to on-chip memory. The communication task performs the copy operation, and can be seen in this configuration as a virtual DMA controller.

Figure 5 illustrates the functionality that we have developed for vSCC. It is a classic design, comparable to DMA functionality of standard computer systems. Our implementations works through new memory mapped register, which we have integrated to the host driver. Three logical register can be used to pass information to the vDMA controller, about start of the memory location (address) amount of data (count) and coordinating tasks such as trigger the transaction start (control). A straight forward implementation would result in three remote memory accesses to control the virtual controller. For the Intel SCC continuous allocation of memory mapped register with an alignment of 32 B reduces this overhead because the architecture can fuse write operations with a write combining buffer (wcb).

Our prototype implementation uses busy waiting on host as well as on device side. After a core has finished program-

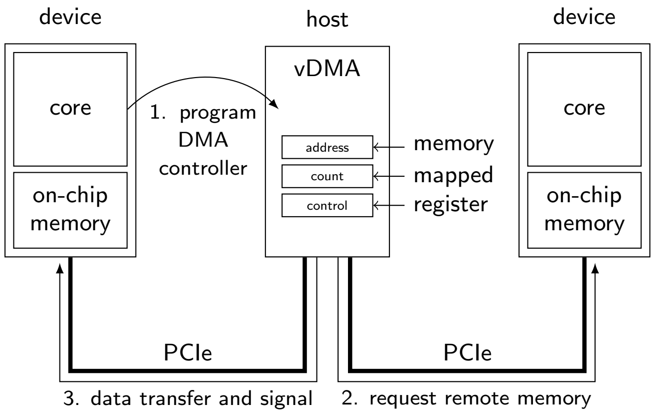


Figure 5: Functional layout of vSCC's DMA support

ming the vDMA controller, a core spins on a flag which is located in its on-chip memory to get the information that a memory copy operation has been completed. This method guarantees a low latency but prevents a core of doing useful work as long as the copy operation is in progress. Due to the fact that standard RCCE applications are limited to blocking communication, this implementation implies no limitations for the results of Section 4.

Because programming the vDMA controller represents a certain overhead, to recover low latency for small messages we have defined a threshold for a core to directly transfer data, which is about 32 B to 128 B dependent on the communication scheme.

4. EVALUATION

In this section, we present results of the implemented communication concept for a cluster of tightly coupled SCC systems. This includes benchmark results for selected RCCE applications and visualization of its communication pattern. For the measurement results, all active cores[4] have booted a Linux instance and are running a RCCE process.

Due to compatibility issues, the SCC systems that have been delivered by Intel Labs to users of the MARC community include a host system. Because the chassis of the host system has only space for a single-slot PCIe expansion card, a connection of more than one SCC device is not possible. Our experiences and experiences of other members of the MARC community have shown, that alternative hardware of the host system, which was necessary for an installation with multiple SCC systems, has to be carefully chosen. We use a two socket Xeon server (Intel S2600CW board) that has been equipped with a single port and a four port PCIe expansion cards (OSS-HIB5-x4). In this unique configuration, we were able to connect 5 SCC devices to a single host. However, we had to face some reliability issues.

The Intel SCC has been developed as a research system and is not a fully tested product. Regarding an installation that consists of multiple SCC devices, the probability for a core failure increases. For our installation, the situation occurs frequently that not all 240 cores are available at startup.

[4]running with (core/mesh/memory) frequencies in MHz: (533/800/800)

In the default configuration of the SCC research system, one independent operating system instance is booted on each core. As a result, a silent core failure does in general not have a negative impact on the stability or performance of the system. We have extended the startup script of RCCE thereby that it creates a new configuration file with all available cores before application run. RCCE takes this file as input to create the mapping of communication ranks to core ids.

4.1 Ping-Pong

As a fist step towards an evaluation of the developed software optimizations, we have compared different schemes of remote communication for the well-known Ping-Pong application. Figure 6 shows an overview on the throughput results of a Ping-Pong communication pattern in the diagram on the left hand side. The results are separated for on-chip and inter-device communication. Due to the fact that the cores of the SCC are based on classic P54C architecture, maximum on-chip communication throughput is about 150 MB/s. We compare the communication performance of RCCE without any pipelining to iRCCE, which introduces software pipelining with a static threshold of 4 kB.

