MESSAGE FROM THE PROGRAM CO-CHAIRS

In continuation of a successful series of events, the 4th symposium of the *Many-core Applications Research Community (MARC)* took place at the *Hasso Plattner Institute for Software Systems Engineering (HPI)* in Potsdam. On December 8th and 9th 2011, researchers from different fields presented their current and future work on many-core hardware architectures, their programming models, and the resulting research questions for the upcoming generation of heterogeneous parallel systems.

While the Intel *Single Chip Cloud Computer (SCC)* serves as common research platform for most MARC members, other interesting research on next generation many-core platforms was also discussed on this event. The symposium focused on topics such as

- Operating system support for novel many-core architectures
- Virtualization solutions to deal with hardware limitations
- Dealing with legacy software on novel many-core architectures
- New approaches for leveraging on-die messaging facilities
- Traditional and new programming models for novel many-core hardware
- Concepts for runtime systems on novel many-core hardware
- Performance issues with modern on-die messaging facilities and caching infrastructures

This proceedings include 14 papers from 5 symposium sessions. Every paper was reviewed by at least three reviewers from the program committee, consisting of:

- Dr. Ulrich Bretthauer (Intel)
- Jaewoong Chung (Intel)
- Saurabh Dighe (Intel)
- Prof. Dr. Michael Gerndt (TU München)
- Diana Göhringer (Fraunhofer IOSB)
- Matthias Gries (Intel)
- Werner Haas (Intel)
- Prof. Dr. Hans-Ulrich Heiß (TU Berlin)
- Jim P. Held (Intel)
- Prof. Dr. Robert Hirschfeld HPI)
- Ulrich Hoffmann (Intel)
- Jason M. Howard (Intel)
- Dr. Michael Hübner (Karlsruhe Institute of Technology)
- Timothy M. Mattson (Intel)
- Georg Müller (Fujitsu)
- Prof. Dr. Jörg Nolte (BTU Cottbus)
- Prof. Dr. Andreas Polze (HPI)
- Dr. Felix Salfner (SAP Innovation Center)
- Prof. Dr. Bettina Schnor (Uni Potsdam)
- Prof. Dr. Theo Ungerer (Universität Augsburg)
- Dr. Peter Tröger (HPI)
- Dr. Daniel Versick (University of Rostock)
- Rob F. Van Der Wijngaart (Intel)

We would like to thank our program committee members for their hard work and for their suggestions in the selection of papers. We would like to thank all those who submitted papers for their efforts and for the quality of their submissions. We also would like to thank Jan-Arne Sobania and Sabine Wagner for their assistance and support.

Thank you for your active participation in the 4th MARC Symposium. We hope you found this event to be productive and enjoyable, and we look forward to seeing you next year at 5th MARC symposium and related events.

Peter Tröger & Andreas Polze, Hasso Plattner Institute, University of Potsdam, Germany

Potsdam, January 2012

CONTENTS

:

Improved RCKMPI's SCCMPB Channel: Scaling and Dynamic Processes Support

Isaías A. Comprés Ureña and Michael Gerndt
Technical University of Munich (TUM), Institute of Informatics,
Boltzmannstr. 3, 85748 Garching, Germany
{compresu,gerndt}@in.tum.de

***Abstract*—The Single-chip Cloud Computer (SCC), a 48 core experimental processor from Intel labs, is a platform for parallel programming research. Its hardware features and memory organization map naturally to message passing models. Standard and non-standard message passing libraries are already available for the SCC; one of the standard solutions is the RCKMPI library. RCKMPI's main features are three SCC specific MPICH2 channels. In this work, improvements to the SCCMPB channel are introduced; performance results for the new channel show better scaling with process count. The added flexibility of the new design also allows for the support of dynamic processes, a feature previously not supported in RCKMPI.**

***Index Terms*—MPI, dynamic processes, communication protocol**

I. Introduction and Related Work

The Single-chip Cloud Computer (SCC)[2] from Intel Labs is an attractive platform for parallel programming research. Having a distributed memory organization, the message passing model maps naturally to it. The Message Passing Interface (MPI) is a dominant standard for message passing; it is widely used in super computers and has been shown to scale to hundreds of thousands of cores. A large number of parallel applications that use MPI are available; these applications can be compiled and run in systems that have a compatible MPI library. Support for MPI on the SCC was possible since early in the chip's life, through the use of a network driver[11] and MPI libraries configured to use sockets. The downside of using the driver was that communication performance was much lower than the lightweight but non-standard solutions. In order to reach acceptable performance with message passing on the SCC, applications needed to be ported to its libraries, like RCCE[3] or its non-blocking improvement (iRCCE[4] from RTWH Aachen). In addition to RCCE, other projects have implemented their own message passing based communication protocols, like the TACO[9] and X10[10] ports to the SCC.

Compatibility with MPI with no significant compromise in performance is desirable in new parallel architectures, given the large amount of software and tools available for it. There are currently two MPI projects for the SCC: the RCKMPI[5] and the SCC-MPICH[8] libraries. With the introduction of RCKMPI, MPI applications reached performance that was comparable to that of the non-standard lightweight solutions on the SCC. RCKMPI's main contribution was the introduction of three SCC specific MPICH2[7] channels. Being a first attempt at efficient MPI on the SCC, it is natural to expect that there is potential for performance improvements in the channels; one such improvement was presented by Christgau et al. in [1], by the addition of topology-awareness to the library. In this paper, an improved communication protocol is presented for the SCCMPB channel of RCKMPI; the new design shows improvements in scaling with process count and supports dynamic processes from MPI-2.

II. Improved SCCMPB Channel

RCKMPI introduced three SCC channels: SCCMPB, SCCSHM and SCCMULTI. The SCCMPB channel uses only the Message Passing Buffer (MPB) for communication; in this work, this channel is improved.

At initialization for the SCCMPB protocol, each MPB of a participating process was partitioned in sections of equal size. The main disadvantage of this design is that the size of each EWS becomes smaller as the size of an MPI job increases; in the 48 process case, the EWS size is 160 bytes (with 12 bytes used for protocol metadata). The size of the EWS influences channel performance, since with smaller buffers the communication protocol requires more round trips to complete the transmission of a packet. The second disadvantage is that, because these are initialized at job startup and remain static until job termination, MPI-2 dynamic processes can not be supported. Finally, it was not possible to share the MPB with other subsystems or use it directly for optimized collectives. The new channel design addresses all of these shortcomings.

The new SCCMPB channel is partitioned differently and works with two different protocols. The first protocol is the original one found in RCKMPI and is labeled as the base protocol. The second protocol is labeled as the extended protocol, and it depends on the base protocol for coordination.

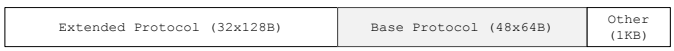


Fig. 1. MPB areas used by the base and extended protocols.

A. Base Protocol

Similarly to the channels in RCKMPI, the base protocol consists of statically allocated Exclusive Write Sections (EWSs) placed at the receiver and a polling based strategy for

new message detection. In contrast to the original, the size of these EWSs is not modified depending of the number of participating processes. They are always 48 and fixed at 64 bytes in size, for a total of 3KBs at each core's MPB. The 64 bytes in the static EWS setup allows for 48 bytes of payload. The remaining 16 bytes are used by the channel for metadata. The size of 64 bytes was selected based on the following observations:

- MPICH2 packet headers are 32 bytes.
- Latency sensitive operations (like barriers) benefit from dedicated buffers.
- Packets smaller than 48 bytes occur with high frequency at the channel.

The last was first observed empirically by using RCKMPI's channel statistics feature. Inspecting the MPICH2 device layer reveals that preamble steps involving barriers and other collective operations are common, when operating with larger buffers; these preamble operations result in small point to point traffic that is typically smaller than 16 bytes in payload. A packet with 16 bytes of payload result in 48 bytes total at the channel (32 header bytes plus 16 payload bytes).

The 16 bytes of metadata contain the following:

- **Checksum:** A checksum to improve consistency in case of a hardware error.
- **General purpose EWS control:** Used to control access to the general purpose EWS (gEWS) used by the extended protocol (described in detail in II-B).
- **Message size:** Bytes of payload currently available in the EWS. This size can exceed 48 bytes if part of the payload is located at the gEWS.
- **Packet size:** This is the total size of the MPICH2 packet in transit. This value is independent of the actual payload available in the EWS.
- **Receive sequence:** This value is used to indicate that a message was received at the remote core that owns the EWS.
- **Send sequence:** Sequence number of the message that is currently in the payload area of the EWS.

The progress engine polls this metadata when receiving messages. It determines if to use the base protocol together with the extended protocol, based on the gEWS control data.

B. Extended Protocol

The extended protocol uses a 4KB EWS that is labeled as the general purpose EWS (gEWS) internally. This buffer differs from the original EWSs in that it is placed at the sender and can be used to send messages to several receivers simultaneously. The gEWS can also be locked for its use in other operations, like spawn operations or optimized collectives. The size of 4KB was selected because it is the page size for the P54C architecture and it can be controlled with a single 32 bit field.

Together with the previously available metadata, a gEWS control field (32 bits) is specified by the sender. Each bit represents 128 bytes of the gEWS. All zeros indicate that the gEWS was not used for a particular message, while all ones indicates that the full 4KBs were used. When a new message is detected at the receiver, it reads the specified number of bytes starting from the payload area of the base protocol and then (if available) from the specified gEWS 128 byte slots.

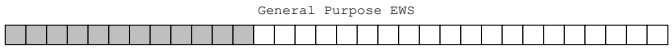


Fig. 2. Example gEWS state for 1536 bytes (bit field set to 0xFFF00000).

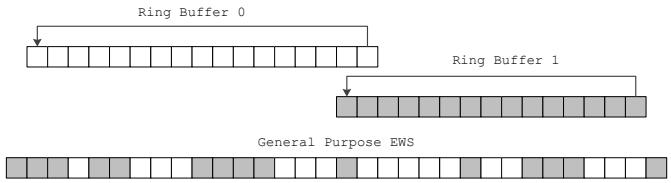


Fig. 3. Ring buffers on a fragmented general purpose EWS.

The extended protocol design can be used to serialize messages by writing payload to the gEWS in 128 byte chunks and then updating the relevant bits in the gEWS control entry. The gEWS can also be treated as one or multiple ring buffers; one ring buffer can be constructed per each remote core with the use of the bit field. In case of fragmentation, the bit field is used to specify which chunks are used to build a ring buffer (as shown in figure 3).

C. Protocol Characterization

To see why the addition of the extended protocol results in improved channel performance, an understanding of the original channel's behavior is necessary. When transmitting an MPICH2 packet, the total round trip time is the aggregation of the time required by several simpler operations. These times can be approximated with the following equation:

$$T_x(B,b,n) = [t_{sp}(n) + t_w + t_{rp}(n) + t_r + p_h t_h] \left\lceil \frac{B}{b} \right\rceil \quad (1)$$

where B is the size of the MPICH2 packet to send, b is the size of the EWS (in bytes) and n is the number of processes of the MPI job. The terms in the left factor represent the time required for writing, reading, polling and handling. The sender needs to poll the receive flag for the target process; this time is represented by t_{sp}. After the target EWS is available for writing, the bytes are written in t_w seconds. At the receiver, the progress engine polls the metadata to detect new messages; t_{rp} seconds are spent in doing this and then t_r seconds of CPU time are used reading the available payload. Polling times depend on the number of processes n. If the MPICH2 packet is complete with the last read payload, then t_h seconds are spent handling it; handling of a packet occurs with an application dependent probability of p_h.

These operations are done for each round trip of the communication protocol. The number of round trips required is the ceiling of the size of the packet divided by the size of

the EWS (the $\left\lceil \frac{B}{b} \right\rceil$ factor in formula 1). The time required to write at the sender and to read at the receiver are the same: $t_r = t_w = t_{rw}$. These are *memcpy* operations and their aggregated time t_{arw} depends on the total number of bytes to transfer, independently of the number of round trips. The time required for polling at the sender and receiver can be represented by a single variable for their combined worst case as t_{wcp}. Furthermore, packet handling is done with a much lower frequency; packets are only handled when they are done after several protocol round trips and can be ignored. With these observations, 1 can be simplified as:

$$T_x(B, b, n) = 2t_{arw}(B) + t_{wcp}(n) \left\lceil \frac{B}{b} \right\rceil \quad (2)$$

Conclusions can more easily be drawn from 2. The $2t_{arw}$ term is a function of the total bytes B of the packet. The polling overhead t_{wcp} depends on the process count and increases linearly with it, since metadata is polled in a round robin fashion. The number of round trips $\left\lceil \frac{B}{b} \right\rceil$ depends on the process count as well, since the size of b is determined at initialization based on the MPI job size.

The new design can be modeled similarly to the original one. The effect of the gEWS in the protocol, is that depending of the probability of it being free, the round trips required to transfer a packet are greatly reduced:

$$T_x(B, n) = 2t_{arw}(B) + t_{wcp}(n) \left[p \left\lceil \frac{B}{b_{4KB}} \right\rceil + [1-p] \left\lceil \frac{B}{b_{48B}} \right\rceil \right] \quad (3)$$

where p is the probability of the gEWS being free and is application dependent. The number of round trips now depends on the application alone (given that the EWS and gEWS are now fixed with size b_{4KB} and b_{48B}), and not on the number of processes. The worst case polling time still depends on process count, and is therefore not improved with respect to the original protocol.

From 3 it is easy to see that $p \approx 1$ is desirable. Because of the way MPICH2 collectives (with a logical ring topology and other schemes) and most MPI applications (that send messages to only a few processes at the same time) are implemented, this is very often the case. Communication between a pair or processes is never stopped if the gEWS is not available, performance is just degraded by the limitation of 48 bytes of payload per round trip of the static EWS.

III. Support for MPI-2 Dynamic Processes

The dynamic processes functionality of MPI-2 is not used with the same frequency as point to point, collectives or one sided communication on current MPI applications. For this reason, their exclusion was found to be acceptable in the first release of RCKMPI. The spawn, connect and disconnect operations are necessary to support dynamic processes. Connect and accept are implemented at the channel, while spawn involves the interaction of the process manager and several parts of the library.

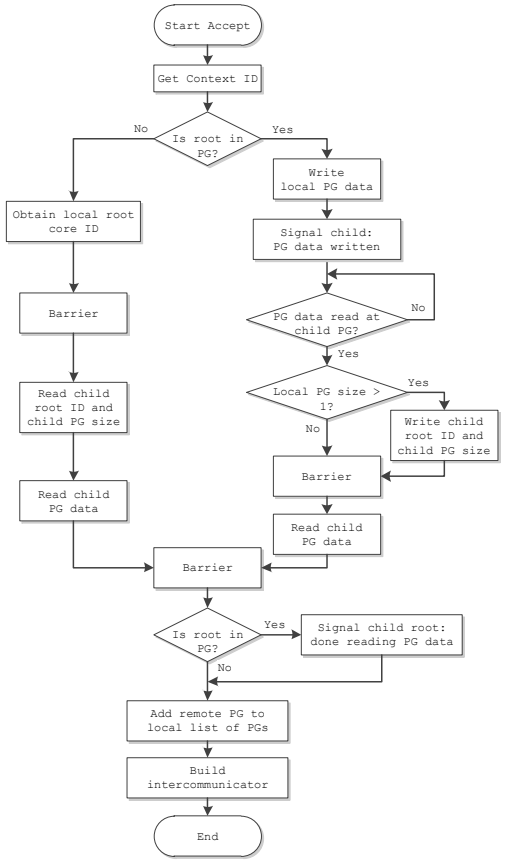


Fig. 4. Flow diagram of the accept algorithm.

For the addition of connect and accept to the channel, the gEWS was used. Since it is a shared resource, in this case used by the point-to-point communication subsystem, there are necessary steps before it can be used. The owner of the gEWS is the local core; therefore, its global state is stored in a local 32 bit field. If the gEWS is in use, the channel calls the progress routines until the send queues are cleared; after that, the gEWS is free to be used for any other purpose.

The spawn operation involves two process groups: the parent group and the child group. At each of these groups, one of the processes is the root process. The algorithm used is similar to the default one found in MPICH2, but latency optimized by the use of the gEWS directly, instead of relying on the non blocking point to point functionality provided by the channel implementation.

The parent group does an accept operation while the child group does a connect operation (flow charts shown in figures 4 and 5). The root process of each group writes the core IDs and other data required to build a process group structure, on its own gEWS. Then, they contact each other, exchange minimal but essential data: core ID where each is running, process group ID and size of each remote process group. After this, each root process shares this information with its peers, which read the rest of the process group data directly from the gEWS

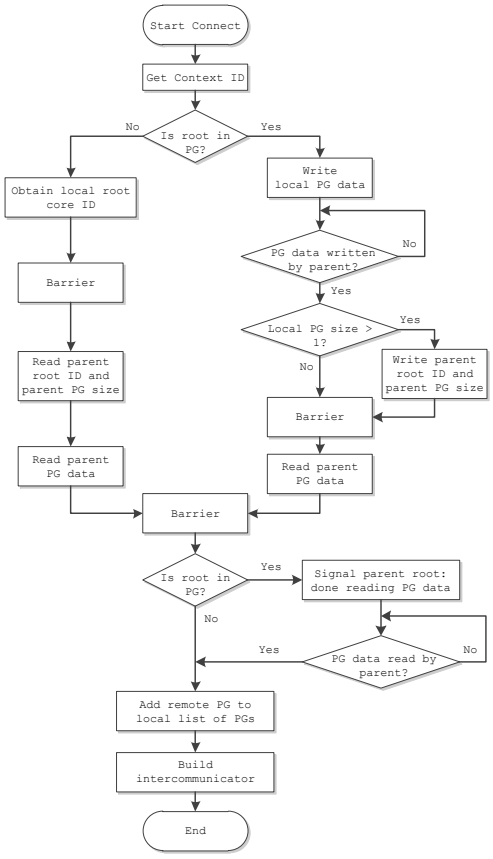


Fig. 5. Flow diagram of the connect algorithm.

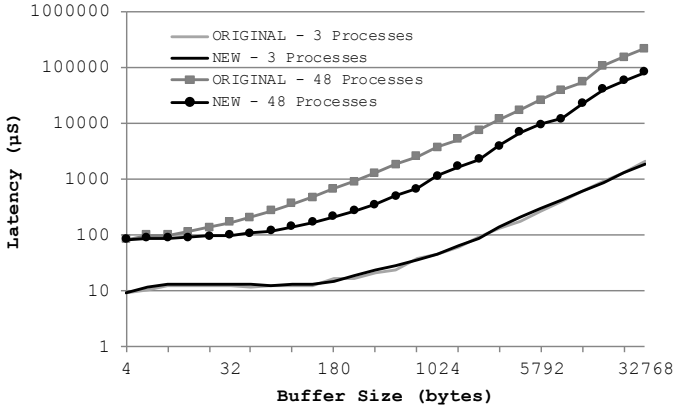


Fig. 7. *MPI_Gather* scaling with buffer size.

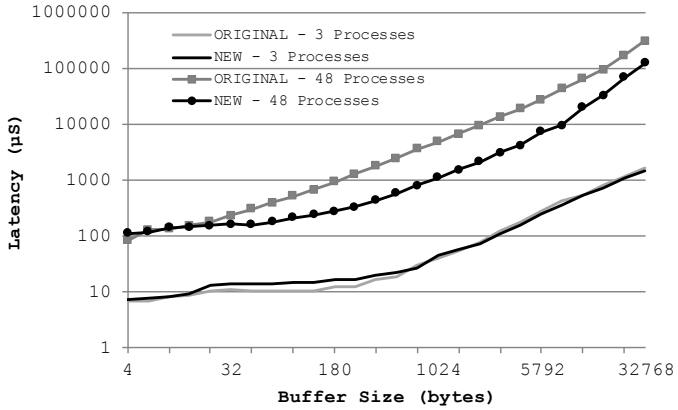


Fig. 8. *MPI_Scatter* scaling with buffer size.

of the remote root process.

Before this operation is possible, the process manager passes the business card of the root parent to the root process at the newly spawned child process group. The root process of the parent group then waits for the child root to initialize the communication, and this is where the connect and accept operations start. The new process group generated by these pair of operations, is added to the process group list of each process. These groups are disjoint and can be reached through an inter-communicator, as specified in the standard.

IV. Performance Evaluation

In this section, the original and new SCCMPB channels' performance is evaluated with the use of the SKaMPI 5.0 benchmark suite and the NAS 3.3 LU and BT benchmarks. The software, hardware and configuration are the same for all tests in this section. The Rocky Lake SCC systems used for testing were configured with maximum frequency settings: 800MHz for the cores, 1600Mhz for the tiles and routers and 1066Mhz for the DDR3 memory. A Linux 2.6.38 image was loaded in all cores. The GCC compilers version 4.5 we used to compile the kernel, libraries and applications. Both C and Fortran MPI applications were compiled with the -O3 flag.

A. SKaMPI 5

SKaMPI[12] is a benchmark suite that covers most of the MPI-2 API. Results for point to point and collective communication are presented here. Figure 6 shows point to point latency scaling with *MPI_Sendrecv* (round trip times) for different message sizes. When running with 48 processes, the new channel scales better than the original for buffers greater than 128 bytes; at 16KB messages, their point to point performance differs by a factor of 6.25.

MPI_Gather scaling for different buffer sizes is presented in figure 7. In the 48 process case and for 16KB buffers, their latency differs by a factor of 2.6. Scaling results for *MPI_Scatter* are shown in figure 8. For the 48 process case, the new channel outperforms the original by a factor of up to 4.6.

MPI_Bcast scaling results, for 1KB and 256KB buffers, are presented in figure 9. In the 1KB buffer case, scaling is similar in both channels (as shown in 9(a)); however, absolute performance is much better in the new channel. For 1KB buffers, the latency differs by a factor of 3.5 in the 48 process case. For 256KB buffers, the difference in latency does not change much with process count (as presented in 9(b)). The latency of the old and new channels differ by a factor of 3.2, in this case.

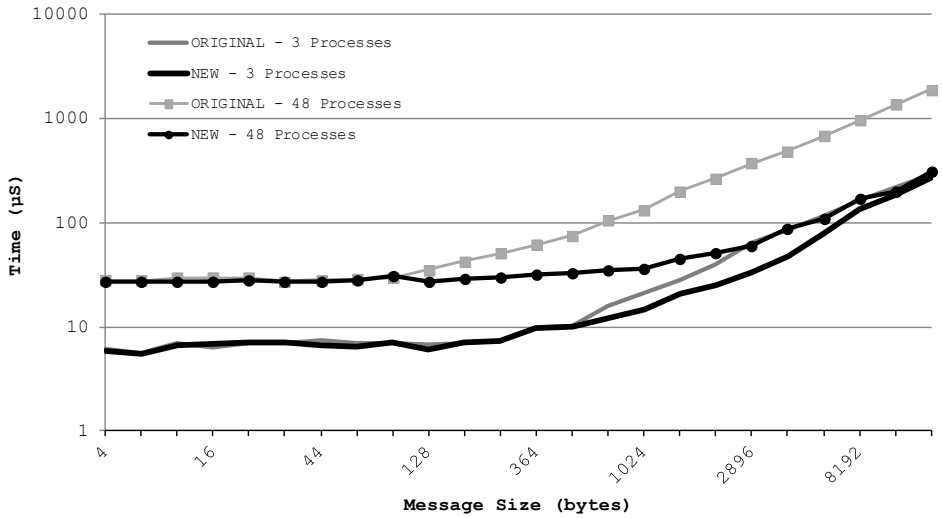
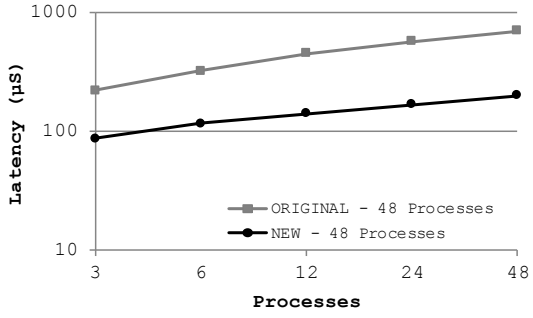
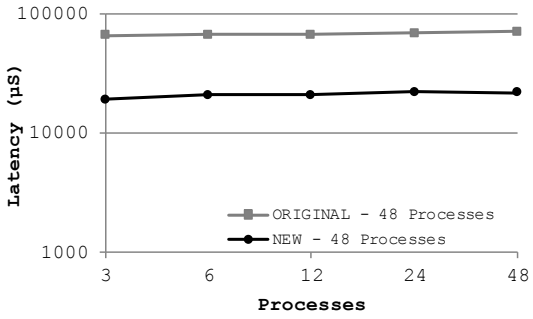


Fig. 6. *MPI_Sendrecv* scaling with message size.



(a) 1KB buffers



(b) 256KB buffers

Fig. 9. *MPI_Bcast* scaling with process count for 1K and 256K buffers.

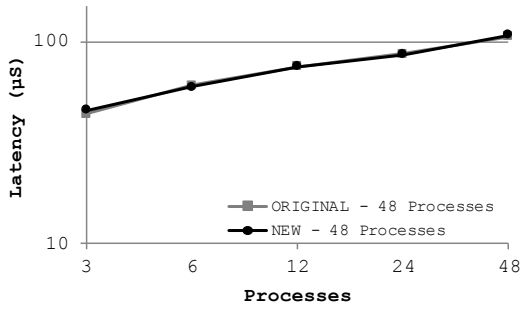


Fig. 10. *MPI_Barrier* scaling.

Scaling results for *MPI_Barrier* are presented in figure 10. The latencies for this operation are very similar for both channels. This is expected since for small payloads (16 bytes and below) the same communication protocol is used by both channels.

B. NAS Benchmarks

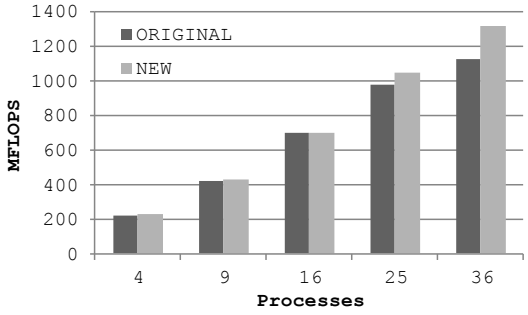
The NAS parallel benchmarks[13] are useful for evaluating parallel computers. The algorithms used by it are found very often in scientific applications. Results for the BT and LU benchmarks, at sizes W and A, are presented in this section.

Both channels perform nearly the same when running the BT benchmarks (shown in figure 11) with 4 to 16 processes. For the BT benchmark, the new channel shows a performance improvement over the original one when running with 25 and 36 processes (as shown in 11(a) and 11(b)). The improvement is higher for the W size of the benchmark.

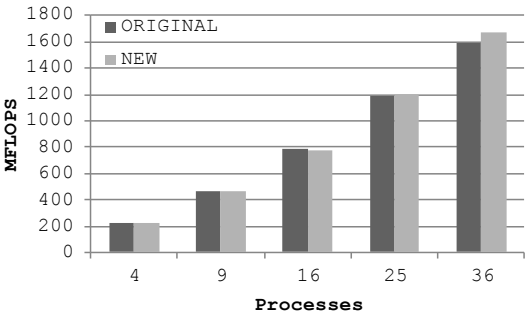
Results from the LU benchmark (figure 12) are very similar to those in the BT one. Performance when running with 4 to 16 processes is nearly the same with both channels. When running with 32 processes, the new channel shows better results (as presented in 12(a) and 12(b)).

V. Conclusion and Future Work

An improved design for the SCCMPB channel of RCKMPI was presented. The design consists of a base and an extended protocol that compliment each other. In contrast to the original design, the use of these new protocols resulted in channel bandwidth that is less dependent of process count; this was a consequence of the use of large EWS placed at the sender, that is shared for communication with several processes si-

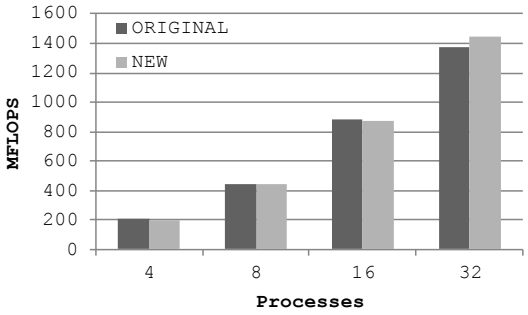


(a) Size W

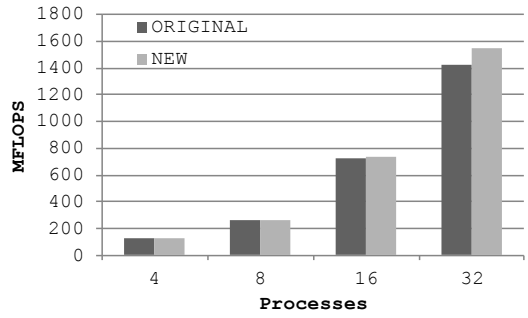


(b) Size A

Fig. 11. NAS BT scaling with process count.



(a) Size W



(b) Size A

Fig. 12. NAS LU scaling with process count.

multaneously and reduces the number of round trips required to complete a transfer.

Performance results from the SKaMPI and NAS parallel benchmarks were presented. The new design of the SCCMPB channel clearly outperformed the original one when running MPI jobs that use the 48 cores of the SCC; the advantage could be better observed in the SKaMPI point-to-point and collective tests. Good results were also observed for the NPB benchmarks for process counts larger than 25; the improvements in these benchmarks are not as large, since they have high computation areas and the improved channel performance only affects MPI communication times.

The way the MPB is partitioned was also modified. The new scheme allowed the MPB to be used by other subsystems of the MPI library and for optimized operations. This flexibility was used to add support for MPI-2 dynamic processes, by the addition of accept and connect operations that use the MPB directly. Future implementations of optimized collectives and one sided operations were also made possible by this new approach; these are good targets for future performance improvements to the RCKMPI library.

REFERENCES

[1] Steffen Christgau, Simon Kiertscher, and Bettina Schnor. The benefit of topology awareness of MPI applications on the SCC. In Diana Göhringer, Michael Hübner, and Jürgen Becker, editors, *MARC Symposium*, pages 47–51. KIT Scientific Publishing, Karlsruhe, 2011.

[2] Jim Held. "Single-chip Cloud Computer" an IA tera-scale research processor. *Euro-Par Workshops*, volume 6586 of *Lecture Notes in Computer Science*, page 85. Springer, 2010.

[3] Timothy G. Mattson, Rob F. Van der Wijngaart, Michael Riepen, et al. The 48-core SCC processor: The programmer's view. Supercomputing Conference. ACM/IEEE, New Orleans, LA, USA, November 2010.

[4] Carsten Clauss, Stefan Lankes, Jacek Galowicz, Thomas Bemmerl, iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer December 17, 2010, Chair for Operating Systems, RWTH Aachen University

[5] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. RCKMPI - lightweight MPI implementation for intel's single-chip cloud computer (SCC). *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2011.

[6] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Lightweight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45:73–83, February 2011.

[7] William Gropp. MPICH2: A new start for MPI implementations. *Lecture Notes in Computer Science*, 2474:7, 2002.

[8] Carsten Clauss, Stefan Lankes, and Thomas Bemmerl. Performance tuning of SCC-MPICH by means of the proposed MPI-3.0 tool interface. *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 318–320. Springer, 2011.

[9] Randolf Rotta. On efficient message passing on the intel SCC. In Diana Göhringer, Michael Hübner, and Jürgen Becker, editors, *MARC Symposium*, pages 53–58. KIT Scientific Publishing, Karlsruhe, 2011.

[10] Keith Chapman, Ahmed Hussein, and Antony Hosking. X10 on the scc. Santa Clara, United States, March 2011. Presented at the Second MARC Symposium.

[11] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Lightweight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45:73–83, February 2011.

[12] R. Reussner, P. Sanders, L. Prechelt, and M. Mueller. SKaMPI: A detailed, accurate MPI benchmark. *Lecture Notes in Computer Science*, 1497:52, 1998.

[13] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-91-002, NAS Systems Division, January 1991.

:

The Path to MetalSVM: Shared Virtual Memory for the SCC

Stefan Lankes*, Pablo Reble*, Carsten Clauss* and Oliver Sinnen†
*Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
Email: {lankes,reble,clauss}@lfbs.rwth-aachen.de
†Department of Electrical and Computer Engineering, University of Auckland
Private Bag 92019, Auckland 1142, New Zealand
Email: o.sinnen@auckland.ac.nz

***Abstract*—In this paper, we present first successes with building an SCC-related shared virtual memory management system, called MetalSVM, that is implemented using a bare-metal hypervisor, located within a virtualization layer between the SCC's hardware and the operating system. The basic concept is based on a small kernel developed from scratch by the authors: A separate kernel instance runs on each core and together they build the virtualization layer. High performance is reached by the realization of a scalable inter-kernel communication layer for MetalSVM. In this paper we present the employed concepts and technologies. We briefly describe the current state of the developed components and their interactions leading to the realization of a Shared Virtual Memory system on top of our kernels. First performance results of the SVM system are presented in this work.**

***Index Terms*—Many-Core, SCC, SVM, Non-Cache-Coherent Shared-Memory**

I. Introduction

Since the beginning of the multicore era, parallel processing has become prevalent across-the-board. A further growth of the number of cores per system implies an increasing chip complexity on a traditional multicore system, especially with respect to hardware-implemented cache coherence protocols. Therefore, a very attractive alternative for future many-core systems is to waive the hardware-based cache coherency and to introduce a software-oriented approach instead: a so-called Cluster-on-Chip architecture.

The Single-chip Cloud Computer (SCC) experimental processor [1] is a *concept vehicle* created by Intel Labs as a platform for many-core software research, which consists of 48 P54C cores. This architecture is a very recent example for such a Cluster-on-Chip architecture. The SCC can be configured to run one operating system instance per core by partitioning the shared main memory in a strict manner. However, it is possible to access the shared main memory in an unsplit and concurrent manner, provided that the cache coherency is then ensured by software.

A common way to use such an architecture is the utilization of the message-passing programming model. However, many applications show a strong benefit when using the shared memory programming model. The project *MetalSVM* aims the realization of a SCC-related shared virtual memory management system that is implemented in terms of a bare-metal hypervisor and located within a virtualization layer between the SCC's hardware and the current operating system. This new hypervisor will undertake the crucial task of coherency management by the utilization of special SCC-related features such as its on-die Message-Passing Buffers (MPB). In order to offer a maximum of flexibility with respect to resource allocation and to an efficiency-adjusted degree of parallelism a dynamic partitioning of the SCC's computing resources into several coherency domains will be enabled.

This paper focuses on the design of the MetalSVM kernel and its drivers optimized for the SCC as well as the SVM system. In Section II we refer to our previous work on the SCC and summarize related work regarding SVM system. We present a detailed insight in Section III to the design of *MetalSVM* and our small self-developed operating system kernel that builds the base of *MetalSVM*. The realization of an SVM system prototype is presented in Section VI. Important facts on the SCC supporting the path to *MetalSVM* are mentioned in Section IV and V with a focus on the memory system of the SCC followed by the implementation of a communication layer for *MetalSVM*. Section VII contains the knowledge on the port of a virtual IP interface to the SCC and presents related benchmark results. In Section VIII we describe first results for an exemplary parallel program using the SVM system prototype.

II. Previous Work

Referring to our previous work on the SCC we present further development on the fast inter-kernel communication layer as well as a closer look at the SVM system in this paper. The motivation and concept of our MetalSVM has been introduced at the 3rd MARC Symposium [2]. In addition to a summary of previous work on cluster-based SVM systems we first outline the potential of our approach. Other contributions to this Symposium have also shown that the memory system of the SCC is special and established methods hold a high potential for optimization. [3]

In [4], we evaluated different programming models (especially shared-memory and message-passing) for the SCC and we have shown how these models can be improved with

respect to the SCC's many-core architecture. Our experiments have shown that in particular the shared-memory programming is very complex and involved if caches are enabled because of the missing hardware cache coherency.

The Chair for Operating Systems (LfBS) at the RWTH Aachen University developed since 1996 the *Shared Memory Interface* (SMI) [5] as a programming interface that provides a large function set such as allocation and management of cluster-wide shared memory regions and its distribution and synchronization services. SMI provides no virtual common address space in contrast to an SVM system. However, shared memory regions can be explicitly allocated and managed. A small subset of its capabilities is used in this paper to benchmark our prototype of *MetalSVM*.

Existing SVM solutions are mainly based on traditional message-passing oriented networks. However, the SCC has the capability to directly access memory. From a programmer's perspective this is comparable to the Scalable Coherent Interface (SCI) standard [6] that belongs to the memory-mapped networks. In addition to the offer of a transparent read/write access to remote memory, SCI also defines a cache coherency protocol. But, PCI-SCI adapter cards that are available on the market do not support this feature. Several research projects used SCI-based PC clusters, which possessed a similar characteristic like SCC. Both systems consist of several processing units which are able to communicate transparently over shared memory regions without the support of cache-coherency.

At the LfBS, we have developed an SVM system for Intel architecture based compute clusters, called *SVMlib* [7], [8], which stores write notices and related changes in the global memory to realize a *Lazy Release Consistency* [9] model. Experiments have shown that the implementation of *SVMlib* at user level decreases the usability.

Furthermore, SVM systems can be integrated into virtual machines providing a simpler and more transparent access to the shared memory for an easy application of common operating systems and development environments. The vSMP architecture by ScaleMP[1] enables a cluster-wide cache-coherent memory sharing by implementing a virtualization layer underneath the OS that handles distributed memory accesses via InfiniBand-based communication on x86-based compute clusters. A similar project is vNUMA [10], which used Ethernet as interconnect. This project shares characteristics with our hypervisor approach such that the implementation of the SVM system takes an additional virtualization layer between the hardware and the operating system.

In fact, we want to exploit the SVM system with SCC's distinguishing capabilities of transparent read/write access to the global off-die shared memory.

III. Design of MetalSVM

The concept of *MetalSVM* is to run a common Linux version without SVM-related patches on the SCC in a multicore manner. For a better understanding, the structured diagram of Figure 1 illustrates the design approach of *MetalSVM*.

A major advantage of our approach, as introduced in [2], is no binding of *MetalSVM* to a certain version of Linux, because *integrating* would for example mean *patching* the kernel. The light weight hypervisor is based upon the idea of a small virtualization layer based on a monolithic-kernel developed from the scratch by the authors. A well-established interface to run Linux as para-virtualizated guest which is part of the standard Linux kernel is used to realize our hypervisor. Consequently, no modifications to the Linux kernel are needed.

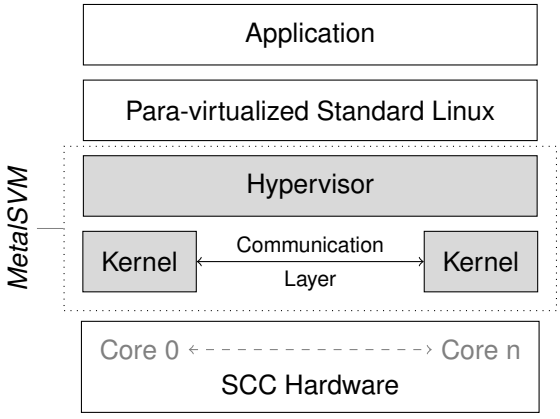


Fig. 1: Concept and Design of MetalSVM

The aim of common processor virtualization is to provide multiple virtual machines for separated OS instances. We want to use processor virtualization that provides *one* logical but parallel and cache coherent virtual machine for a single OS instance, for instance Linux, on the SCC. Hence, the main goal of this project is to develop a bare-metal hypervisor, that implements the required SVM system (and thus the memory coherency by applying appropriate consistency models) within this hardware virtualization layer in such a way that an operating system can run almost transparently across the entire SCC system.

IV. Memory System

In this section we first briefly recap the memory system of the SCC and second outline the effects on the realization of an SVM system.

The SCC possesses four memory controllers providing a maximum capacity of 64 GByte of DDR3 memory. Each core has logically assigned 8 kByte of a tile's local memory buffer, called message passing buffer (MPB). To close the gap between register and main memory access time, the SCC cores have a classical memory hierarchy consisting of a local Level 1 and Level 2 cache. In addition to a Level 1 data and instruction cache size of each 8 kByte, all cores have a local Level 2 cache size of 256 kByte. Caches are organized with a cache-line size of 32 Byte in a non cache-coherent manner.

Intel Labs extended the P54C instruction set architecture (ISA) by a new instruction `CL1INVMB` that is closely connected to a new memory type (`MPBT`) indicated by a flag on

[1] http://www.scalemp.com

page granularity to support the use of the MPB. Accesses to this new memory type bypass the Level 2 cache and by default message-passing buffer entries are tagged.

Moreover, the flag that indicates `MPBT` can be used in a more generic way. Generally speaking, information about a special data type is tagged in hardware. However, this mapping is not fixed and can be adapted to use the hardware support that facilitates a coherent view on the MPB also for an SVM system.

Another extension of the SCC cores to the P54C architecture is a write combine buffer that holds one cache-line of 32 Byte. In write through mode accesses touching the same cache-line are wrapped together and written back en block from the Level 1 cache to the next level in memory hierarchy. This behavior may turn out to be useful for the SVM system. The intention for adding this feature was to accelerate the message transfer between the cores [1].

The P54C architecture uses an external Level 2 cache without the possibility to flush contents using hardware support. A flush routine has been developed that replaces all L2 contents by reading invalid data but this turned out to be costly. [11] We limit our first experiments to an SVM system prototype that only enables L1 caching for a shared memory region. To control write strategy of cached data a page table entry contains a bit, that the memory management of *MetalSVM* sets for shared pages to uses a *write through* strategy.

Obviously, a drawback of this solution is a significantly smaller amount of cache in use for shared regions. But to waive the use of Level 2 cache for shared memory regions a major advantage arises that is the possibility to tag SVM related data. Thus, a selective invalidate of cached data via `CL1INVMB` is possible. Due to the fact that our current SVM system uses *write through*, a method called *fool write combine buffer* is sufficient to flush cached data. The method simply touches an MPBT tagged cache-line that is only used for this purpose. Thus, the off-die memory holds current data.

V. Communication Layer

The realization of the hypervisor needs a fast inter-core communication layer, which will be used to manage resources between the kernels. An important requirement to this communication layer is the support of asynchronous message-passing because it is not predictable when a kernel needs an exclusive access to a resource that is owned or managed by another kernel instance. As a result, the synchronous communication library *RCCE* [12] is not suitable for *MetalSVM*. An alternative approach is to copy the message to the message-passing buffer of the receiving core and afterwards to signalize the incoming message with a remote interrupt.

Interrupt Handling

Realization of event based communication between the SCC-cores needs either interrupts or events have to be checked at defined points in time. We followed an interrupt driven approach for our communication layer to enable a fast communication. On the one hand the latency of signal passing is important. On the other hand the time to process signals and its scalability influences the performance of our communication layer.

Previous versions of *sccKit* only supported the generation of an Inter-Processor Interrupt (IPI) by writing directly to the receiving core's configuration register. Hence, the receiving core can be interrupted this way but no information can be obtained about the sender of a specific interrupt. Since *sccKit 1.4.0* the system FPGA holds a Global Interrupt Controller (GIC) [13]. In addition to the direct method to generate an IPI the possibility arises to indirectly generate an IPI using the GIC. Consequently, this IPI can be used to obtain the information by which core it has been raised.

Event processing of the mailbox system, described in the following, is realized in the interrupt handler of *MetalSVM*. With the focus on scalability the information on the sender of an interrupt creates the option for a mailbox system to selectively check mailboxes.