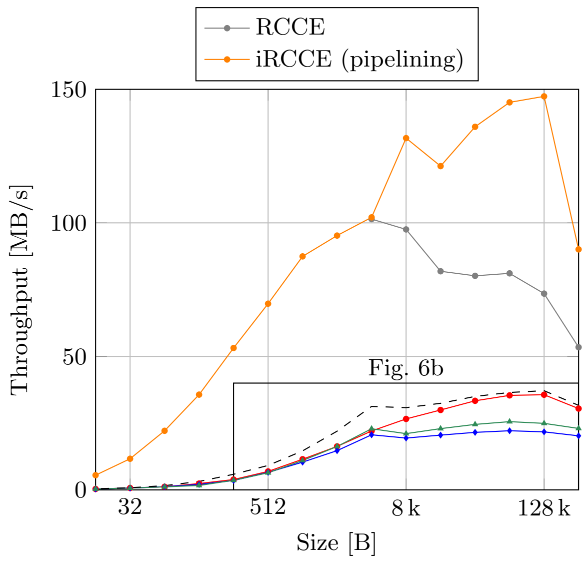
The diagram of Fig. 6b shows the inter-device communication throughput and represents a zoom in of Fig. 6a. Here, the throughput has been extensively improved by the presented optimizations in comparison to a simple packet routing. A direct control of the communication task leads to reasonable results.

The black curves in Fig. 6b represents the lower and upper bound for host accelerated default communication schemes. Because of the high latency from host to device or vice versa, the inter-device communication throughput will never be higher, than the variant which uses fast write acknowledges of the on-board FPGA. Moreover, the black dashed curve represents an upper limit because the hardware acceleration is not scalable for up to 240 cores. The obtained throughput results for offload of communication to the host are promising. We obtained a maximum throughput which is 71.72 % of the limit, for the worst case inter-device communication scheme (*local put/remote get*).

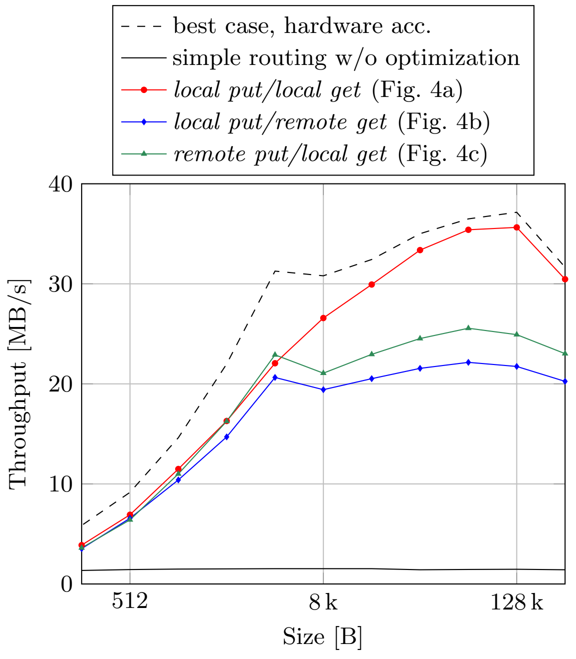
For all variants of inter-device communication, a drop down of the throughput for a message size from 8 kB can be observed. This value is the threshold, where a message does not completely fit into the MPB [5]. Consequently, the data transmission has to be split into two transfers, whereas the second transfer contains a small payload. This effectively reduces the resulting throughput. The gap between green and blue curve of Fig. 6 indicate, that *local put* scheme better suits a low-latency interconnect, which corresponds to local communication.

Our measurements show, that point-to-point communication throughput of the *local put/local get* scheme is close to the hardware accelerated version. Furthermore, the slope at 8 kB of the hybrid local communication pattern could be removed. This is the result of an optimization that we applied to the communication pattern. For larger messages, sender and receiver can progress communication in parallel by interleaving put and get operations to local on-chip memory. Here, the communication task can introduce a pipelining effect.

[5]The Local Memory Buffer of 8 kB holds the MPB and flags for synchronization



(a) Ping-Pong Throughput (on-chip and inter-device)



(b) Details of inter-device communication

Figure 6: Performance comparison of point-to-point communication for vSCC

4.2 Application Benchmark

We have evaluated the scalability of our system with multiple SCCs by using the BT Benchmark from the NAS Parallel Benchmark (NPB) suite [1]. Mattson et al. ported the BT benchmark from the well-known NAS parallel benchmark suite to RCCE [10].

Figure 7 shows absolute performance results for the BT benchmark in class C problem size ($162 \times 162 \times 162$). This problem size is suitable for the vSCC with 240 cores, as each core has a peak performance of 533 MFLOP/s. A resulting theoretical peak performance for the grid of SCC processors with 225 cores is 120 GFLOP/s. Due to the applied work distribution method and communication pattern 225 represents the maximum configuration, since the application can only handle a number of processes, which is a square number.

Overall, we see a good scalability of the application with host accelerated inter-device communication. As in the given case, for an application without specific topology awareness, the benchmark results demonstrate good performance of the communication extensions that are presented in this paper. To indicate the necessity of our optimizations, the diagram of Fig. 7 holds the results for optimal and worst configuration of inter-device communication. Lower black curve and red curve of Fig. 6 show the corresponding point-to-point communication throughput, which is a good performance indicator for the given application because only `send` and `recv` functions are used.

The NPB BT application uses a communication pattern which is based on locality. Message passing applications need an appropriate placement of processes to nodes of a cluster to achieve a good performance, especially for heterogenous networks. In other words, applications should prefer connections with high throughput for communication. This basic assumption is also true for a cluster-on-a-chip processor such as the Intel SCC, where processes are mapped to cores.

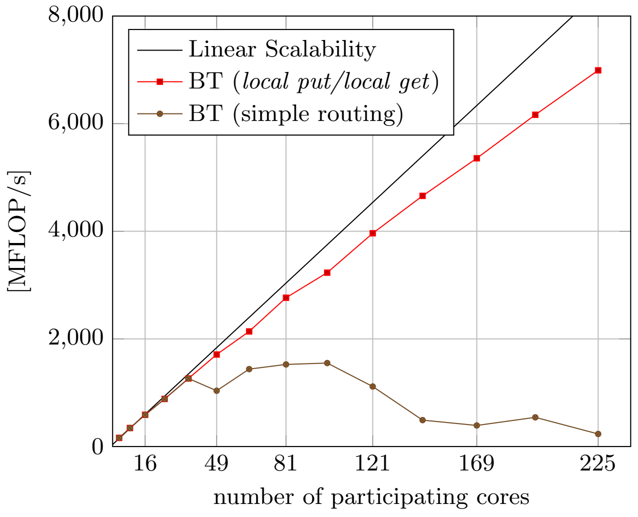


Figure 7: NPB Performance Results (class C)

Figure 8 visualizes the communication traffic for the investigated NPB BT application on vSCC, to further detail the communication performance of our new prototype. Each filled square of the diagram indicates a communication between two ranks (x is sender and y receiver), whereas dark means high and light means low communication traffic.

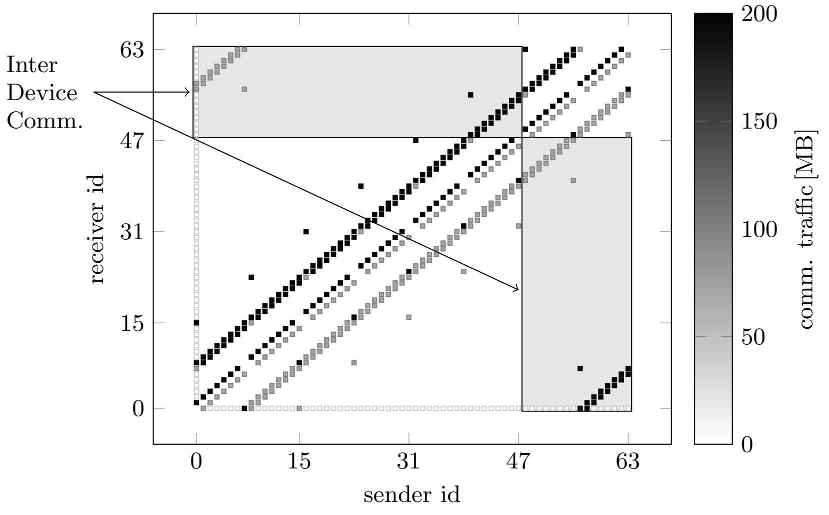


Figure 8: NPB BT (class C) communication traffic of 64 cores

The diagram visualizes a neighboring based communication pattern, since the majority of data points are located close to the diagonal. For sake of clarity, we selected a session size of 64 for the shown graphic, which highlights inter-device traffic by grey boxes.

The overall amount of inter-device communication clearly influences the application performance, since this communication path represents a bottleneck. For the given example, the maximum communication traffic between two ranks is about 186 MB. The communication pattern, which represents a ring, is challenging for a grid of SCCs as been presented in this work. Compared to the communication task that implements a simple routing of on-chip packets, a significant better speedup can be achieved.

5. CONCLUSION AND OUTLOOK

Expecting many-core processors with hundreds of processor cores in the future, on-chip communication latencies will automatically rise. Processor architectures that follow a design concept of many small cores with restricted functionality represent an option to fulfill a rising demand of energy efficiency. Classic concepts that include dedicated cores for the acceleration of communication were successfully applied to target similar challenges in the past [7]. The aspect of building many-core systems as a coprocessor makes those concepts even conspicuous for current computing systems.

In this paper, we have followed a reverse acceleration approach to scale a prototype of a cluster-on-a-chip architecture to 240 cores. We named our new research framework vSCC due to its composition of multiple SCC devices, which are tightly coupled through a virtual on-chip interconnect. Because the virtual extension works by tunneling the proprietary on-chip network protocol through a single host via PCIe, this setup raises latencies by a factor of 120. Our concept to handle those challenges is a separation between throughput and latency aware operations. In order to create a scalable grid of cluster-on-a-chip processors, we have implemented a reverse-acceleration model and applied classic software techniques to hide latencies for throughput oriented operations.