Mailbox System

A mailbox system has become part of *MetalSVM's* communication layer and extends iRCCE [14] to enable an event driven and fast asynchronous communication path between the SCC cores. For each communication path between two cores a mailbox of one cache-line size is reserved at each local MPB. Thus, the mailbox system takes 1.5 kByte of MPB space per core assuming a maximum number of 48 cores. RCCE provides a memory allocation scheme to manage the remaining MPB space of 6.5 kByte.

Accesses to a specific mailbox of a target core are restricted by only allowing the receiver to read data and toggle a send flag that the mailbox contains. A sender with the intention to pass a signal is allowed, in addition to toggle the send flag, to write data to the mailbox. Whenever a receiver toggles the send flag a signal has been processed and when a sender toggles the send flag a new signal has been placed. As a result of this communication method the generation of a *single reader single writer* problem leads to a simplified synchronization scheme that is enabled by the restriction of accesses to the mailboxes.

Signals between the cores are passed in a *remote write and local read* approach in contrast to the *local write and remote read* approach of the RCCE library. The mailbox system reverses the data flow compared to the RCCE send respective receive methods because event processing is realized in the interrupt handler.

VI. SVM System

The SVM system manages pages located in shared memory. A coherent view on the virtual common address space is enabled by flushing cached data at defined points in time. For a first prototype three functions are sufficient to enable the use of the SVM system and thereby explore the capabilities of the SCC for a software managed coherence scheme. Following SMI like functions are provided under *MetalSVM* to a kernel task of the current SVM version:

- `svm_alloc`
- `svm_flush`
- `svm_invalidate`

The function `svm_alloc` is used to allocate an amount of bytes in a cached shared memory region. The function `svm_flush` is used to implicitly write back modified data[2], and `svm_invalidate` to remove possibly outdated data from the cache. This is either done within the interrupt handler of the current page owner or within the page fault handler on the core where the access violation occurs.

The SVM system of *MetalSVM* uses the mailbox system for the crucial task to change access permissions of shared pages. Therefore, a signal is sent to the page owner which can be identified because the information of ownership is located in a shared memory region and therefore accessible by all cores. If the ownership has changed in the meantime, e. g. another core has requested the page, the receiver of the signal has to forward the message to its new destination. As a result, the first sender of a signal in addition to the address of the target shared page is necessarily encoded by a signal, so that the owner vector entry can be updated.

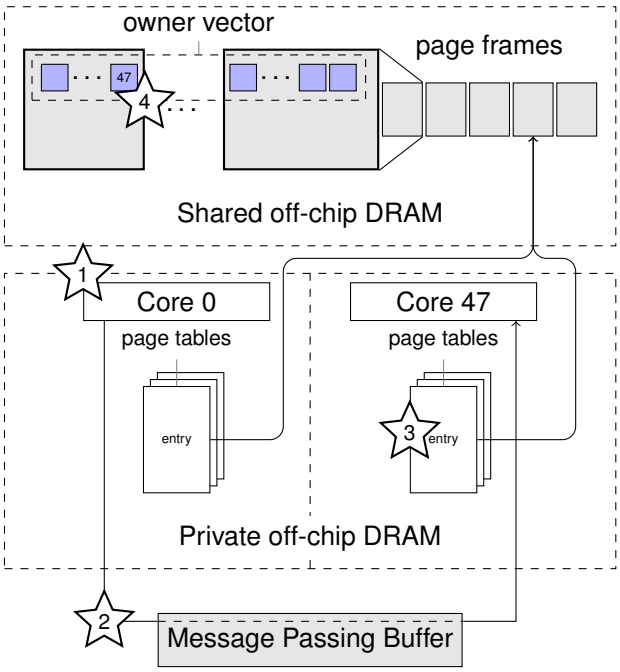


Fig. 2: Concept and design of the SVM subsystem

A strong consistency model is supported by the prototype implementation of our SVM system. At each point in time only one owner of a page exists which is allowed to read or write to it. This ownership is registered in an ownership vector, which is also located in the off-die memory as exemplarily illustrated by Figure 2. Each core possesses its private page tables.

Whenever a page is accessed without permission a kernel enters the page fault handler and sends a request to the current owner via the mailbox system. Regarding the strong consistency model no parallel access to shared pages is allowed and the ownership has to be exchanged. First, the current owner of the page clears its access permission. Second, it flushes the cache and third sets the new owner id to the ownership vector as an acknowledgment. As a result the core that requested access is registered as the new owner. After this procedure the requesting core can continue its calculation. Obviously, the performance of the mailbox system has a direct impact to the performance of the SVM system.

Figure 2 shows an example where an SVM related page fault occurs at Core 0 involving Core 47. Following steps have to be performed:

1) A page fault occurs at Core 0
2) After sending a message to Core 47 requesting the page, Core 0 is polling on the owner vector entry
3) Core 47 flushes its cache and changes the page table entry
4) Core 47 changes the ownership

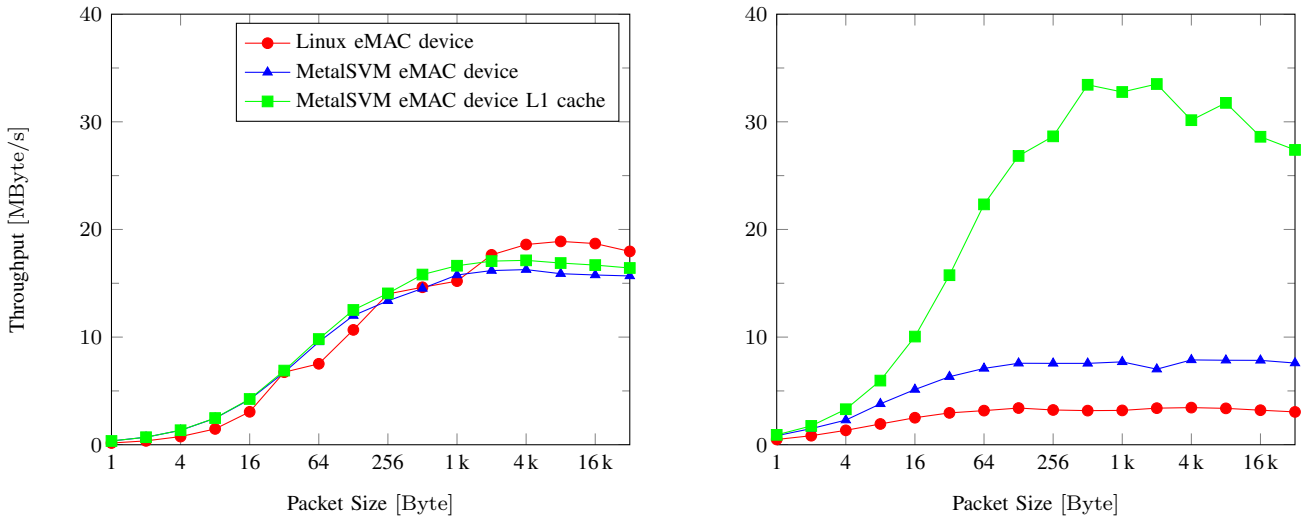
After this procedure Core 0 is the new owner and hereby has full access permissions.

VII. IP Stack

In this section we present the realization of two IP devices, one memory mapped virtual device for the realization of on-die communication and one eMAC device for the off-die communication. For this purpose the light-weight IP (lwIP) stack [15] has been integrated into the *MetalSVM* kernel. As a result, established BSD sockets are supported to enable an easy integration of standard application. In addition, we analyze a variant that interacts with the IP driver using an overloaded socket that bypasses the full IP stack. For further performance optimizations the developed devices are fully configurable having options to choose the MPB or off-chip DRAM for communication and to enable L1 caching. Applications for the described devices can be a monitoring the SVM system or providing an IP service to the guest operating system. Here, the guest can use a tunnel device to hand down IP packets to *MetalSVM*.

In principle, the first driver is a porting of Linux's eMAC device driver to lwIP and builds an interface to the Ethernet ports that are connected to the SCC. We used the driver of SCC Linux from sccKit 1.4.1 within the scope of Linux kernel `2.6.38.3-jbrummer` as a reference, which uses non-cachable memory for the communication between kernel and hardware device. Again, the SCC offers the possibility to invalidate in one cycle the cache entries for `MPBT` tagged pages. The option to enable the L1 cache for the receive buffers of the eMAC device generates the possibility to visualize the benefit of this hardware support for communication. Here, specific cache entries have to be invalidated before the receiver reads data from its receiver buffer. When compared to the Linux driver that holds the L1 cache disabled for the receive buffers, a positive impact on performance is expected for the *MetalSVM* driver that reads a whole cache-line from the memory.

[2]In this scenario, flushing of the write combining buffers.



(a) SCC→MCPC

(b) MCPC→SCC

Fig. 3: Transfer Throughput between MCPC and SCC via eMAC

Obviously this method reduces the number of memory accesses up to a factor of:

$$\frac{8 \cdot t_{CM}}{t_{CM} + 7 \cdot t_{CH}}$$

Where, t_{CM} is the time for a cache miss and t_{CH} is the time for a L1 cache hit.

The second driver uses an established standard and enables a virtual IP interface to realize inter-kernel communication. The support of standard interfaces for communication is not in the focus of MetalSVM. However, a driver has been written that realizes communication via memory mapped regions. For this driver a configuration exists to use either the off-die or the on-die memory (MPB).

The first configuration uses the off-die shared memory for communication and therefore generates no load to the MPB. An application might be to monitor the SVM system. The use of the second configuration is preferred to reach a higher performance. However, using the MPB can generate noise to the SVM system that runs in parallel.

In principle, each receiver optionally creates its own receive buffer either in on-die or off-die memory. The senders copy their data directly into the receive buffer and wake up the receiver via an inter processor interrupt. To allow parallel access between the receiver and senders, the receive buffer is managed as heap. The maximum transfer size is:

$$\frac{1}{2} \cdot sizeof(buffer) - sizeof(cacheline)$$

The result of the split of larger messages into smaller sub-messages is that the receiver is able to process sub-messages that are present during the next transfer operation of the sender.

The data structure to manage the heap is located at the off-die memory to increase the size of the receiver buffer. In contrast to the presented mailbox system the lwIP drivers use only one receive buffer per core. This is because the incoming messages are clearly larger than a mail of the mailbox system. Accesses to the receive buffers have to be synchronized. Therefore, the current version uses RCCE locks which enable an access to the hardware implemented Test-And-Set registers. Many features of the IP stack are needless for the inter-core communication. For instance, on the SCC it is not possible to receive corrupt data. To benefit from this behavior, we have developed a prototype, which emulates the BSD socket interface, bypasses the IP stack and forwards the messages directly to the receivers. In our approach, a parallel using of the IP stack and the bypassing approach is possible.

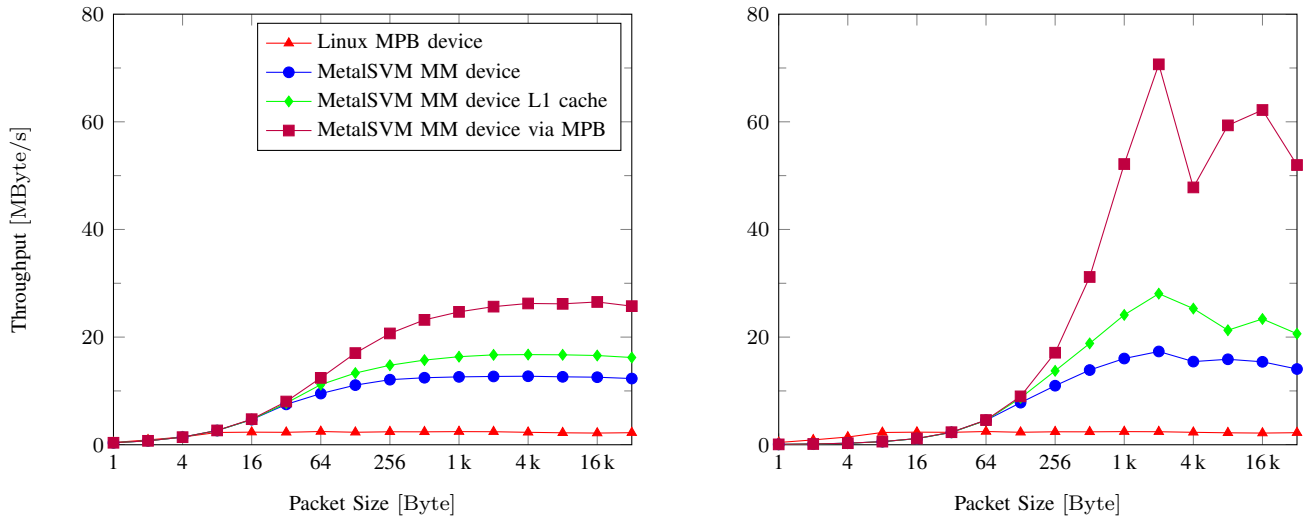
A. Benchmark Results

All diagrams of this section show the throughput average by different package size from small packages of 1 Byte up to large packages of 32 kByte. The test platform has been configured with a core frequency of 533MHz, a memory and mesh frequency of 800MHz. The driver uses as receive buffer size either 8 kByte for the off-die or 7 kByte for the on-die memory. For the evaluation of the performance of *MetalSVM*'s IP stack the established benchmark *netio*[3] has been used.

First of all, we present the results of our eMAC driver in comparison to the driver of SCC Linux. We used a standard configured SCC and MCPC from the *MARC Data Center*. Figure 3b shows the throughput from MCPC to SCC and Figure 3a illustrates the inverse direction. By enabling the cache for the receiving buffer of the SCC, the sending throughput of MCPC is increased by factor 5. These results document the huge impact of the MBPT flag.

Figure 4a shows the performance of the inter-core communication using the full IP stack. The performance of the current Linux driver is added as a reference.

[3]http://www.ars.de/ars/ars.nsf/docs/netio



(a) Full lwIP version

(b) Bypassing version

Fig. 4: Sending Throughput from Tile 0 to Tile 1

It can be noticed that the current Linux driver shows a poor performance and should be improved. All versions of our driver, which optionally use the off-die memory —●—, the off-die memory with enabled L1 cache —◆— or the message passing buffers —■— perform clearly better than the standard Linux driver, which also uses the message passing buffer as transport medium.

Figure 4b shows the results of bypassing the IP stack. When the throughput of the bypassing version is compared with the throughput of the lwIP versions it can be noticed that bypassing the IP stack results in a higher peak performance. However, regarding small packets below a size of 256 Bytes the lwIP version benefits from the usage of Nagle's algorithm that combines small packages. [16] The maximum of the throughput —■— is reached at a package size of 2 kByte. Here, the package size is the largest size to the power of two that fits twice into the message passing buffer regarding the requirements of the RCCE library.

VIII. Application

For the demonstration of our SVM system we have chosen a classical synchronous iteration program example. The heat distribution of square metal sheet with known temperatures at its edges represents a two-dimensional Laplace problem. Figure 5 illustrates the further described method.

The resulting partial differential equation can be solved with the common Jacobi Over Relaxation (JOR) algorithm standing for a simple parallel program example using a shared memory approach. The Jacobi iterations can be described by the following formula:

$$u_{i,j}^{k+1} = \frac{1}{4} \cdot [u_{i-1,j}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i,j+1}^{k}]$$

An analysis of the capabilities offered by the *MetalSVM* layer is reached by executing kernel threads in the *MetalSVM* kernel. Therefore, the function `svm_alloc` is used in a collective way to allocate a shared memory region with Level 1 cache enabled.

Allocated memory is used as follows: The simulation data of 1024×512 **`double`** values are stored in two arrays namely `old` and `new`. After each iteration the values from `new` are moved to `old` by exchanging the references. A barrier follows to ensure that iterations are processed synchronously. We used the linear barrier implementation of the RCCE library. A static distribution to n cores of the squared problem size is used. Each core iterates over N/n lines. The shared memory application assumes a synchronous behavior after each iteration which creates the requirements for an SVM system to provide correct data. Enabled caches have to be flushed and invalidated implicitly, regarding a strong release consistency model, or explicitly, regarding a lazy release consistency. The current version of *MetalSVM* supports both as described in Section VI.

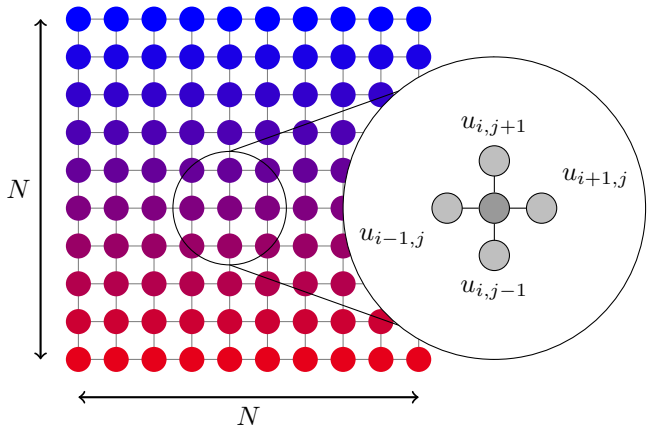


Fig. 5: Heat Distribution Problem

Figure 6 shows benchmark results of the previously described application with a different core count on the SCC platform [4]. Curve —— depicts terms of a message passing laplace variant based on iRCCE [14], which uses a non-blocking behavior to exchange rows after each iteration. Curves —●— and —▲— represent the performance measurements of a strong consistency model. The first setup —●— is the usage of only one memory controller (MC) holding the entire matrix. Here, the well known *memory wall* problem occurs. The consequence is a reduction of the scalability. As a second setup the matrix is statically partitioned to all four MC's to distribute the memory load. The result —▲— is a better scalability up to 8 cores. The scalability has to be improved for the use of more than 8 cores. As a third setup a lazy release memory model has been applied to the given problem. Here, the caches are flushed after each iteration without the generation of an interrupt or an exception. Measurements of this setup —■— show a nearly optimal result.

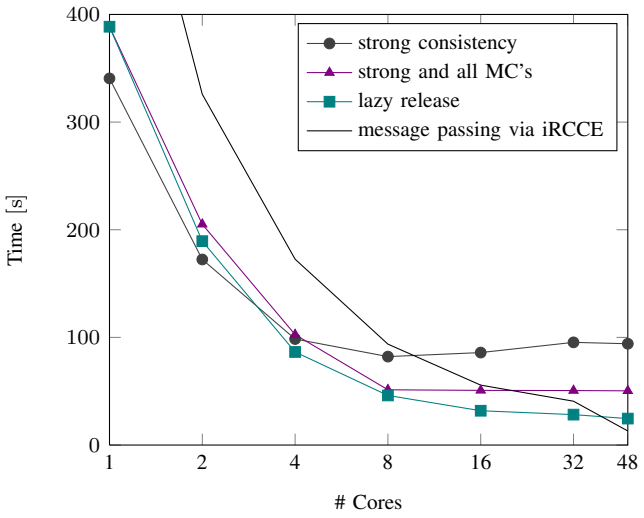


Fig. 6: Laplace Runtimes

Nevertheless, it has to be considered that the JOR algorithm is an extremely stressful example for an SVM system. Here, the barrier after each iteration leads to a synchronized access of all cores to their neighbors' data. In the case of a lazy consistency, the majority of cores send a request mail and IPI to its neighbor just after the synchronization point. Certainly, for such an extremely stressful example the results are excellent. The linear runtime of the shared memory application is approximately half of the linear runtime of the message passing application. What shows the impact of the write combining buffer. In this experiment the message passing application reaches a super-linear speedup in a region of 32 to 48 cores by using the L2 Cache. Here, the problem size fits into the L2 Cache.

[4]core/mesh/memory frequency of 533/800/800 MHz

IX. Conclusions and Outlook

In this paper, we have presented our first steps to design and implement a strong memory model for the SVM system that has been integrated into *MetalSVM*. The basic concept is based on a mailbox system with a low-latency inter-kernel communication layer. First benchmark results of our SVM system prototype are promising. In fact, the overhead of the Strong Release Consistency compared to the Lazy Release Consistency Model is tolerable. Moreover, this paper shows that the current drivers of SCC Linux's IP stack have potential for improvement. In the majority of the presented benchmarks the IP stack of *MetalSVM* reaches a significantly better performance.

In the future, we will investigate other, weaker memory models, to achieve the best performance for our bare-metal hypervisor. We plan to use experiences [17] from the design of kernel extensions for NUMA systems to reach a more dynamic memory distribution strategy like *Affinity-on-Next-Touch* [18]. In addition, improvements regarding the scalability of our synchronization layer and the collective operations ,provided by MetalSVM, are in progress.

We aim for the nearer future to increase of the usability of *MetalSVM* to address a broader audience. Besides, we recommend an integration of our improved IP solution back to SCC Linux so that all MARC members can benefit from this work.

Acknowledgment

The research and development is funded by Intel Corporation. The authors would like to thank especially Ulrich Hoffmann, Michael Konow and Michael Riepen of Intel Braunschweig for their help and guidance.

REFERENCES

[1] *SCC External Architecture Specification (EAS)*, Intel Corporation, November 2010, Revision 1.1.

[2] P. Reble, S. Lankes, C. Clauss, and T. Bemmerl, "A Fast Inter-Kernel Communication and Synchronization layer for MetalSVM," in *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011.

[3] 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.

[4] 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 International Conference on High Performance Computing and Simulation (HPCS2011), Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, Istanbul, Turkey, July 2011.

[5] M. Dormanns, K. Scholtyssik, and T. Bemmerl, "A Shared-Memory Programming Interface for SCI Clusters," in *SCI: Scalable Coherent Interface*, H. Hellwagner and A. Reinefeld, Eds. Springer Verlag, 1999, pp. 281–290.

[6] IEEE, Ed., *Standard for Scalable Coherent Interface (SCI)*, ser. IEEE Standards. The Institute of Electrical and Electronics Engineers, Inc., 1992, no. 1596.

[7] S. Paas, T. Bemmerl, and K. Scholtyssik, "Win32 API Emulation on UNIX for Software DSM," in *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, Washington, USA, August 1998, pp. 39–46.

[8] K. Scholtyssik and M. Dormanns, "Simplifying the use of SCI shared memory by using software SVM techniques," in *Proceedings of 2. Workshop Cluster Computing*, Karlsruhe, Germany, March 1999.

[9] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 13–21.

[10] M. Chapman and G. Heiser, "vNUMA: A virtual shared-memory multiprocessor," in *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, Jun 2009, pp. 349–362.

[11] M. van Tol, "SCC L2 flush routine," http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=195.

[12] T. Mattson and R. van der Wijngaart, *RCCE: a Small Library for Many-Core Communication*, Intel Corporation, May 2010, Software 1.0-release.

[13] *The sccKit 1.4.x User's Guide*, Intel Labs, October 2011.

[14] C. Clauss, S. Lankes, T. Bemmerl, J. Galowicz, and S. Pickartz, *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*, Chair for Operating Systems, RWTH Aachen University, July 2011, Users' Guide and API Manual.

[15] A. Dunkels, *Design and Implementation of the lwIP TCP/IP Stack*, Swedish Institute of Computer Science, 2001.

[16] J. Nagle, "Congestion control in IP/TCP internetworks," *SIGCOMM Computer Communication Review*, vol. 14, no. 4, pp. 11–17, 1984.

[17] S. Lankes, B. Bierbaum, and T. Bemmerl, "Affinity-On-Next-Touch: An Extension to the Linux Kernel for NUMA Architectures," in *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics (PPAM 2009), Workshop on Memory Issues on Multi- and Manycore Platforms, Springer Berlin / Heidelberg, Volume 6067/2010 of LNCS*, Wroclaw, Poland, 2010, pp. 576–585.

[18] L. Noordergraaf and R. van der Pas, "Performance Experiences on Sun's WildFire Prototype," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Portland, Oregon, USA, November 1999.

Parallel AI Planning on the SCC

:

Vincent Vidal, Simon Vernhes, and Guillaume Infantes

***Abstract*—We present in this paper a parallelized version of an existing Artificial Intelligence automated planner, implemented with standard programming models and tools (hybrid OpenMP/MPI). We then evaluate this planner with respect to its sequential version through extensive experiments over a wide range of academic benchmarks, on two different target architectures: a small standard cluster and the research processor SCC ("Single-chip Cloud Computer") developed by Intel Labs and made available to the research community through the MARC program ("Many-core Applications Research Community"). We obtain interesting speedups (super-linear in some cases) on both architectures. Interestingly enough, these experiments also reveal different behaviors between the cluster and the SCC.**

I. Introduction

Automated Planning in Artificial Intelligence [1] is a general problem solving framework which aims at finding solutions to combinatorial problems formulated with concepts such as actions, states of the world, and goals. For more than 50 years, research in Automated Planning has provided mathematical models, description languages and algorithms to solve this kind of problems. We focus in this paper on Classical Planning, which is one of the simplest model but has seen spectacular improvements in algorithm efficiency during the last decade.

The sequential planning algorithm that will form the basis of our parallel algorithm has been implemented in the YAHSP2 planner [2][3] which participated to the 4th and 7th International Planning Competitions[1] (IPC) in 2004 and 2011. It uses a forward state-space heuristic search algorithm with relaxed plan extraction inspired by the FF planner [4]. The main differences with FF are that (1) the search algorithm is a complete weighted-A* algorithm [5] (while FF first tries an incomplete one), (2) the heuristic function is based on h^{add} [6] instead the length of the relaxed plan length and (3) at each node of the search, a lookahead strategy is performed before classical node expansion in order to try to reach a node closer to a goal state, in a computationally easy way by using actions from the relaxed plan.

The parallelization scheme we propose is based on the principle already used in TDS [7] and HDA* [8]: to distribute search nodes among the processing units (PUs) based on a hash key computed from planning states. In this way, the list of nodes to be expanded (the open list) owned by each PU are disjoint: computations made on a given state (applicable actions, heuristic function, lookaheads, etc.) are performed only once, by the PU the node belongs to. Another important aspect is that communication between PUs can be performed in an asynchronous way: a PU expands nodes from its open list, sends sons to the PUs they belong to, and periodically checks its incoming messages to incorporate new nodes into its open list (between OpenMP threads, this last step is seamlessly performed by writing to shared memory).

We evaluate the performance of the parallel algorithm with respect to its sequential version over a wide range of academic benchmarks issued from the IPCs, on two architectures: a small standard cluster composed of four 12-core servers (48 cores in total), and the research processor SCC ("Single-chip Cloud Computer") embedding 48 cores on a single chip developed by Intel Labs. These experiments show that interesting speedups, sometimes super-linear, are obtained thanks to the parallelization. Unfortunately, some super-linear speed-downs are also observed, which suggests some improvements to the parallelized algorithm that could combine the advantages of both. This behavior was not unexpected, as the parallelized algorithm does not perform the same computations as the sequential version: the order of node evaluation being modified, the search space is not explored the same way, which can help or deserve the parallel algorithm.

The paper is outlined as follows. After introducing the research domain of Classical Planning in Artificial Intelligence and the mathematical STRIPS model of planning, we present the sequential algorithm implemented into the YAHSP2 planner. We then briefly explain the principles of the parallelization we propose, and the main modifications of the sequential algorithm. After having described the experimental evaluation, we conclude and draw some future works.

II. Background on Classical Planning in AI

Classical Planning is about finding a sequence of actions (possibly optimal) leading from an initial state towards a defined goal. We make some assumptions about the world:

- finite number of possible states of the world,
- full observability: one always know the state of the world,
- determinism: the result of applying an action to a state s is always a single state s'.

An example of an Automated Planning problem is described in Figure 1. There, a robot arm can move a single block at a time. It is able to `unstack` two blocks by taking the upper one; `stack` a block on another; `pick-up` a block from the table or `put-down` a block on the table. A planning algorithm should find a plan (a sequence of defined actions) that the robot can execute to reach the goal state from the initial one.

Planning is hard, in our case it has been shown to be PSPACE-complete [9]. The major problem for planning algorithms is to deal with the combinatorial explosion of the number of states during search.

All authors are working at Onera, the French Aerospace Lab, in the DCSD department, Toulouse center. Email addresses: first-name.last-name@onera.fr.

This work has been funded by the Onera research program PR-SCC and supported by Intel Labs through a research proposal for working with Intel SCC and the Many-core Applications Research Community (MARC).

[1]See http://ipc.icaps-conference.org/ for more information about the IPCs.

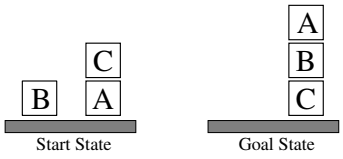


Fig. 1. An Automated Planning classic domain: BlocksWorld

a) PDDL (Planning Domain Definition Language): PDDL [10] is a language commonly used to represent planning problems, as for instance in IPCs. It helps to compare planners with well-established benchmarks[2] (over 40 different application domains and several thousand instances).

The operator `stack` of the previous domain (Figure 1) written using PDDL syntax is shown below:

```
(:action stack
 :parameters  (?ob ?underob)
 :precondition (and (clear ?underob) (holding ?ob))
 :effect (and (arm-empty) (clear ?ob)
          (on ?ob ?underob) (not (clear ?underob))
          (not (holding ?ob))))
```

After parsing a PDDL problem, planners transform the PDDL first-order language into a set-theoretic representation (sets of propositions) like STRIPS (see below). To do so, PDDL operators, like `(stack ?ob ?underob)`, are instantiated into ground actions {`(stack A B)`, `(stack A C)`, ...}.

b) The STRIPS model of Classical Planning: Planning problems can be expressed into the STRIPS model defined as follows. A *state* of the world is represented by a set of ground atoms. A *ground action* a built from a set of atoms A is a tuple $\langle pre(a), add(a), del(a)\rangle$ where $pre(a) \subseteq A$, $add(a) \subseteq A$ and $del(a) \subseteq A$ represent the preconditions, add effects and del effects of a respectively. A *planning problem* can be defined as a tuple $\Pi = \langle A, O, I, G\rangle$, where A is a finite set of *atoms*, O is a finite set of ground actions built from A, $I \subseteq A$ represents the *initial state*, and $G \subseteq A$ represents the *goal* of the problem. The *application* of an action a to a state s is possible if and only if $pre(a) \subseteq s$ and the resulting state is $s' = (s \setminus del(a)) \cup add(a)$. A *solution plan* is a sequence of actions $\langle a_1, \ldots, a_n\rangle$ such that for $s_0 = I$ and for all $i \in \{1, \ldots, n\}$, the intermediate states $s_i = (s_{i-1} \setminus del(a_i)) \cup add(a_i)$ are such that $pre(a_i) \subseteq s_{i-1}$ and $G \subseteq s_n$.

c) Prior work on Automated Planning: Different approaches exist [1]. One of the most successful for suboptimal planning is state-space search where each node corresponds to a state of the world and edges between nodes are applicable actions which allow to move from a state s to a state s' (state transition). Finding a path from the initial state I (node) to the goal state G provides a plan for a problem. Heuristic search algorithms like A* are mainly used to find such a path. Various domain-independent heuristics have been developed to guide search. Many successful state-of-the-art sequential planners are based on Fast Downward [11].

Several approaches to parallel planning have been proposed in recent years. Most of them are modifications of the A* algorithm, trying to transform sequential planning techniques into parallel ones. Some algorithms use a distributed hash function to allocate generated states to a unique processing unit and avoid unnecessary state duplications, like HDA* [8]. Parallel Frontier A* with Delayed Duplicate Detection [12] uses a strategy based on intervals computed by sampling to distribute the workload among several workstations, targeting distributed-memory systems. The Adaptive K-Parallel Best-First Search [13] algorithm presents an asynchronous parallel search for multi-core architectures. This paper also provides a recent bibliography about parallel planning. In the IPC 2011 competition, a multi-core track has been started. The most efficient planners were the ones using a portfolio-based approach, meaning they run different planners (or the same planner with different configurations) on each processor (or core) like ArvandHerd [14] and ay-Also-Plan Threaded [15].

[2]The benchmark problems used in past planning competitions are all available on the IPC webpages

Algorithm 1: plan-search

input : a planning problem $\Pi = \langle A, O, I, G\rangle$ and a weight ω for the heuristic function
output : a plan if search succeeds, $\perp$ otherwise

```
1  open ← closed ← ∅
2  create a new node n:
3    n.state ← I
4    n.parent ← ⊥
5    n.steps ← ⟨⟩
6    n.length ← 0
7  n' ← compute-node(Π, ω, n, open, closed)
8  if n' ≠ ⊥ then return extract-plan(n')
9  else
10   while open ≠ ∅ do
11     n ← arg min_{n∈open} n.heuristic
12     open ← open \ {n}
13     foreach a ∈ n.applicable do
14       create a new node n':
15         n'.state ← (n.state \ del(a)) ∪ add(a)
16         n'.parent ← n
17         n'.steps ← ⟨a⟩
18         n'.length ← n.length + 1
19       n'' ← compute-node(Π, ω, n', open, closed)
20       if n'' ≠ ⊥ then return extract-plan(n'')
21   return ⊥
```

III. The sequential planning algorithm

Algorithm 1 (`plan-search`) constitutes the core of the best-first search algorithm (a weighted-A* here). The first call to `compute-node` may find a solution to the problem without search, by recursive calls to the lookahead process. If not, nodes are extracted from the open list following their heuristic evaluation and are expanded with the applicable actions (already computed and stored in nodes inserted into the open list), and a solution plan is returned as soon as possible. Search can be pursued in an anytime way, in order to improve the solution, with pruning of partial plans whose quality is lower than that of the best plan found so far. In our experiments, the weight ω has been set to 3.

Algorithm 2 (`compute-node`) first performs duplicate state detection. It then computes the heuristic, checks if the goal is obtained or cannot be reached, and updates the node with the heuristic and the applicable actions given by `compute-hadd`. The node is then stored in the open list and a lookahead

Algorithm 2: compute-node

input : a planning problem $\Pi = \langle A, O, I, G \rangle$, a weight ω for the heuristic function, a node n, the $open$ and $closed$ lists
output : a goal node if search succeeds, $\perp$ otherwise; $open$ and $closed$ are updated

```
1  if ∃ n' ∈ closed | n'.state = n.state then return ⊥
2  else
3      closed ← closed ∪ {n}
4      ⟨cost, app⟩ ← compute-hadd(Π, n.state)
5      gcost ← Σ_{g∈G} cost[g]
6      if gcost = 0 then return n
7      else if gcost = ∞ then return ⊥
8      else
9          n.applicable ← app
10         n.heuristic ← n.length + ω × gcost
11         open ← open ∪ {n}
12         ⟨state, plan⟩ ← lookahead(Π, n.state, cost)
13         create a new node n':
14             n'.state ← state
15             n'.parent ← n
16             n'.steps ← plan
17             n'.length ← n.length + length(plan)
18         return compute-node(Π, ω, n', open, closed)
```

state/plan is computed by a call to `lookahead`. A new node corresponding to the lookahead state is then created and `compute-node` is recursively called. Recursion is stopped when a goal, duplicate or a dead-end state is reached.

The other algorithms are not shown here due to lack of space (more details can be found in [3]), but their role can be summarized as follows. Algorithm `compute-hadd` computes h^{add} and returns a vector of costs for all atoms and actions, as well as actions applicable in the state for which h^{add} is computed obtained as a side-effect. Algorithm `lookahead` computes a lookahead state/plan from a relaxed plan given by a call to `extract-relaxed-plan`. Once a first applicable action of the relaxed plan is encountered, it is appended to the lookahead plan and the lookahead state is updated. A second applicable action is then sought from the beginning of the relaxed plan, and so on. When no applicable action is found, a repair strategy tries to find an applicable action of minimum cost from the whole set of actions, in order to replace an action of the relaxed plan which produces an unsatisfied precondition of another action of the relaxed plan, and the process loops. Algorithm `extract-relaxed-plan` computes a relaxed plan from a vector of action costs. A sequence of goals to produce is maintained, starting from the goals of the problem. The first one is extracted, and an action which produces it with the lowest cost is selected and stored in the relaxed plan. Its preconditions are appended to the sequence of goals, and the process loops until the sequence of goals is empty. An atom already satisfied, i.e. produced by an action of the relaxed plan, is not considered twice. The relaxed plan is finally sorted before being returned, by increasing costs first, and for equal costs by trying to order first an action which does not delete a precondition of the next action.

IV. An hybrid OpenMP/MPI Parallel Planning Algorithm

The main idea for parallelizing YAHSP2 is based on the same principle than in TDS [7] and HDA* [8]: to distribute search nodes among the PUs based on a hash key computed from planning states. A PU can either be an MPI process running a single thread, or an OpenMP thread started with several others by an MPI process.

One important consequence of the hash-based distribution principle is that several occurrences of a given state, encountered in any PU, will be sent to the same PU that will either discard it if it has already been encountered, or expand it in the opposite case. Another consequence is that this communication scheme can be performed in an asynchronous way: PUs send nodes that do not belong to them (i.e. the state hash key identifies another PU), and receive nodes from any other PUs, while expanding nodes they currently own in their open list.

The main differences with respect to TDS and HDA* are that (1) we focus on suboptimal planning, while TDS and HDA* search optimal plans, (2) we have integrated the lookahead strategy into this framework, and (3) we implemented this principle as an hybrid OpenMP/MPI algorithm (while TDS and HDA* only use MPI). The advantages of using OpenMP are that problem parsing, instantiation, and all other preprocessing tasks are performed only once (thus saving memory), and communication between threads by shared memory is much more efficient than communication between MPI processes. The main drawback of using OpenMP is that memory locks are sometimes necessary; but fortunately, this does not happen often because the algorithm spends most of its time in computing the h^{add} heuristic (Algorithm 2 line 4).

Each PU (either an MPI process running a single thread, or an OpenMP thread inside an MPI process) runs the search algorithm described in Algorithm 1, with its own open and closed lists, with several modifications:

The first initial node (lines 2–6) is created only by the master thread of the first MPI process.

The main loop condition (line 10) is modified in order for the loop to be executed even if the PU has no node yet (i.e., it is waiting for states sent by other PUs). This loop is stopped when a PU finds a solution, which will be handled by special messages. We have not yet implemented a distributed termination algorithm in the case where the search space is completely explored without finding a solution (which happens extremely rarely on academic benchmarks). In HDA*, a termination algorithm from [16] has been used.

Calls to `compute-node` (lines 7 and 20) are performed only if the corresponding nodes belong to the current PU; in the opposite case, they are sent to their associated PU. This is performed by either sending a message to another MPI process, or by incorporating the node into the open list of another thread within the same MPI process. In the latter case, it is required to lock the open list of the destination thread.

Before choosing the next node to be expanded (line 11), incoming MPI messages are checked and all new nodes are incorporated into the open list: either the open list of the current PU, or into the open list of another thread of the same MPI process –which requires once again a lock on this list.

In order to completely follow the hash-based distribution principle, Algorithm 2 should also be modified in order to send nodes to their appropriate PUs (as in the third point above): line 18 should be executed only in the case where the obtained

node belongs to the current PU, and if not, this node should be sent to its correct destination. However, after some preliminary experiments, we observed that the strategy of performing full lookaheads –i.e. the full recursive calls of `compute-node` inside a single PU– was more efficient than distributing them. One consequence is that a given node may appear in different PUs, thus duplicating the work of expanding it. Many other variations and strategies can be imagined, and the description and comparison of various node distribution policies will be the subject of a more extensive study.

The last main modification of the sequential algorithm is about the reconstruction of the solution, which is distributed among the different PUs. Indeed, when nodes are communicated between PUs, the actions attached to it (which represent the path from a node to its son) are kept in the PU they are computed, in order to minimize the traffic. All messages thus have the same size, as states are represented with bit arrays whose size is the number of ground atoms of the problem, which is determined during the planning problem instantiation. When a PU finds a solution, it sends a special message to all nodes meaning that a synchronization step is required. These messages are checked in the same place than MPI incoming messages are treated (before line 11 of algorithm 1). In the case of synchronization, a function is called by all PUs at this place, into which they exchange messages to build the solution plan and aggregate statistics on the current run (everything being controlled and owned by the master thread of the first MPI process). This procedure was a bit tricky to implement, but do not deserves more details in this paper. One important remark though is that all PUs play exactly the same role in the algorithm, except in two minor cases where the master thread of the first MPI process plays a special role: when search starts (initial node of the problem) and when a solution is built.

V. Experimental Evaluation

In order to evaluate the different parallel implementations of the algorithm, we conducted a set of experiments with 1171 benchmarks from the 3rd to the 7th IPCs (all sequential problems from these IPCs). The cluster is composed of 4 servers with two 6-core Intel Xeon X5670 running at 2.93GHz and 24GB of RAM. The different configurations are:

- c1: 1 process with the sequential algorithm;
- c48: 48 MPI processes uniformly distributed;
- c4x12: 4 MPI processes, each one of them including 12 OpenMP threads, uniformly distributed (cluster only);
- c48-I and c4x12-I: 48 independent MPI processes or 4 MPI processes including 12 independent threads (cluster only) executing the sequential algorithm in exactly the same way with no communication (all PUs are equivalent to configuration c1 and perform identical computations on the same data), in order to assess the impact of memory contention.

In the following, we compare absolute (wall-clock) time used for finding a solution with a timeout of 600 sec., and the ratio of search times used by compared versions. This will be shown as *speedup* in the figures. Thus, the shown speedup will be the amount of time used by the quickest implementation divided by the amount of time used by the slowest one.

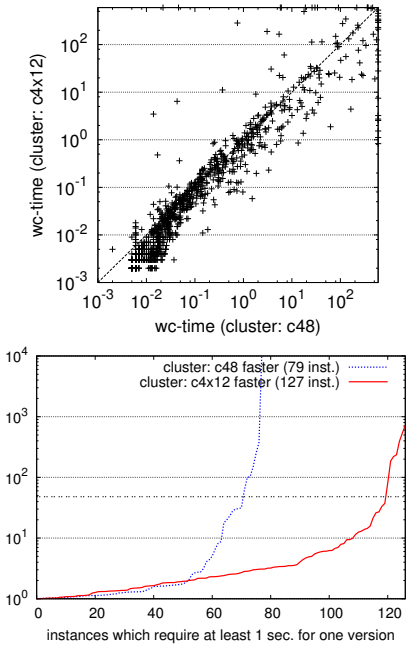


Fig. 2. Comparison of the wall-clock time in seconds between cluster versions (either 48 MPI processes, or 4 MPI processes of 12 threads each) and speedup of both versions (Blue curve –dotted lines– represent the speedup of the 48 processes version when it is faster, while red curves –plain lines– represent the speedup of the 4x12 version when it is faster.).

a) Cluster versions compared: Figure 2 compares c48 and c4x12 on the cluster. On the top figure, the solving time is compared, problem by problem. As most of the points are on the bottom right part, we can deduce that c48 performs worse. While the number of threads is the same, the overhead caused by the MPI message passing mechanism makes this version generally worse than c4x12. In further experiments, we then will only compare c4x12 to c48 on the SCC. The bottom figure shows the speedup of both versions, for the problems were the particular version performs better. Again, it can be seen that c4x12 performs better in a larger number of cases than c48. Interestingly, when problems become harder, the speedup can become extremely large: one of the versions typically go around 10000 times faster than the other one.

b) Parallel vs. sequential: On Figure 3 can be seen the comparison between the sequential version and the parallel implementation. Hopefully, the parallel version performs better than the sequential one as soon as the problem is complex enough to take more time to solve than the overhead induced by communication mechanisms. Another reason for the parallel version not to always perform better is that the order in which nodes are explored is not the same, and the aim of the heuristic used is to make the sequential version use a very good order, while in parallel version there is much more variation around the order implied by the use of the heuristic value. One can also remark a very large number of problems unsolved before timeout for the sequential version, especially on the SCC: they are the many points on the right frame.

c) Detailed speedup analysis: Figure 4 shows the comparison between c4x12 and c1 on the cluster. We show different curves in order to emphasize the effect of the com-

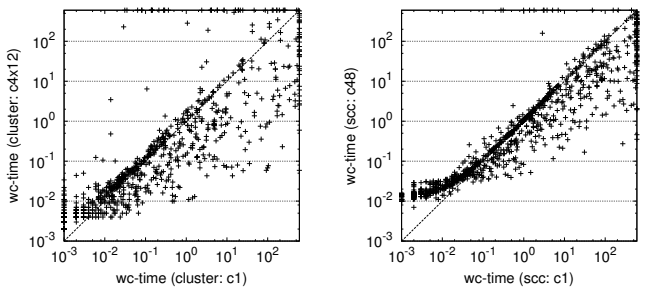


Fig. 3. Comparison of the wall-clock time in second between parallel (4x12 on the cluster and 48 processes on the SCC) and sequential versions.

munication overhead, independently of the parallelism itself; this because communication overhead has a large influence for very simple problems, and becomes less important for larger ones. So we show results for problems taking at least 0.001 second to solve for one of the version, 0.1 second to solve and so on. As expected, the sequential version performs better for problems that can be very quickly solved, but the parallel version becomes generally better as soon as the problems need at least 0.1 second to be solved. On the other hand, this trend becomes less obvious when the problems are very complex (more than 100 seconds to be solved for one version). We think that this is because both versions get trapped into long useless explorations that do not lead to find the goal.