We have investigated in this work new communication schemes for a cluster-on-a-chip processor with benefits of offload communication and relaxed consistency. As a result of our optimizations, we were able to recover 24 % of effective on-chip communication throughput for the virtual extension, which is an impressive result for the investigated high-latency communication path. To achieve these results we propose extensions to the experimental SCC many-core architecture.

Parallel applications which extensively use blocking point-to-point communication with a neighborhood communication pattern show an excellent scalability. Our setup creates a prototype of a highly scalable many-core architecture with useful workarounds in software for hardware limitations. Consequently, our work may be used as a reference guide for similar application environments.

For future work, we plan to extend our communication concept to accelerate asynchronous communication.

6. ACKNOWLEDGMENTS

Pablo Reble has been supported by an Intel PhD fellowship. We thank Intel Labs for physical access to multiple SCC systems. We would also like to thank the anonymous reviewers for their constructive feedback that helped to improve this paper.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. *Intl. journal of supercomputer applications*, 5(3):66–73, 1991.

[2] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: a 64-core SoC with mesh interconnect. In *Solid-state circuits conference, 2008. ISSCC 2008. digest of tech-*

nical papers. IEEE international, Feb. 2008, pp. 588–598. DOI: 10.1109/ISSCC.2008.4523070.

[3] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *Proceedings of the 2011 international conference on high performance computing and simulation.* In HPCS. Istanbul, Turkey, July 2011, pp. 525–532. DOI: 10.1109/HPCSim.2011.5999870.

[4] C. Clauss, S. Lankes, P. Reble, J. Galowicz, S. Pickartz, and T. Bemmerl. iRCCE: a non-blocking communication extension to the RCCE communication library for the intel single-chip cloud computer. Users Guide and API Manual (Version 2.0). Chair for Operating Systems, RWTH Aachen University, Mar. 2013. DOI: 10.13140/2.1.2360.0961.

[5] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh. SCORPIO: a 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering. In *Proceedings of the 41st international symposium on computer architecture, ISCA 2014.* Minneapolis, MN, USA, July 2014. DOI: 978-1-4799-4394-4/14.

[6] M. Gries, U. Hoffmann, M. Konow, and M. Riepen. SCC: a flexible architecture for many-core platform research. *Computing in science & engineering*, 13(6):79–83, 2011.

[7] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel computing*, 20(3):389–398, Mar. 1994. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(06)80021-9.

[8] B. Joó, D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. Lee, P. Dubey, and I. Watson W. Lattice QCD on Intel Xeon Phi Coprocessors. In, *Supercomputing.* Vol. 7905, in Lecture Notes in Computer Science, pp. 40–54. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-38749-4. DOI: 10.1007/978-3-642-38750-0_4.

[9] T. Mattson and R. van der Wijngaart. *RCCE: a small library for many-core communication.* Version 2.0. Intel Corporation, Jan. 2011. URL: http://communities.intel.com/docs/DOC-5628.

[10] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE conference on supercomputing (SC10).* New Orleans, LA, USA, Nov. 2010.

[11] S. Pakin, M. Lang, and D. K. Kerbyson. The reverse-acceleration model for programming petascale hybrid systems. *IBM journal of research and development*, 53(5):721–735, Sept. 2009. ISSN: 0018-8646. DOI: 10.1147/JRD.2009.5429074.

[12] S. Potluri, A. Venkatesh, D. Bureddy, K. Kandalla, and D. Panda. Efficient intra-node communication on Intel-MIC clusters. In *Proceedings of 13th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid)*, May 2013, pp. 128–135. DOI: 10.1109/CCGrid.2013.86.

[13] P. Reble, C. Clauss, M. Riepen, S. Lankes, and T. Bemmerl. Connecting the cloud: transparent and flexible communication for a cluster of Intel SCCs. In *Proceedings of the many-core applications research community (MARC) symposium at the RWTH Aachen University.* Germany, Dec. 2012, pp. 13–19. ISBN: 978-300-039545-1.

[14] P. Reble and G. Wassen. Towards predictability of operating system supported communication for PCIe based clusters. In, *Euro-par 2013: parallel processing workshops.* Vol. 8374, in Lecture Notes in Computer Science, pp. 833–842. Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-54419-4. DOI: 10.1007/978-3-642-54420-0_81.

[15] *SCC external architecture specification (EAS).* Revision 1.1. Intel Corporation, Nov. 2010. URL: http://communities.intel.com/docs/DOC-5852.

[16] M. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. Jesshope. Efficient memory copy operations on the 48-core Intel SCC processor. In *Proceedings of the 3rd marc symposium.* KIT Scientific Publishing, Ettlingen, Germany, July 2011, pp. 13–18. ISBN: 978-3-86644-717-2.