We conducted the very same comparison on the SCC, as shown on Figure 5. Interestingly, the trend observed on the cluster for the larger problems (that the parallel version does not perform better and better compared to the sequential one) is not present here (even with comparable complexity obtained by comparing 30 sec. of cluster time with 600 sec. of SCC time –not shown here–). So the SCC parallel version performs better and better with the problem complexity, whereas the cluster version just performs better, but not better and better.

At this point, we are unsure why this occurs. One explanation is that the amount of data exchanged increases super-linearly with the problem complexity, thus the SCC implementation would be less sensitive to the problem complexity. It may also be the case that all threads being trapped into bad explorations may occur only for a larger timeout...

d) Influence of the amount of data exchanged: In order to figure this out, we present Figure 6, where one can see the speedups related to the amount of data exchanged. This is performed on a selection of 5 problems in each planning domain (210 instances in total), for anytime runs of 100 seconds (search continues after a solution is found, producing solutions of increased quality). In the cluster version, there is a clear trend of worse speedup when the amount of data exchanged increases, whereas there is no correlation for the SCC. Indeed on the cluster, for the instances where the exchanges are about 10GB (nearly 800 Mb/sec) we seems to reach the I/O capacities (1Gib/sec). This seems to be a good explanation of the "less-sensitivity" to the problem complexity of the SCC implementation compared to the 4x12 cluster one.

e) Influence of concurrent resources access: Finally, we present as Figure 7 the speedups in node generation of parallel

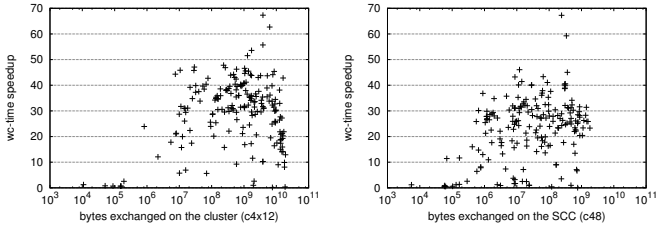


Fig. 6. Wall-clock time speedup of parallel algorithms vs. the sequential version in function of the total number of bytes exchanged between all processes (between the 4 MPI processes) for anytime runs of 100 seconds.

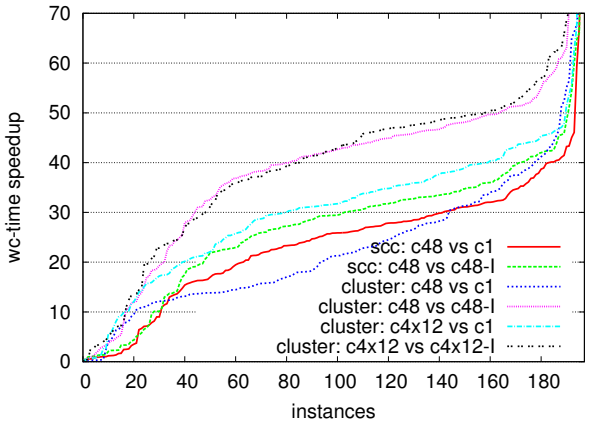


Fig. 7. Wall-clock time speedup in node generation of parallel algorithms vs. sequential versions for anytime runs of 100 seconds, on the cluster and on the SCC. Curves labelled by "architecture: x vs y" compare on the given architecture the speedup of running x processes versus running y processes (single process or 48 non-communicating processes running the same instance in the same way, or 4 non-communicating MPI processes of 12 OpenMP threads each also running the same way).

implementations relative to one sequential process, but also to the same number of sequential processes, in order to see the speedup obtained with a comparable bottleneck for memory access. This is performed in the same experimental conditions as in the previous experiment (anytime search on 210 problems during 100 seconds).

For small instances, the speedup can be small due to the overhead of message passing, while for larger instances, the complexity of problems causes the sequential algorithm to get trapped into exploring non-interesting states for a very long time, making very large speedups. This shows that for complex problems the sequential algorithm would perform better simply by avoiding such traps. More interestingly, the "center" part of the curves, for average instances show very large differences between the SCC and the cluster implementations. More precisely, there is a large difference between the "cluster: c48 vs c1" and the "cluster: c48 vs c48-I" curves (same for "c4x12" versions) meaning that on the cluster there is a lot of memory contention (c48-I is a lot less efficient than c1: only one process). This is less the case for the SCC versions: on the SCC, the sequential non-communicating processes almost do not slow down each other. Several conclusions can thus be stated: on a cluster implementation, good cooperation is mandatory in order to achieve large speedups, in order to reduce memory usage of each core. On the other hand, our

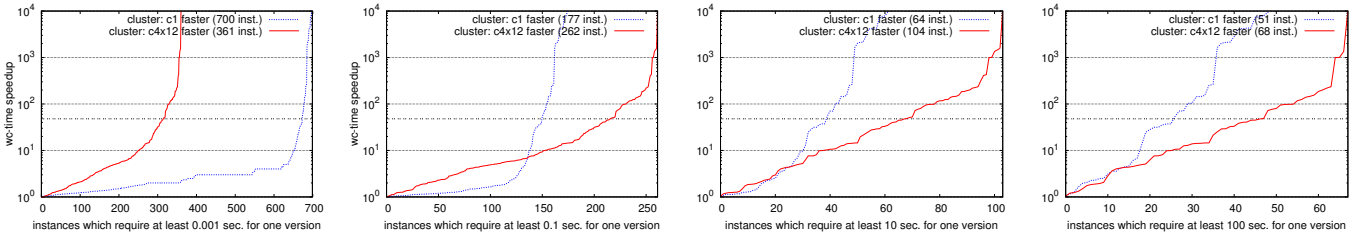


Fig. 4. Wall-clock time speedup of the parallel algorithm with 4 MPI threads of 12 threads each vs. the sequential version running on the cluster, for all instances which require at least a given number of seconds (see x-axis) for one version.

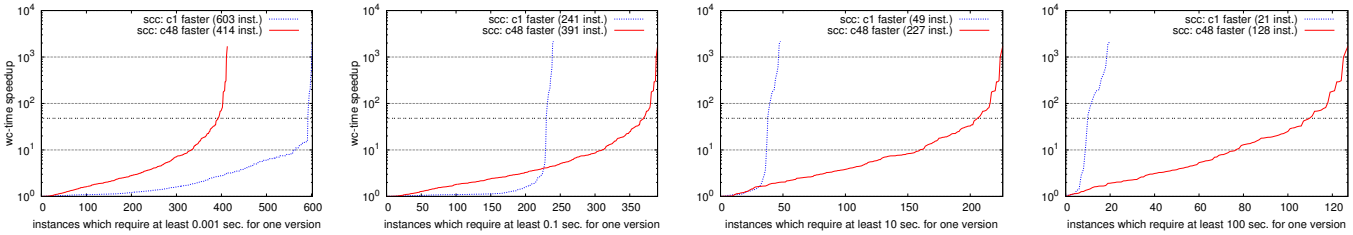


Fig. 5. Wall-clock time speedup of the parallel algorithm with 48 MPI threads vs. the sequential version running on the SCC, for all instances which require at least a given number of seconds (see x-axis) for one version.

implementation achieves a good speedup between the "c48 vs c48-I", meaning that the main bottleneck for improving it is the memory contention problem itself (which will be hard to avoid for our algorithm). On the SCC, this is not a problem, so we can either try to improve the communication scheme for more complementarity, or try a very different "portfolio" approach, where the cores are more independent, and try to solve the problem in different ways.

VI. Conclusion

We described in this paper the parallelization of an automated planner based on forward heuristic search and lookaheads for suboptimal sequential classical planning. It is based on a hash-based node distribution, implemented in hybrid OpenMP/MPI. Experiments show performance improvements with respect to the sequential version, especially for difficult problems. As the search space is not explored the same way in the sequential and parallel versions, super-linear speedups are observed, but also super-linear speed-downs. This suggests trivial improvements of the parallel version, for example by running the sequential version on a single processing unit and the parallel algorithm on the remaining processing units. More elaborate strategies can be imagined, that will make the subject of further studies. The experiments also revealed some differences in the behavior of the parallel algorithm on a standard cluster and on the SCC. These differences suggest that improvements of the parallel version may be more beneficial to an execution on the SCC (which suffers less from memory contention and benefits from faster communications), but clearly more in-depth studies are needed to understand these differences in order to better take advantage of the capabilities of the SCC.

Acknowledgment

The authors would like to thank Intel Labs for providing access to the SCC, and for their reactivity in solving all problems that arose during the SCC exploitation. They also thank Eric Noulard from Onera for insightful discussions.

References

[1] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning, theory and practice*. Morgan-Kaufmann, 2004.
[2] V. Vidal, "A lookahead strategy for heuristic search planning," in *Proc. ICAPS*, 2004, pp. 150–159.
[3] ——, "YAHSP2: Keep it simple, stupid," in *Proc. of the 7th International Planning Competition (IPC'11)*, 2011.
[4] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *JAIR*, vol. 14, pp. 253–302, 2001.
[5] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artificial Intelligence*, vol. 1, no. 3, pp. 193–204, 1970.
[6] B. Bonet, G. Loerincs, and H. Geffner, "A robust and fast action selection mechanism for planning," in *Proc. AAAI*, 1997, pp. 714–719.
[7] J. W. Romein, A. Plaat, H. E. Bal, and J. Schaeffer, "Transposition table driven work scheduling in distributed search," in *Proc. AAAI*, 1999.
[8] A. Kishimoto, A. S. Fukunaga, and A. Botea, "Scalable, parallel best-first search for optimal sequential planning," in *Proc. ICAPS*, 2009.
[9] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, no. 1-2, pp. 165–204, 1994.
[10] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL – The Planning Domain Definition Language," Yale Center for Computational Vision and Control, New Haven, CI, USA, Tech. Rep. CVC TR-98-003/DCS TR-1165, 1998.
[11] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, no. 1, pp. 191–246, 2006.
[12] R. Niewiadomski, J. N. Amaral, and R. C. Holte, "Sequential and parallel algorithms for frontier a* with delayed duplicate detection," in *Proc. AAAI*, 2006.
[13] V. Vidal, L. Bordeaux, and Y. Hamadi, "Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning," in *Proc. 3rd Symposium on Combinatorial Search (SOCS'10)*, 2010.
[14] R. Valenzano, H. Nakhost, M. Muller, and J. Schaeffer, "Arvandherd: Parallel planning with a portfolio," in *Proc. 7th International Planning Competition (IPC'11)*, 2011.
[15] J. Ernits, C. Gretton, and R. Dearden, "Ay also plan: Bitstate pruning for state-based planning on massively parallel compute clusters," in *Proc. 7th International Planning Competition (IPC'11)*, 2011.
[16] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, vol. 2, no. 3, pp. 161–175, 1987.

Performance modeling for power consumption reduction on SCC

:

Bertrand Putigny[1,2], Brice Goglin[1,2], Denis Barthou[2],
[1] Inria
[2] Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France

***Abstract*—As power is becoming one of the biggest challenge in high performance computing, we are proposing a performance model on the Single-chip Cloud Computer in order to predict both power consumption and runtime of regular codes. This model takes into account the frequency at which the cores of the SCC chip operate. Thus, we can predict the execution time and power needed to run the code for each available frequency. This allows to choose the best frequency to optimize several metrics such as power efficiency or minimizing power consumption, based on the needs of the application. Our model only needs some parameters that are code dependent. These parameters can be found through static code analysis. We validated our model by showing that it can predict performance and find the optimal frequency divisor to optimize energy efficiency on several dense linear algebra codes.**

***Index Terms*—Intel SCC, performance model, performance prediction, power, energy efficiency, optimization.**

I. Introduction

Reducing power consumption is one of the main challenge in the HPC community. Indeed power is the leading design constraint for next generation of supercomputers [4]. Therefore energy efficiency is becoming an important metric to evaluate both hardware and software.

The Intel Single-chip Cloud Computer (SCC) is a good example of next generation hardware with an easy way to control power consumption. It provides a software API to control core voltage and core frequency. This opens promising opportunities to optimize power consumption and to explore new trade-offs between power and performance.

This paper aims at exploring the opportunities offered by SCC to reduce power consumption with a small impact on performance. It is organized as follow: Section II describes the model used to predict performance, the Section III demonstrates the reliability of our model by applying it to several basic linear codes, we will also explain how to choose the frequency to optimize a given metric. Sections IV, V, and VI present respectively the related work, future work and conclusion.

II. Performance model

In this section we provide a performance model in order to predict the impact of core frequency scaling on the execution time of several basic linear algebra kernels on the SCC chip.

Project ProHMPT is funded by the French National Agency for Research under the ANR-08-COSI-013 grant.

As we focus on dense linear algebra, we only need a few data to predict a given code performance. The considered datasets being too large to fit in cache, we need the execution time of one iteration of the innermost loop of the kernel and the memory latency.

A. Memory model

To build the memory model, we assume that the application can exploit perfectly data reuse and therefore we assume that each data is accessed only once. We do not take the number of cache accesses into account in the prediction of the overall memory access time because they are not actual memory accesses since the request does not have to go all the way to DRAM. Moreover the cache is not coherent. Therefore there is no overhead due to the cache coherence protocol.

On SCC, a memory access takes 40 core cycles + $4\times n\times$ 2 mesh cycles + 46 memory cycles (DDR3 latency) where n is the number of hops between the requesting core and the memory controller [1]. In our case, we are only running sequential code, therefore we are assuming that the memory access time is $40\times c+46\times m$ cycles, where c is the number of core cycles and m the number of memory cycles. Accessing memory takes 40 core cycles plus 46 memory cycles.

Frequency scaling only affects core frequency, the memory frequency is a constant, (in our case 800MHz). Therefore, changing frequency mostly impacts the code performance if it is computation bound. The number of core cycles to perform one DDR3 access is: $40 + 46 \times \frac{core\_freq}{800}$.

As we can see from the formula dividing the core frequency by 8 (from 800MHz to 100MHz) will only reduce the memory performance by 46%

As the P54C core used in the SCC supports two pending memory requests, we can assume that accessing x elements will take $\frac{x}{2}(40 + 46 \times \frac{core\_freq}{800})$ core cycles.

B. Computational model

In order to predict the number of cycles needed to perform the computation itself we need the latency of each instruction. Agner Fog measured the latency of each x86 and x87 instruction [7]. We used his work to predict the number of cycles to perform one iteration of the innermost loops of each studied kernel. The computation model is very simple, as most of the instructions use the same execution port, there is almost no instruction parallelism. A more complex performance model, considering also measured latencies as a building block of the

Freq divisor	Tile freq (MHz)	Voltage (volts)
2	800	1.1
3	533	0.8
4	400	0.7
5	320	0.6
6	266	0.6
7	228	0.6
8	200	0.6
9	178	0.6
10	160	0.6
11	145	0.6
12	133	0.6
13	123	0.6
14	114	0.6
15	106	0.6
16	100	0.6

TABLE I: Relation between voltage and frequency.

model, is used in the performance tuning tool MAQAO [2]. We use such tool to measure the execution time of one iteration of the innermost loop. As most of the execution time of the codes we consider is spent in inner loops, this performance estimation is expected to be rather accurate.

From this computation model the impact of frequency scaling on the computation performance is straightforward. The number of cycles to perform the computation is not affected by the frequency. Thus, reducing the core frequency by a factor of x will multiply the running time by x.

C. Power model

We use a very simple power model to estimate the power saved by reducing the core frequency. Table I shows the voltage used by the tile for each frequency, these data are provided by the SCC Programmer's guide [1].

The power consumption model used in this paper is the general model:

$$P = CV^2 f$$

where C is a constant, V the voltage and f the frequency of the core. As shown In Table I the voltage is a function of the frequency, thus, we can express the power consumption as a function of the core frequency only.

We choose not to introduce a power model for the memory for two reasons: first we have no software control on the memory frequency at runtime. We can change the memory frequency by re-initializing the SCC platform but not at runtime. Thus, the memory energy consumption is constant and we have no control over it. Therefore it would be almost worthless to complicate our model with such information. The other reason is that until now we used models that can be transposed to other architectures. As the memory architecture of the SCC is very different from more general purpose architecture, its energy model would not fit for those architectures. Thus, the model described in this paper is completely general and can be easily transposed to other architectures.

D. Overall model

In this section we describe how to use both the memory and computational models to predict the performance of a given code.

As the P54C core can execute instructions while some memory requests are pending, we assume that the execution time will be the maximum between the computation time and the memory access time:

$$runtime(f_c) = MAX\left(\frac{computation}{f_c}, mem\_access(f_c)\right)$$

with f_c the core frequency.

With this runtime prediction, we estimate how a code execution is affected by changing the core frequency. Taking the decision to reduce the core frequency in order to save energy can be done with a static code analysis.

As show in Section II-A the memory access performance is almost not affected by reducing core frequency, while reducing core frequency increases dramatically the computation time. From this observation we see that reducing core frequency for memory bound code is highly beneficial for power consumption because it will almost not affect performance while reducing dramatically energy consumption. However, reducing core frequency for compute bound code will directly affect performance.

III. Model evaluation

In this section we compare our model with the real runtime of several regular codes in order to check its validity. We used three computation kernels, one BLAS-1, one BLAS-2 and one BLAS-3 kernels namely dot product, matrix-vector product and matrix-matrix product.

First let us describe how we applied our model to these three kernels: In the following formulas, f_{div} denotes the core frequency divisor (as shown in Table I) and $power(f_{div})$ the power used by the core when running at the frequency corresponding to f_{div} (see Table I). An important point is that we used large data sets that do not fit in cache so as to measure the execution time of the code. Thus, the kernel actually gets data from DRAM and not from caches. However, the matrix-matrix multiplication is tiled in order to benefit from data reuse in cache.

A. Dot product multiplication

For the dot product kernel, the memory access time in cycles is:

$$cycles_{mem}(f_{div}) = size \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

The computation time in cycles is given by:

$$cycles_{comp}(f_{div}) = size \times \left(\frac{body}{unroll}\right),$$

with $body$ the execution time (in cycles) of the innermost loop body and $unroll$ the unroll factor of the innermost loop. In the case shown on Figure 1 $body = 36$ and $unroll = 4$. Then the power efficiency is:

$$power_{eff}(f_{div}) = \frac{flop}{\frac{model(f_{div})}{freq} \times power(f_{div})},$$

with $flop$ the number of floating point operations of the kernel, $model(f_{div})$ the number of cycles predicted by our model and $freq$ the actual core frequency ($\frac{1600}{f_{div}}$). In the case shown on Figure 1,

$$model(f_{div}) = MAX\left(cycles_{mem}(f_{div}), cycles_{comp}(f_{div})\right) = cycles_{mem}(f_{div})$$

Figure 1a shows that the number of cycles for both the memory model and obtained through benchmark decreases when frequency decreases. The reason is that frequency scaling only affects core frequency. For memory bound codes such as dot product, reducing the core frequency reduces the time spent in waiting for memory requests. However, the code is not executing faster, as shown in Figure 1b.

B. Matrix-vector product

Similarly the model for the matrix-vector product is:

$$cycles_{mem}(f_{div}) = \frac{matrix\_size}{2} \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

$$cycles_{comp}(f_{div}) = matrix\_size \times \left(\frac{body}{unroll}\right)$$

With $matrix\_size = 512 \times 1024$ elements, $body = 64$ cycles, and $unroll = 4$ for the case shown on Figure 2.

$$power_{eff}(f_{div}) = \frac{flop}{\frac{model(f_{div})}{freq} \times power(f_{div})}$$

In this case, again, the memory access time is more important than the time for the computation, thus, the runtime is given by the memory access time. (ie. $model(f_{div}) = cycles_{mem}(f_{div})$)

Figure 2a shows that the number of cycles for both the memory model and obtained through benchmark decreases when frequency decreases. The reason is the same as for the dot product.

C. Matrix-matrix product

The model for the matrix-matrix multiplication is:

$$cycles_{mem}(f_{div}) = 3 \times \frac{matrix\_size^2}{2} \times \left(40 + 46 \times \frac{2}{f_{div}}\right)$$

$$cycles_{comp}(f_{div}) = matrix\_size^3 \times \left(\frac{body}{unroll}\right)$$

$$power_{eff}(f_{div}) = \frac{flop}{\frac{model(f_{div})}{freq} \times power(f_{div})}$$

With $matrix\_size = 160$ elements (each matrix is 160×160 elements big), $body = 43$ cycles, and $unroll = 1$ for the case shown on Figure 3.

For this BLAS-3 kernel, as expected, the computation time is bigger than accessing memory, thus, $model(f_{div}) = cycles_{comp}(f_{div})$)

D. Power efficiency optimization

Our objective in this section is to show that thanks to the performance model we built, the frequency scaling that optimizes power efficiency can be selected. Then the higher performance version is chosen among the most power efficient versions.

We can see that the dot and matrix-vector products are memory bound while the matrix-matrix product is compute bound. Power efficiency is measured through the ratio of GFlops/W. The best frequency optimizing power efficiency of those two kind of code are different. For the case of memory bound codes, the core frequency can be reduced by a large divisor as performance is limited by memory bandwidth which is not very sensitive to core frequency. On the contrary, for computation bound codes, the performance in Gflops decreases linearly with the frequency.

Figures 1c, 2c and 3c represent power efficiency in GFlops/W for respectively dot, matrix-vector and matrix-matrix products. They show that our performance model is similar to the measured performance (from which we deducted power efficiency). Power efficiency for matrix-matrix product is optimal from a frequency divisor of 5, to 16. Among those scalings, the best performance is obtained for the scaling of 5 according to Figure 3a. For the dot product 1c, codes are more energy efficient using a frequency scaling of 5, and their efficiency increases slowly as frequency is reduced. According to our performance model, around 25% of Gflops/W is gained from a frequency divisor of 5 to a frequency divisor of 16, and for this change, the time to execute the kernel has been multiplied by a factor 2.33 (according to our model). In reality, these factors measured are higher than those predicted by the model, but the frequency values for optimal energy efficiency, or some tradeoff between efficiency and performance are the same. Note that for divisor lower than 5, energy efficiency changes more dramatically since the voltage also changes.

We have chosen to show how to optimize energy efficiency, but as our model predicts both running time and power consumption for each frequency, it is easy to build any other metric depending on power and runtime and optimize it. Indeed using this model allows to compute the metric to optimize for each frequency divisor and then to choose the one that fits the best the requirement. Even with a very simple model as we presented, we can predict the running time of simple computational kernels within an error of 38% in the worst case.

Our energy efficiency model is interesting because it shows exactly the same inflection points as the curve of the actual execution. This point allows us to predict what is the best core frequency in order to optimize the power efficiency of the target kernel.

It is also interesting to see that even with a longer running time all the kernels (even matrix multiplication which is compute bound) benefits from frequency reduction. This is caused by the following facts:

- The run time of such kernels is proportional to the frequency;
- The power consumption is also proportional to the fre-

quency.

So the energy efficiency does not depend on the core frequency. But the 3 firsts step of frequency reduction also reduce the voltage which has an huge impact on power consumption.

IV. Related Work

Power efficiency is a hot topic in the HPC community and has been the subject of numerous studies, and the Green500 List is released twice a year. Studies carried out at Carnegie Mellon University in collaboration with Intel [6] have already shown that the SCC is an interesting platform for power efficiency. Philipp Gschwandtner *et al.* also performed an analysis of power efficiency of the Single-chip Cloud computer in [11]. However, this work focuses on benchmarking, while our contribution aims at predicting performance according to a theoritical proposed model.

Performance prediction in the context of frequency and voltage scaling has also been actively investigated [5], [10], [12], and the model usually divides the execution time into memory (or bus, or off-chip) [8], [9], instruction and core instruction, as we did in this paper.

Our contribution is slightly different from usual approach as we do not use any runtime information to predict the impact of frequency and/or voltage scaling on performance. As we use static code analysis to predict performance of a kernel, this could be done at compile time it and does not increase the complexity of runtime system. Static Performance prediction has also been used in the context of autotuning. Yotov *et al.* [13] have shown that performance models, even when using cache hierarchy, could be used to select the version of code with higher performance. Besides, In [3], the authors have shown that a performance model, using measured performance of small kernels, is accurate enough to generate high performance library codes, competing with hand-tune library codes. This demonstrates that performance models can be used in order to compare different versions, at least for regular codes (such as linear algebra codes).

V. Future work

The next step for this study is to extend the performance model presented in this paper to parallel kernels. This is much easier on the SCC Chip than on more classical architectures as the cache access time is constant because of its non-coherence. Bandwidth taken by cache-coherency protocol and possible contention are difficult to model in general. Moreover memory contention on NUMA architecture is a difficult problem. Indeed in such architectures, memory contention not only depends on the memory access pattern but also on the process placement. Philipp Gschwandtner *et al.* showed how memory contention on a single memory controller when several cores are accessing it [11]. We believe it would be very interesting to lead the same experiments for several sets of core frequency. Indeed reducing the core frequency could lead to reducing the stress on the memory controller by spacing memory requests.

Also we would like to improve the model in order to take into account that applications are usually composed of several phases, some compute bound phases followed by others that might be memory bound. Enlarging our model to predict what would be the best frequency for each of those phases.

It would also be interesting to develop a framework, inside a compiler or a performance tuning tool such as MAQAO [2], in order to perform the code analysis automatically. This would reduce the time to build the model for new codes, allowing us to do it on a large number of codes.

VI. Conclusion

We have described a method to predict performance of some linear algebra codes on the Single-chip Cloud Computing architecture. This model can predict performance of a given code for all available frequency divisor and using the known relation between frequency scaling and voltage, it can also predict power efficiency. Based on this prediction we can choose what will be the best frequency to run the kernel. We have shown that we can save energy through this method, but it is actually even more powerful: using the running time prediction and the power model we can choose the frequency in order to optimize either the running time, or the power consumption, or the energy efficiency.

References

[1] The scc programmer's guide, 2011.

[2] Denis Barthou, Andres Charif Rubial, William Jalby, Souad Koliai, and Cdric Valensi. Performance tuning of x86 openmp codes with maqao. In Matthias S. Mller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 95–113. Springer Berlin Heidelberg, 2010.

[3] Denis Barthou, Sebastien Donadio, Alexandre Duchateau, Patrick Carribault, and William Jalby. Loop optimization using adaptive compilation and kernel decomposition. In *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, pages 170–184, San Jose, California, March 2007. IEEE Computer Society.

[4] S. Borkar. The exascale challenge. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, pages 2 –3, april 2010.

[5] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 19:1396–1410, October 2008.

[6] R. David, P. Bogdan, R. Marculescu, and U. Ogras. Dynamic power management of voltage-frequency island partitioned networks-on-chip using intel's single-chip cloud computer. In *Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on*, pages 257 –258, may 2011.

[7] Agner Fog. Instruction tables lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. http://www.agner.org/optimize/, 2011.

[8] R. Ge and K.W. Cameron. Power-aware speedup. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –10, march 2007.

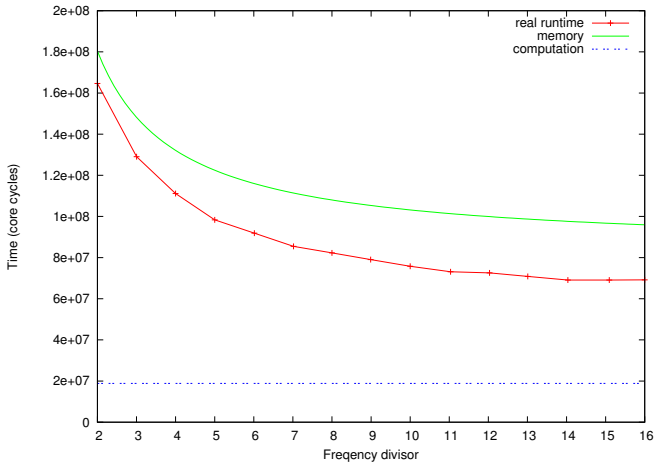
[9] Sang jeong Lee, Hae kag Lee, and Pen chung Yew. Runtime performance projection model for dynamic power management. In *Asia-Pacific Computer Systems Architectures Conference*, pages 186–197, 2007.

[10] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Conf. Computing Frontiers*, pages 287–296, 2010.

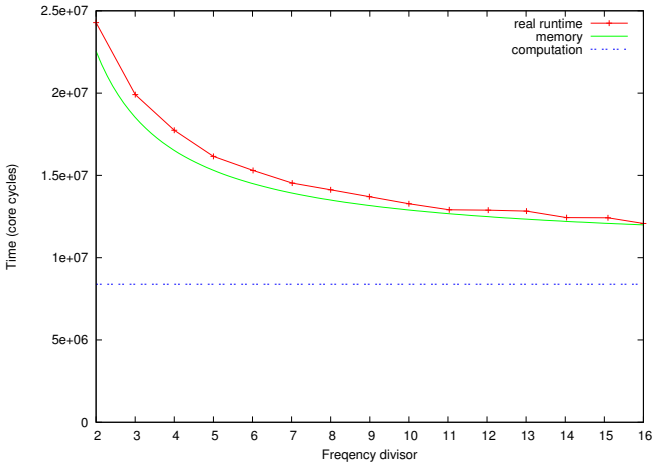
[11] Radu Prodan Philipp Gschwandtner, Thomas Fahringer. Performance analysis and benchmarking of the intel scc. In *Conference on Cluster Computing*, pages 139–149, 2011.

[12] B. Rountree, D.K. Lowenthal, M. Schulz, and B.R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, july 2011.

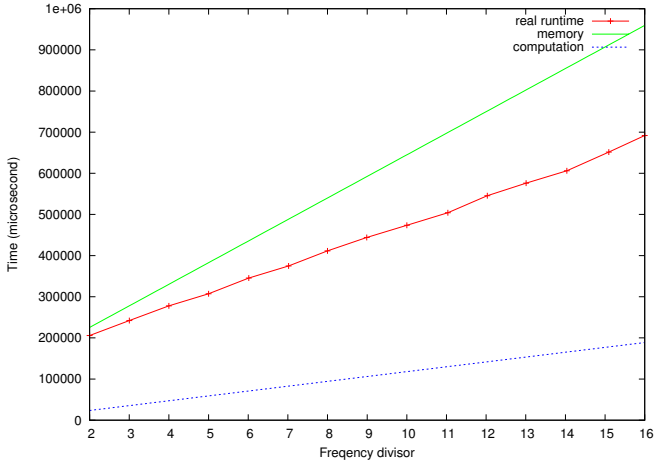
[13] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 63–76, San Diego, CA, June 2003.



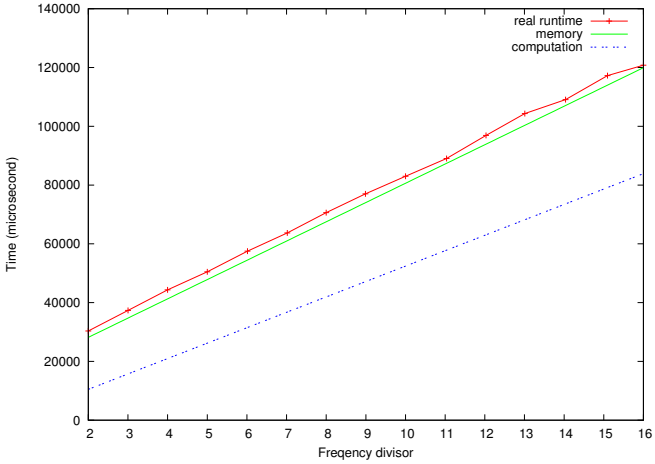
(a) Dot product: the cycle count is shown according to the core frequency divisor



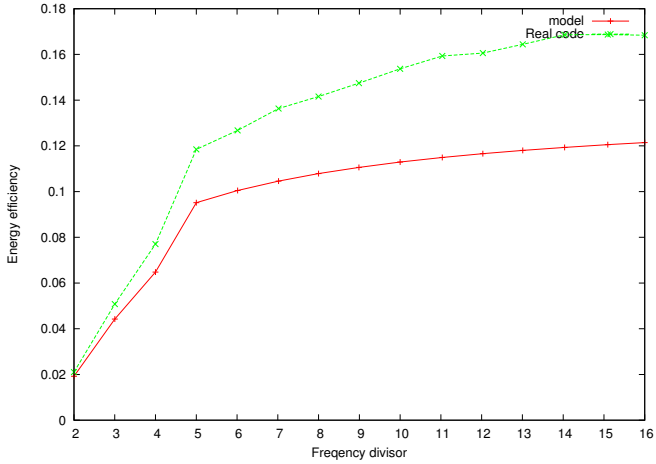
(a) Matrix-vector product: the cycle count is given according to the core frequency divisor.



(b) Dot product: runtime in microsecond depending on the core frequency divisor

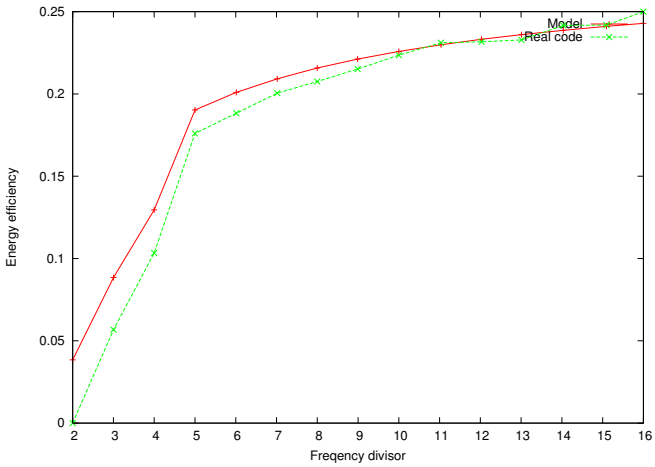


(b) Matrix-vector product: the execution time is given in microsecond depending on the core frequency divisor



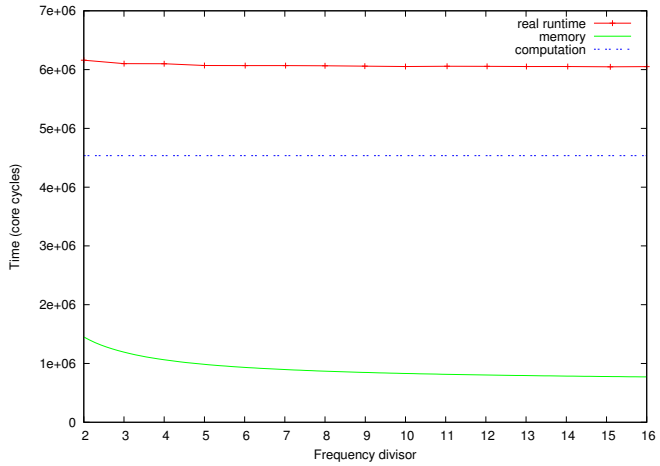
(c) Dot product: power efficiency (in GFlops/W) depending on the core frequency divisor

Fig. 1: Vector dot product model: sequential dot product with 2 vectors of 16 MB.

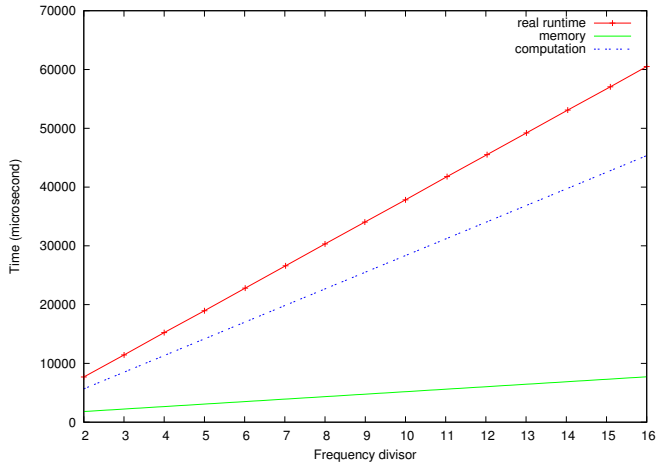


(c) Matrix-vector product: power efficiency (in GFlops/W) depending on the core frequency divisor

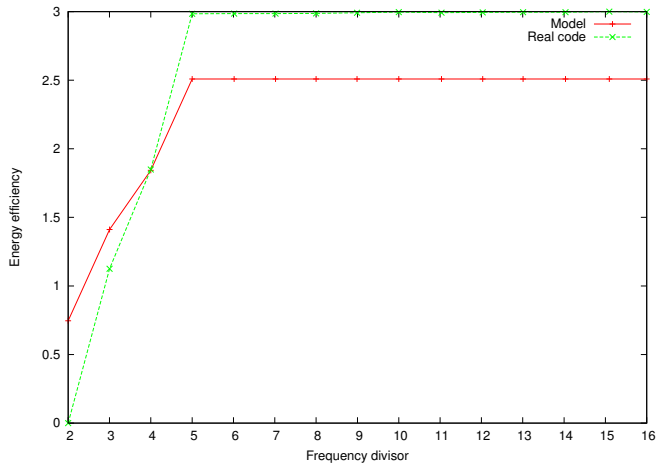
Fig. 2: Matrix-vector multiplication model: sequential code with a 512 by 1024 element size matrix.



(a) Matrix-matrix product model: the cycle count is given according to the the core frequency divisor



(b) Matrix matrix product model: the time in microsecond depending on the core frequency divisor



(c) Matrix-matrix product model: power efficiency (in GFlops/W) depending on the core frequency divisor

Fig. 3: Matrix-matrix multiplication model: sequential code with two matrices of 160 by 160 elements.

Performance and Power Analysis of RCCE Message Passing on the Intel Single-Chip Cloud Computer

:

John-Nicholas Furst Ayse K. Coskun
Electrical and Computer Engineering Department, Boston University, Boston, MA 02215 USA
{jnfurst, acoskun}@bu.edu

***Abstract*— The number of cores integrated on a single chip increases with each generation of computers. Traditionally, a single operating system (OS) manages all the cores and resource allocation on a multicore chip. Intel's Single-chip Cloud Computer (SCC), a manycore processor built for research use with 48 cores, is an implementation of a "cluster-on-chip" architecture. That is, the SCC can be configured to run one OS instance per core by partitioning shared main memory. As opposed to the commonly used shared memory communication between the cores, SCC cores use message passing. Intel provides a customized programming library for the SCC, called RCCE, that allows for fast message passing between the cores. RCCE operates as an application programming interface (API) with techniques based on the well-established message passing interface (MPI). The use of MPI in a large manycore system is expected to change the performance-power trends considerably compared to today's commercial multicore systems. This paper details our experiences gained while developing the system monitoring software and benchmarks specifically targeted at investigating the impact of message passing on performance and power of the SCC. Our experimental results quantify the overhead of logging messages, the impact of local versus global communication patterns, and the tradeoffs created by various levels of message passing and memory access frequencies.**

I. Introduction

Processor development has moved towards manycore architectures in recent years. The general trend is to utilize advances in process technology to include higher numbers of simpler, lower power cores on a single die compared to the previous trend of integrating only a few cores of higher complexity. This trend towards integrating a higher number of cores can be seen in desktops, servers, embedded platforms, and high performance computing (HPC) systems. Future manycore chips are expected to contain dozens or hundreds of cores.

While integrating a high number of cores offers the potential to dramatically increase system throughput per watt, manycore systems bring new challenges, such as developing efficient mechanisms for inter-core communication, creating strategies to overcome the memory latency limitations, and designing new performance/power management methods to optimize manycore system execution. A significant difference of manycore systems compared to current multicore chips comes from the on-chip communication: manycore systems are likely to incorporate a network-on-chip (NoC) instead of a shared bus to avoid severe performance limitations. One method of enabling inter-core communication on a NoC is a message passing interface (MPI).

In order to enable new research in the area of manycore design and programming, Intel Labs created a new experimental processor. This processor, called the "Single-Chip Cloud Computer" (SCC), has 48 cores with x86 architecture. The SCC chip provides a mesh network to connect the cores and four memory controllers to regulate access to the main memory [5]. The SCC includes an on-chip message passing application framework, named RCCE, that closely resembles MPI. RCCE provides multiple levels of interfaces for application programmers along with power management and other additional management features for the SCC [9].

The objective of this paper is to investigate the on-die message passing provided by RCCE with respect to performance and power. To enable this study, we first develop the monitoring tools and benchmarks. Our monitoring infrastructure is capable of logging messages, track performance traces of applications at the core level, and measure chip power simultaneously. We use this infrastructure in a set of experiments quantifying the impact of message traffic on performance and power. Significant findings of this paper are: overhead of our message logging method is negligible; execution times of applications increase with larger distances between communicating cores; and observing both the messages and the memory access traffic is needed to predict performance-power trends.

We present the monitoring infrastructure for the SCC in Section II. Section III describes the applications we developed for SCC. Section IV documents the experimental results on message logging overhead, effects of various message/memory access patterns, and energy efficiency. Section V discusses related work. Section VI concludes the paper.

II. Monitoring Infrastructure for the SCC

Analyzing the message passing system on the SCC requires monitoring performance and power consumption of the system at runtime. As the SCC was designed as a research system it includes special hardware and software features that are not typically found in off-the-shelf multi-core processors. Additional infrastructure is required to enable accurate and low-cost runtime monitoring. This section discusses the relevant features in the SCC architecture and provides the details of the novel monitoring framework we have developed.

Hardware and Software Architecture of the SCC:

The SCC has 24 dual-core tiles arranged in a 6x4 mesh.

Each core is a P54C CPU and runs an instance of Linux 2.6.38 kernel. Each instance of Linux executes independently and the cores communicate through a network interface. Each core has private L1 and L2 caches. Cache coherence is managed through a software protocol as opposed to commonly used hardware protocols. Each tile has a message passing buffer (MPB), which controls the message exchange among the cores. The SCC is connected by a PCI-Express cable to a PC acting as the Management Console (MCPC).

The SCC system has a power sensor used for measuring the full SCC chip power consumption. Power is measured by polling this sensor during application execution.

Each P54C core has two performance counters. These counters can be programmed to track various architectural events, such as number of instructions, cache misses or memory accesses. Performance counters can be accessed from the core in which they are located by reading dedicated registers.

The SCC software includes RCCE, which is a lightweight message passing library developed by Intel and optimized for SCC [9]. It uses the hardware MPB to send and receive messages. At the lower layer, the library implements two message passing primitives *RCCE_put* and *RCCE_get*. These primitives move the data between a local core buffer to the MPB of another core.

Our system setup includes the SCC, the MCPC, and the monitoring framework we developed. On the SCC we implemented utilities to track performance counters, collect power measurements, and log message traffic. On the MCPC we developed software to load desired benchmarks and experimental configurations to the SCC. After running experiments, we analyze the collected data using our automated software.

Software Modules Developed for the SCC:

Performance Counters: To implement performance counter polling we added a patch to the Linux kernel that RCKOS runs. RCKOS is the operating system the SCC cores run and is provided by Intel. Our patch involves a device used for polling and the `ioctl` infrastructure to communicate with the device. The `ioctl` is a system call that is device-specific and allows for user-land access to protected kernel functionality. Once the kernel is patched, RCKOS is recompiled and a new binary image is created and used to flash the cores.

Performance polling is performed by setting the Control and Event Select Register (`CESR`). The P54C cores on the SCC have two registers (0x12, 0x13) allotted for performance counters. The `CESR` contains a 6-bit Event Select field (ES), a Pin Control bit (PC) and a three bit control field (CC) for each of the two counters. The `CESR` is located at `0x11` and is visualized in Figure 1. Two independent events can can be counted by setting the appropriate codes for each ES. The Counter Control is used for enabling / disabling the counters. The `CESR` is programmed through using the Model Specific Registers (MSR) which are available on RCKOS through the `/dev/msr0` device.

To measure L1 cache misses and instructions, we wrote *0xD600CE* to the MSR. For tracking memory access density,

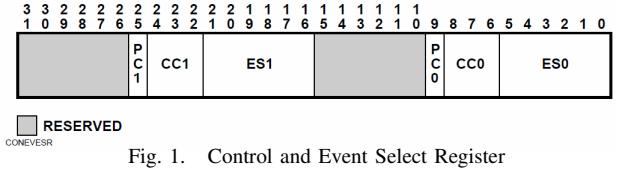


Fig. 1. Control and Event Select Register

we poll the counter "non-cacheable memory reads." We empirically determined non-cacheable memory reads as a good metric for quantifying memory access intensity through measurements with a set of custom designed microbenchmarks that vary in their memory access density [6]. To measure memory accesses and instructions, we wrote *0xD600DE* to the MSR. We ran multiple experiments to collect all three parameters (cache misses, memory accesses, number of instructions). It is also possible to multiplex the register polling to increase the number of parameters collected with little loss of accuracy.

Message Logger: We modified the lower level *RCCE_put* and *RCCE_get* routines in the RCCE library to log the number of messages sent and the source/destination of each message. At the end of each parallel thread the library generates a log containing the communication matrix. Each element in the matrix $\{m_{i,j}\}$ corresponds to the number of messages that $core_i$ has sent to $core_j$. In addition, we program the RCCE library to trigger the logging of the performance counters at the beginning of each of the parallel threads and save the trace at the end of execution.

Software Modules Developed for the MCPC:

- *Stress files and app-loader*: These files contain the benchmark sequences for the tests. For each benchmark, the stress file provides the name, number of threads, and the cores to allocate the benchmark. The *app-loader* loads the files on the SCC to start the experiments. We wrote a set of python scripts that run on the MCPC. These scripts load the stress configuration files and start the RCCE benchmarks in SCC.
- *Post-processing SW*: We designed software for processing the collected data. This script interfaces with the collected data and the stress file. For each benchmark, the script collects the logs and parses them to extract useful statistics. The script then stores the parsed measurements in a MySQL database stored on the MCPC. A custom web-based front end to this database was created to display the results. The data are available for access by Matlab or Excel allowing the implementation of other complex analysis functions.

In this paper we use the monitoring infrastructure described above for analyzing the message passing system on the SCC. The framework can also be leveraged for enabling runtime management policies on the SCC computer.

III. Application Space

We employ a set of benchmarks to evaluate the performance of the SCC and explore a variety of configurations. Two of these benchmarks are Block Tridiagonal (*BT*) and Lower-Upper (*LU*) from the NAS parallel benchmarks (NPB) suite [1]. *BT* and *LU* have been re-programmed for the Intel SCC, and are available to the MARC community. We also

use other benchmarks provided by Intel for the SCC. We build upon the existing benchmarks to create a wider set of operating scenarios in terms of number of cores used and the message traffic. We also design a broadcast benchmark to emulate one to multiple core communication. The complete benchmark set we run in our experiments is as follows.

Benchmarks provided by Intel:

- *BT*: Solves nonlinear Partial Differential Equations (PDE) with the Block Tridiagonal method.
- *LU*: Solves nonlinear PDEs with the Lower-Upper symmetric Gauss-Seidel method.
- *Share*: Tests the off-chip shared memory access.
- *Shift*: Passes messages around a logical ring of cores.
- *Stencil*: Solves a simple PDE with a basic stencil code.
- *Pingpong*: Bounces messages between a pair of cores.

Custom-designed microbenchmark:

- *Bcast*: Sends messages from one core to multiple cores.

The broadcast benchmark, *Bcast*, sends messages from a single core to multiple cores through RCCE. We created the benchmark based on the *Pingpong* benchmark, which is used for testing the communication latency between pairs of cores using a variety of message sizes.

Table I categorizes the Intel benchmarks based on instructions-per-cycle (IPC), Level 1 instruction (code) misses (L1CM), number of messages (Msgs), execution time in seconds, and memory access intensity. All parameters are normalized with respect to 100 million instructions for a fair comparison. Each benchmark in this categorization runs on two neighbor cores on the SCC. The table shows that the *Share* benchmark does not have messages and is an example of a memory-bounded application. *Shift* models a message intensive application and *Stencil* models an IPC heavy application. *Pingpong* has low IPC but heavy L1 cache misses. *BT* has a medium value for all performance values except for the number of messages. *LU* is similar to *BT* except that it has even higher number of messages and the lowest number of L1 code cache misses.

We update the *Stencil*, *Shift*, *Share*, and *Pingpong* benchmarks so that they can run on cores in configurations determining which cores communicate and which cores are utilized. Note that for all configurations of these benchmarks, communication occurs within "pairs" of cores (i.e., a core only communicates to a specific core and to no other cores). The configurations we used in our experiments are as follows:

- *Distance between the two threads in a "pair"*:
  - *0-hops*: Cores on the same tile (e.g., cores 0 and 1)
  - *1-hop*: Cores on neighboring tiles (e.g., cores 0 and 2)
  - *2-hops*: Cores on tiles that are at 2-hops distance (e.g., cores 0 and 4)
  - *3-hops*: Cores on tiles that are at 3-hops distance (e.g., cores 0 and 6)
  - *8-hops*: Cores on corners (e.g., cores 0 and 47)
- Parallel execution settings:
  - *1 pair*: Two cores running, 46 cores idle
  - *2 pairs*: Four cores running, 44 cores idle
  - *3 pairs*: Six cores running, 42 cores idle
  - *4 pairs*: Eight cores running, 40 cores idle
  - *5 pairs*: Ten cores running, 38 cores idle
  - *6 pairs*: Twelve cores running, 36 cores idle
  - *24 pairs*: 48 cores running

  The idle cores run *SCC Linux* but do not run any user applications and they are not in sleep states.
- *Broadcast*: The *Bcast* benchmark is run with one core communicating to N cores, where $1 \leq N \leq 47$.

TABLE I. BENCHMARK CATEGORIZATION. VALUES ARE NORMALIZED TO 100 MILLION INSTRUCTIONS.

Benchmark	L1CM	Time	Msgs	IPC	Mem.Access
Share	High	High	Low	Low	High
Shift	High	Low	High	Medium	Low
Stencil	Low	Low	Low	High	Medium
Pingpong	High	Medium	Medium	Low	Low
BT.W.16	Medium	Medium	High	Medium	Medium
LU.W.16	Low	Medium	High	Medium	Medium

Benchmark Categorization (normalized to 100M inst)—*Numerical*					
Benchmark	L1CM	Time	Msgs	IPC	Mem.Access
Share	372361	3.3622	871	0.0558	0.05
Shift	307524	0.7784	147904	0.2410	0.001
Stencil	97715	0.5528	23283	0.3393	0.03
Pingpong	280112	2.1116	68407	0.0888	0.001
BT.W.16	251096	1.11	229411	0.1682	0.03
LU.W.16	94880	1.15	305988	0.1631	0.03

The applications were run 5 times and the collected data have been averaged. An additional warmup run was conducted before the experimental runs. All of the experiments were conducted with the tiles at 533 MHz, the mesh at 800MHz and the DDR's at 800MHz. Our recent work also investigates the impact of frequency scaling on the SCC power and performance [2].

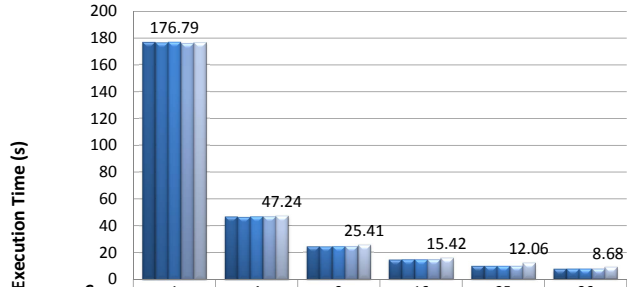
IV. EXPERIMENTAL EVALUATION

The purpose of the experiments is to quantify the performance and power of the Intel SCC system while running applications that differ in number of messages, message traffic patterns, core IPC, and memory access patterns. In this way, we hope to understand the performance-energy tradeoffs imposed by using MPI on a large manycore chip.

A. Overhead of Message Logging

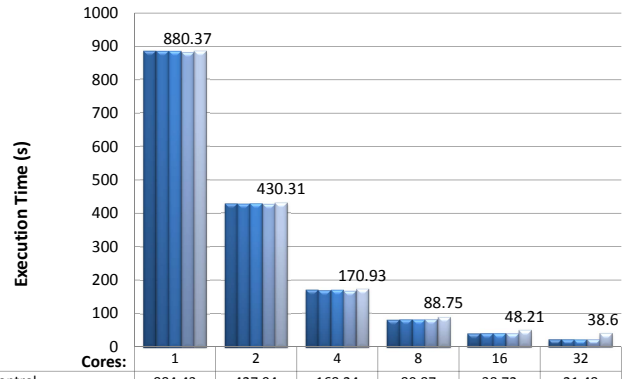
We first analyze the overhead caused by our message logging and performance monitoring infrastructure. Figures 2 and 3 demonstrate the overhead measured in execution time caused by different levels of measurement while running *BT* and *LU*. We choose *BT* and *LU* to study message over logging overhead as they are standard multicore MPI benchmarks. In the figures, *control* represents the case without any logging, *performance counters* results are for tracking performance counters only, *counting messages* is for logging both counters and number of messages, *message target* also logs the sender/receiver cores for each message, and *message size* logs the size of each message on top of all the other information.

We see in figures 2 and 3 respectively that while there is an overhead associated with the message logging, it is very small.



Cores:	1	4	9	16	25	36
Control	176.76	46.67	24.32	14.14	9.81	7.37
Performance Counters	176.39	46.36	24.23	14.24	9.81	7.45
Counting Messages	176.79	46.79	24.39	14.28	9.8	7.43
Message Target	175.82	46.8	24.43	14.24	9.77	7.47
Message Size	176.45	47.24	25.41	15.42	12.06	8.68

Fig. 2. *BT* Class W Execution Time(s) vs. # of Cores vs. level of logging. The execution time is shown for a varying number of cores. In each case the the addition of logging shows very small overhead.



Cores:	1	2	4	8	16	32
Control	884.43	427.94	168.24	80.87	38.72	21.48
Performance Counters	883.98	427.32	166.81	81.85	39.27	21.54
Counting Messages	884.04	428.18	167.92	81.51	39.37	21.3
Message Target	880.37	426.68	164.96	81.99	39.12	21.55
Message Size	884.42	430.31	170.93	88.75	48.21	38.6

Fig. 3. *LU* Class W Execution Time(s) vs. # of Cores vs. level of logging. The execution time is shown for a varying number of cores. Again, the overhead of message logging is very low.

We have seen similar low overhead when we measured the overhead for the other benchmarks. For example, for *message target* logging, we see only a 0.21% overhead in execution time when running the *Stencil* benchmark.

When logging message size is added to the infrastructure we see a significant increase in execution time, especially when a large number of cores are active (see Figures 2-3). For the *Pingpong* and *Stencil* benchmarks we have seen an increase over 200%. For message intensive benchmarks such as *Shift*, the execution time is over 600% longer compared to the *message target* logging. These large overheads are due to the large amount of data logged when the size of the messages is considered. The message size distribution varies depending on the benchmark. Some benchmarks such as *Pingpong* are heterogeneous in their message sizes, as shown in Figure 4. Benchmarks *Stencil* and *Shift* have fixed sized messages of 64 bytes and 128 bytes, respectively.

The rest of the experiments use the *message target* logging, which logs the performance counters, number of messages, and message sender/receiver information at a low overhead.

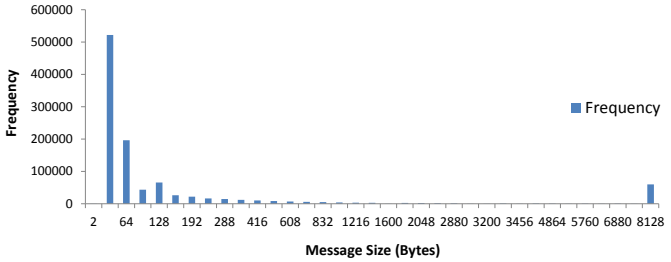


Fig. 4. *Pingpong* message size histogram. The majority of *Pingpong* messages are small; however, there are also a significant number large messages. As the broadcast benchmark is derived from *Pingpong* it has the same distribution of message sizes.

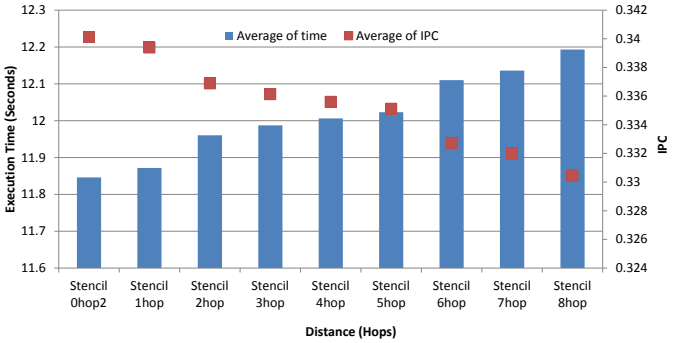


Fig. 5. *Stencil* with one pair of cores. The distance between the cores increases from local communication on the same router to the maximum distance spanning 8 routers.

B. Impact of Communication Distance

Next, we analyze how the distance between communicating cores affects performance. We look at the case of a single pair of cores that are running on the SCC. Figure 5 demonstrates that as the distance between the cores increases, the execution time increases. In the figure, we plot the execution time of *Stencil* as the distance between cores is increased from 0 hops (local) to the maximum distance of 8 hops (cores 0-47). There are clear linear trends for both the IPC and the execution time. *Stencil* is chosen in this experiment as it demonstrates the largest difference in execution time owing to its high IPC (as outlined in Table I). Similar trends can be seen for *Shift*.

C. Impact of Memory Accesses

To measure the impact of memory accesses, we keep the distance constant but increase the number of cores (i.e., number of pairs simultaneously running). In this way, we expect to increase the accesses to the main memory. In this experiment, we do not see any measurable difference in execution time for *Stencil* or *Shift*, as their memory access intensity is low. The *Share* benchmark, which has a high memory accesses density at 0.05 (see Table I), is prone to significant delays when there is memory contention. Figure 6 demonstrates this point. We see significant delay when 24 pairs of cores are concurrently executing a benchmark that is heavy in memory accesses. The combined load of 24 pairs accessing memory is saturating the memory access bandwidth and causing the delay. While this effect is due to the uncached accesses to the DRAM and specific to the SCC, the trend is observed in many multicore applications which become memory bound.

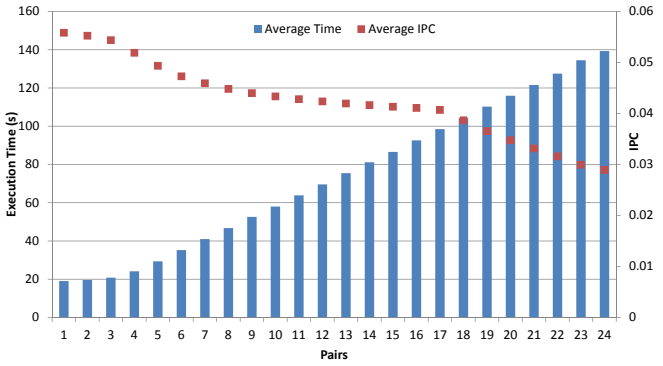


Fig. 6. Execution time of *Share* with local communication, as a function of the number of pairs executed concurrently.

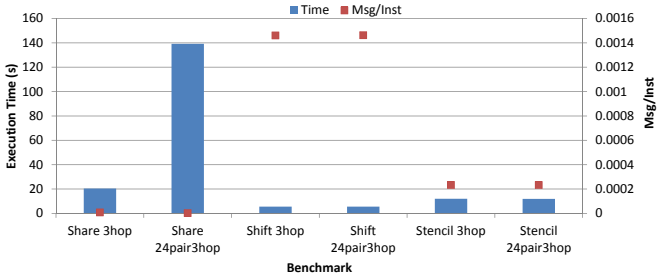


Fig. 7. *Share*, *Shift*, and *Stencil* with 3 hops communication. One pair compared to 24 pairs executed concurrently.

D. Impact of Network Contention

For exploring the impact of network contention, we compare 3-hop communication of a single pair of a benchmark versus running 24 pairs. *Shift*, which has low memory accesses but high message density, does not exhibit significant changes in execution time when comparing one pair with 24 concurrent pairs; this is visualized in Figure 7. When we look at *Stencil*, which has both memory accesses and messages, we still do not see significant differences in execution time as seen in Figure 7. In fact, the only major difference occurs for *Share*, owing to its memory access intensity. We believe these benchmarks have not been able to cause network contention; therefore, the dominant effect on the execution time is the memory access frequency in the figure.

E. Impact of Broadcast Messages

Next, we analyze performance for applications that heavily utilize broadcast messages. We run our *Bcast* benchmark for this experiment. The benchmark is an adaptation of the *Pingpong* benchmark, so as in *Pingpong*, *Bcast* sends many messages of different sizes. Instead of sending the messages to a specific core, *Bcast* sends messages to all of the receiver cores in a one to N broadcast system. Figure 8 demonstrates that as the number of cores in the broadcast increases we have significantly slower execution. It is particularly interesting that there is a peak IPC at $N = 8$ cores. This peak suggests that when $N > 8$ for the *Bcast* benchmark, the performance of the sender core and the network become bottlenecks.

Figure 9 demonstrates how as the number of cores in the broadcast increases, the messages per instruction increases

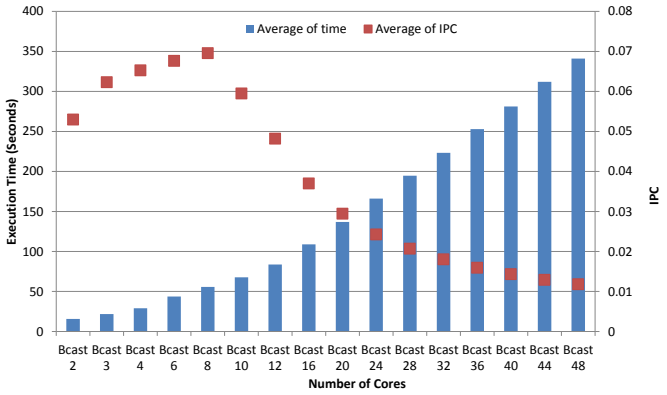


Fig. 8. Execution of Bcast with respect to number of cores. As the number of cores increase we see the growth in execution time.

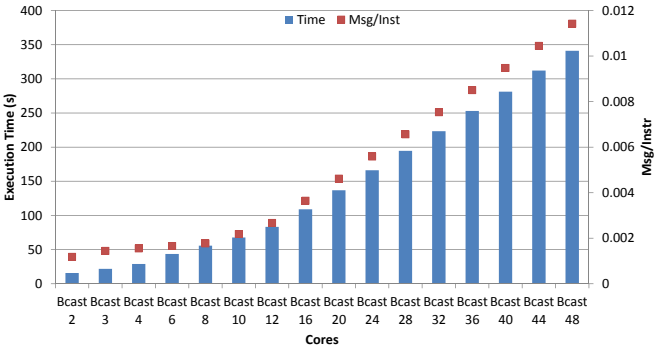


Fig. 9. Broadcast with increasing number of cores. As the number of cores increase we see a higher number of messages per instruction.

with it. Again we see that at $N = 8$ cores there is a local inflection point. This helps confirm that for this particular broadcast benchmark, broadcasting to a large number of cores saturates the traffic from the sender core, which in turn causes delays. This result highlights the importance of carefully optimizing broadcasting to ensure desirable performance levels. A potential optimization policy would be to allow for broadcasting to a small number of cores at a given time interval.

F. Power and Energy Evaluation

As part of our analysis, we also investigate the power and energy consumption for each benchmark. Figure 10 compares the *Share*, *Shift*, *Stencil* and *Pingpong* benchmarks in 24-pair 0-hop (local communication) configuration. We see that at full utilization of all 48 cores, a significant difference exists in the amount of power drawn by each benchmark. The *Share* benchmark, heavy in memory accesses and low in messages (see Table I), has relatively low power consumption compared to the *Shift* and *Stencil* benchmarks which have significantly higher IPC and power consumption. Overall, IPC is a reasonable indicator of the power consumption level.

Looking at power alone is often not sufficient to make an assessment of energy efficiency. Figure 11 compares energy-delay product (EDP) (delay normalized to 100 M instructions) for *Share*, *Shift*, *Stencil* and *Pingpong* benchmarks in 24-pair 0-hop configuration. Again, significant differences exist in the EDP across the benchmarks. The high EDP in *Share* is a result

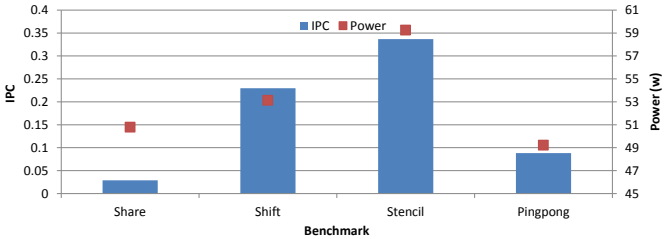


Fig. 10. Comparing IPC vs power for the *Share*, *Shift*, *Stencil* and *Pingpong* benchmarks. All benchmarks were executed with 24 pairs of cores, all with local communication.

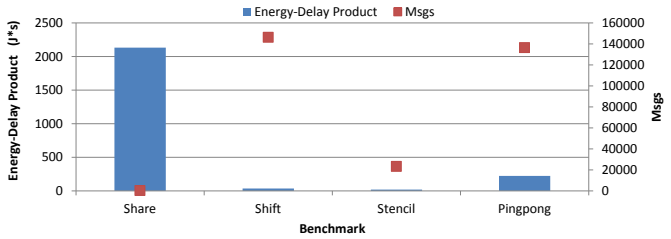


Fig. 11. Comparing EDP vs. number of messages for the *Share*, *Shift*, *Stencil* and *Pingpong* benchmarks. All benchmarks were executed with 24 pairs of cores, all with local communication.

of the high memory intensity and low IPC, which cause high delay. *Stencil* has the highest IPC, a low number of messages, and a medium level of memory accesses, which jointly explain the low EDP. *Shift* and *Pingpong* both have a considerable amount of messages. However, *Pingpong* misses a lot in the L1 cache, resulting in lower performance. Thus, its EDP is higher compared to *Shift*.

We have also compared running one pair of *Share* against 24 pairs. For one pair the power consumed is 31.616 Watts. When 24 pairs are run concurrently the power consumed jumps to 50.768 Watts. The power drawn follows linearly with the number of active cores. Due to the offset and leakage power consumption of the chip, running the system with a large number of active cores when possible is significantly more energy-efficient (up to 4X reduction in EDP per core among the benchmark set).

V. Related Work

There has been several projects relevant to the design, development, and experimental exploration of the Intel SCC. As part of Intel's Tera-scale project, the Polaris 80-core chip can be regarded as the predecessor of the SCC [8]. The main purpose of the Polaris chip was to explore manycore architectures that use on-die mesh networks for communication. However, unlike the SCC, it only supported a very small instruction set and lacked corresponding software packages that facilitate manycore application research [10].

Previous work describes low-level details of the SCC processor hardware [4]. Special focus is given to topics regarding L2 cache policies and the routing scheme of the mesh network. Other recent research on the SCC looks at benchmark performance in RCCE focusing on the effects of message sizes [7]. The authors also provide detailed performance analysis of message buffer availability in RCCE [7].

Another related area is the development of the message passing support. The RCCE API is kept small and does not implement all of the features of MPI. For example, RCCE only provides blocking (synchronous) send and receive functions, whereas the MPI standard also defines non-blocking communication functions. For this reason, some researchers have started to extend RCCE with new communication capabilities, such as the ability to pass messages asynchronously [3].

VI. Conclusion

Future manycore systems are expected to include on-chip networks instead of the shared buses in current multicore chips. MPI is one of the promising candidates to manage the inter-core communication over the network on manycore systems. This paper investigated the performance and power impact of the message traffic on the SCC. We have first described the monitoring infrastructure and the SW applications we have developed for the experimental exploration. Using our low-overhead monitoring infrastructure, we have demonstrated results on the effects of the message traffic, core performance characteristics, and memory access frequency on the system performance. We have also contrasted the benchmarks based on their power profiles and their energy delay product. Overall, the paper provides valuable tools and insights to researchers in the manycore systems research area. For future work, we plan to analyze the traffic patterns in more detail, create various local and global network contention scenarios, investigate opportunities to track other performance metrics (such as L2 cache misses), and utilize the experimental results for designing energy-efficient workload management policies.

Acknowledgments

The authors thank the Intel Many-Core Applications Research Community. John-Nicholas Furst has been funded by the Undergraduate Research Opportunities Program at Boston University.

References

[1] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, March 1994.

[2] A. Bartolini, M. Sadri, J. N. Furst, A. K. Coskun, and L. Benini. Quantifying the impact of frequency scaling on the energy efficiency of the single-chip cloud computer. In *Design, Automation, and Test in Europe (DATE)*, 2012.

[3] 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)*, pages 525 –532, July 2011.

[4] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108 –109, Feb. 2010.

[5] Intel. SCC external architecture specication (EAS). http://techresearch.intel.com/spaw2/uploads/files//SCC_EAS.pdf.

[6] M. A. Khan, C. Hankendi, A. K. Coskun, and M. C. Herbordt. Software optimization for performance, energy, and thermal distribution: Initial case studies. In *IEEE International Workshop on Thermal Modeling and Management: From Chips to Data Centers (TEMM), IGCC*, 2012.

[7] T. G. Mattson et al. The 48-core SCC processor: the programmer's view. In *High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1 –11, Nov. 2010.

[8] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1 –11, Nov. 2008.

[9] T. G. Mattson and R. F. van der Wijngaart. RCCE: a small library for many-core communication. *Intel Corporation*.

[10] S. R. Vangal et al. An 80-tile sub-100-W teraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29 –41, jan. 2008.

Ruby on SCC: Casually Programming SCC with Ruby

:

Kouhei Ueno, and Koichi Sasada

***Abstract*—Ruby is a popular lightweight programming language widely known for its high productivity. Intel single-chip cloud (SCC) is a 48-way many-core CPU with integrated message passing buffers. We explored Ruby as a tool for casually developing distributed programs on SCC. In particular, we have experimented with Distributed Ruby (DRb), a pure-Ruby implementation of the distributed objects environment. We developed a Ruby binding for RCCE, enabling Ruby to use the communication hardware facilities of SCC. We also extended RCCE to support more flexible message passing primitives as required in DRb. In this paper, we share our early experiences of using Ruby on SCC, discuss the implementation of a Ruby binding for RCCE and the optimization for DRb implementation using the RCCE binding, and present the performance evaluations of these methods from the perspective of micro-benchmarks.**

I. Introduction

INTEL single-chip cloud (SCC) is a 48-way many-core CPU platform, which Intel has made available to the research community. The SCC chip consists of a mesh of 24 tiles with 2 processors, a router, and a shared communication buffer on each tile. Each processor is assigned an off-chip private memory that is analogous to the main memory of a conventional PC, and an on-chip communication buffer (shared between 2 processors in a tile) for message passing across cores. A Linux port is available for SCC and can be configured such that each processor runs an instance of the Linux operating system. Thus, SCC can be considered a 48-node distributed memory cluster on a chip.

Distributed programming is difficult. In particular, it is a tough task to write distributed programs that use the SCC's hardware facility. RCCE[1], [2] is a "C" library for utilizing specialized hardware for inter-core communication on SCC. RCCE provides an MPI-like API for messaging passing across cores for SCC, but it still requires a programmer for system programming such as memory management.

There have been efforts to reduce the programmer's effort in distributed programming on SCC. Welc et al.[3] successfully offloaded heavy JavaScript workloads from web browser clients to SCC. X10 is a high-productivity distributed programming language based on the PGAS model. Chapman et al.[4] experimented running X10 on SCC and reported the performance gain by porting RCCE.

Our focus is to use the Ruby programming language[5] to casually write distributed programs for SCC. The design focus of Ruby is the programmer's productivity. Ruby programs are short and concise. Ruby enables users to write code with little effort. If the programmer does not find Ruby grammar satisfactory, domain specific languages (DSLs) within Ruby are supported by meta-programming features. As a dynamic language, Ruby features object-oriented programming with reflective features. Classes can be dynamically created while running the code, and inexistent method calls can be dynamically handled by a user.

A distributed object is a well-known distributed programming primitive. A server exposes access to its objects to a remote node, and a client can send messages (invoke its methods) using the same syntax as that used for local objects. The implementation of the distributed objects environment transparently forwards the messages sent to the local proxy object for remote node communication.

Distributed Ruby (DRb)[6] is a pure-Ruby implementation of the distributed objects environment. It fully uses Ruby's dynamic nature to transparently provide distributed objects without painful interface declarations. A DRb (proxy) object behaves just like a regular Ruby object, automatically translating its method calls to remote method invocation.

In this paper, we share our early experiences with Ruby on SCC. First, we built a Ruby interpreter executable for running on SCC. Next, we developed a Ruby binding for RCCE to access SCC's message passing buffers (MPBs) from Ruby programs. In this process, we extended RCCE to support the communication primitives required for DRb. Last, using the Ruby binding of RCCE, we added the optimization code to DRb, a distributed object implementation for Ruby, to support communication via RCCE. In the following sections, we will discuss the above method and its performance evaluation from the perspective of micro-benchmarks in detail.

II. Ruby on SCC

To run Ruby programs on SCC, we have compiled a Ruby interpreter binary for running on SCC. For our platform, we used the SCC Linux binary available in sccKit 1.3.0.

We chose the original Ruby interpreter in C, also known as CRuby, for the Ruby language implementation. Although other Ruby language implementations are available, CRuby was selected as it has high portability and requires only an ANSI C compiler, a POSIX-compliant OS, and support for C extensions that allow the Ruby library to be written in C.

We used the latest development branch of CRuby, revision 32047. We could build this version of Ruby using Intel C Compiler 8.1 with no source code modification. To avoid problems related to the shared library, we built a statically linked single binary Ruby executable.

We verified that the executable could run Ruby programs correctly. UDP and TCP/IP communication using the standard

K. Ueno and K.Sasada is with the Graduate School of Information Science and Technology, The University of Tokyo.

"socket" module was functioning. Unmodified DRb functioned via the TCP/IP transport.

III. Ruby binding of RCCE library

We developed a Ruby language binding of RCCE[1] to take advantage of SCC's inter-core communication hardware.

SCC has a unique hardware support for message passing. SCC consists of 24 mesh tiles, and each of these tiles contains two x86 processors, a router, and MPB. MPB is implemented as a 16 KB SRAM in each tile, and its data can be accessed from other tiles. Note that MPB is faster than private memory, which is located off-chip and accessed via memory controllers located at the edge of the mesh. For the synchronization primitive, a special register called the test-and-set register, which supports a well-known test-and-set operation, is available per processor core.

RCCE[1], [2] is a lightweight message passing library that uses the MPB attached to each tile. RCCE has two interfaces, namely, the "basic" interface and the "gory" interface; we kept within the "basic" interface for code simplicity. When the "basic" interface was used, RCCE assigned an 8 KB region from MPB to each processor core. The 8 KB region was used for storing the message body and control flags. Message transfer in RCCE is processed by using MPB. To send data to a foreign core, RCCE first copies the message to the local MPB. Then, it updates the "message is sent" flag on the target core's MPB. The receiver core waiting for data notices that the flag is ready, and copies the message body from the remote sender's MPB to the receiver's local private memory. After the transfer is completed, the receiver core notifies the sender core by setting the "next message ready" flag on the sender's MPB.

To use this MPB-based message transfer from Ruby, we developed a Ruby binding of RCCE. The Ruby binding was created as a C extension of the CRuby interpreter; it provides access to the basic RCCE functions `RCCE_init`, `RCCE_finalize`, `RCCE_ue`, `RCCE_barrier_world`, `RCCE_send`, and `RCCE_recv` from Ruby programs.

Although the original RCCE interface specifies a message by passing a pointer to the message body as an argument, Ruby by itself does not provide a way to handle raw pointers. Our binding works around the problem by specifying string values as the message body instead. The String value in Ruby can hold an arbitrary byte sequence and is commonly used as a general buffer.

IV. Adding Message Polling Support to RCCE

As our goal is to create a distributed objects environment over SCC, we needed a flexible message communication system, which is out of the scope of the original RCCE library.

The original RCCE implementation assumes the program to be in the single-program multiple-data (SPMD) style, which is very similar to MPI. However, the implementation of a distributed objects environment needs relatively flexible message communication primitives: **unmatched send-and-receive** and **variable-sized messages**.

```
int RCCE_peek(char* pbuf, size_t sz, int src)
{
  /* return if data is not ready */
  if(! RCCE_probe(RCCE_sent_flag[src]))
    return 1;

  /* copy the content of remote MPB
     to local private memory */
  RCCE_get(
    (t_vcharp)pbuf, RCCE_buff_ptr, sz, src);

  return 0;
}
```

Fig. 1. `RCCE_peek` source code

First, we needed to support **unmatched send-and-receive**. RCCE assumes that `RCCE_send` and `RCCE_recv` are written in pairs. `RCCE_recv`, the message receive function in RCCE, requires the sender core UE, a unique id given to every participating core, which corresponds to " rank " in MPI.

However, in the distributed objects environment, programs do not have prior knowledge of where and when the message is going to be sent or received. Nodes in the distributed objects system are not required to have similar roles. A node may only be responsible for a certain type of task, such as providing a database or giving an access to an external device. It is impossible to know the message flow beforehand. Every node may send data to another random node at any time. This random communication cannot be handled by the existing RCCE communication primitives, `RCCE_recv` and `RCCE_send`.

Next, we needed to support **variable-sized messages**. Method invocations to remote objects are translated to remote node communication in a distributed objects environment. As all methods have different arguments and return values, it is impossible to know the size of the messages beforehand. RCCE requires the receiver to know the size of the incoming message. The RCCE recv function takes the message size as an argument.

To resolve the above issues with minimum modification, we have added a function called `RCCE_peek`, named after `MSG_PEEK` in the UNIX socket API. `RCCE_peek` checks the message arrival from the specified core and retrieves the header part without affecting a subsequent call to `RCCE_recv`. It is defined as shown in Figure 1. The RCCE peek function first checks the local flag to see if the remote core has a message for a particular node. Then, if the flag indicates that the remote core is attempting to send a message, it peeks at the message body from the remote MPB and copies the first N bytes. The `RCCE_peek` function does not set the "next message ready" flag on the remote core MPB. Therefore, the same message can be read again with a subsequent call to `RCCE_recv`.

The `RCCE_peek` function can be used for implementing unmatched send-and-receive and variable-sized messages. We set up a listener thread that periodically peeked into messages from all possible sender cores. All messages were sent with a header containing the message body size. The `RCCE_peek` function is used for checking whether an incoming message

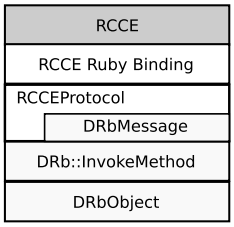


Fig. 2. Architecture overview of DRb on SCC. Light-gray parts are adopted from the original DRb implementation. White parts are our implementation.

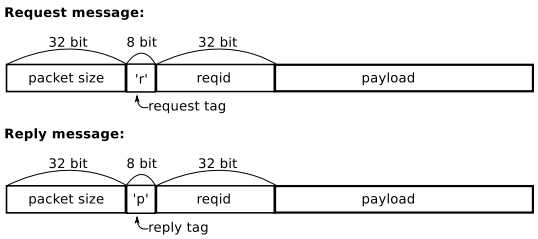


Fig. 3. The request/reply message header in `RCCEProtocol`

exists and read the message header containing the message body size. If an incoming message exists, `RCCE_recv` is called with the message body size specified in the message header.

V. DRb on SCC

Distributed Ruby (DRb)[6] is a distributed objects environment implementation for Ruby. By using Ruby's reflective features, DRb provides distributed objects without any interface declarations.

We extended DRb to support transport over SCC communication hardware. DRb on SCC is achieved by replacing DRb's inter-node communication implementation (Figure 2). By default, DRb uses its TCP-based protocol for remote method invocation and will not use SCC-optimized transports. As DRb's inter-node communication implementation is abstracted, a user can implement an original protocol for remote method invocation by implementing the `DRbProtocol` interface. We developed module called `RCCEProtocol`, an RCCE-based implementation of `DRbProtocol` to support message transfer using MPB.

`DRbProtocol` requires the methods `recv_request`/`send_request` and `recv_reply`/`send_reply` for inter-node messaging, but it assumes a stream-oriented transport, which is `open`-ed and `accept`-ed beforehand. With a stream-oriented transport, a reply is expected to come from the stream in which the request is sent. However on RCCE, a message is not associated with any stream; hence, a received reply message cannot be associated with the sent request by itself.

To emulate this stream-oriented transport on RCCE, every request is given a unique id, a `reqid`. Each message has a header as shown on Figure 3. The reply message always contains the `reqid` to match the request that the reply is for. In the RCCE protocol implementation, all messages are received in a dedicated receive thread. The `recv_reply` is implemented as a function that blocks until the corresponding reply message is received from the server. The receive thread wakes the corresponding thread waiting for the `recv_reply`.

For method invocation serialization and deserialization, we used `DRbMessage` from the original DRb. This allows the `RCCEProtocol` to be called from `InvokeMethod`, which initiates remote method invocation.

TABLE I
BENCHMARK ENVIRONMENT

	MCPC	SCC
CPU	Intel Core i7-950 3.07GHz	x86 P54C 533MHz
Memory	DDR3 12GB	DDR3 256MB per core
Linux	2.6.32	2.6.16 (sccKit 1.3.0)
Mesh Clock	N/A	800MHz

TABLE II
CRUBY PERFORMANCE ON MCPC AND SCC.
EXECUTION TIME (IN SECONDS) IS SHOWN.

benchmark	MCPC	SCC	MCPC/SCC
ackerman	0.07	1.10	6.70%
erb	0.62	14.29	4.35%
factorial	0.12	10.04	1.16%
fib	0.85	10.79	7.88%
mandelbrot	0.21	5.69	3.76%
pentomino	21.01	409.33	5.13%
raise	0.66	21.11	3.15%
strconcat	0.70	11.57	6.08%
tak	1.24	15.11	8.21%
tarai	0.98	12.19	8.00%
uri	1.05	25.73	4.06%

VI. Performance Evalutation

First, we have measured CRuby interpreter performance on SCC using benchmark programs included in CRuby distribution (Table II). We chose benchmarks which depend on performance of interpreter itself and not involving network or filesystem IO. The benchmarks *ackerman, factorial, fib, tak, tarai, mandelblot* compute the corresponding functions. The benchmarks *erb, strconcat, uri* measures time for string manipulation. The *raise* benchmark measures exception handling overhead. We run these benchmarks on SCC Linux using one of its cores, and on host MCPC. The system configuration details are shown on Table I. The same interpreter binary was used in the measurement.

The results show that performance of CRuby on SCC is around 5% that of modern PC. The memory access pattern of the program affects its performance on SCC. The benchmarks which access private memory intensively are slower. For example, *fib* benchmark creates many temporary `Bignum`[1] object for storing its result.

Next, we measured inter-core communication latency for Ruby on SCC (Figure 4). We measured time of ping-pong latency of TCP and UDP/IP using Ruby's `TCPSocket` and `UDPSocket`. The `rckmb` network interface[2] was configured to use off-die shared memory instead of on-die SRAM. The RCCE ping-pong latency was measured using the RCCE Ruby binding we have developed. The latency was calculated by taking fastest time of 3 trials, each consisting of 100 ping-pong communications.

[1]Bignums hold large integers over 31-bit

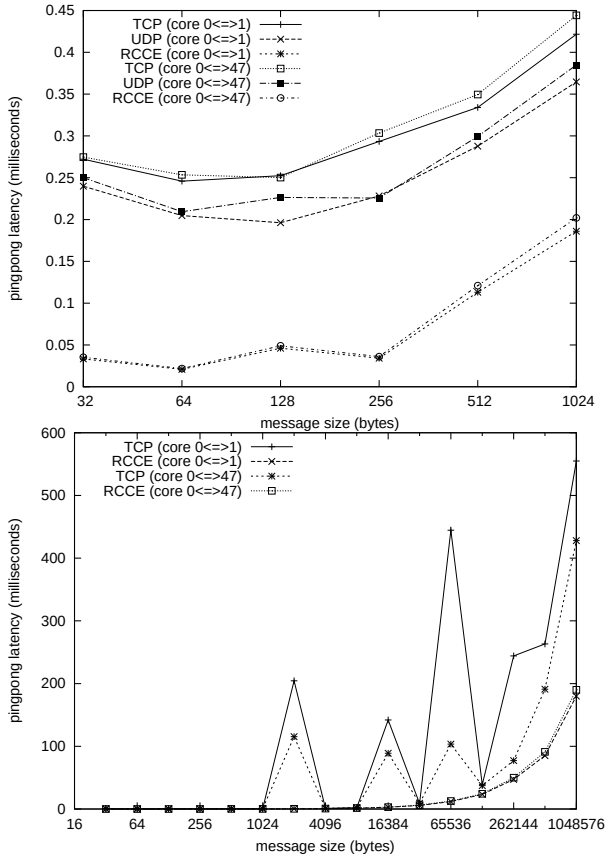


Fig. 4. Pingpong latency of TCP/UDP/RCCE communication on Ruby on SCC

TABLE III
REMOTE METHOD INVOCATION OVERHEAD OF DRB OVER TCP AND RCCE (IN MILLISECONDS)

Argument size (in bytes)	TCP	RCCE
0	3.52	3.44
64	3.65	3.62
256	4.02	4.33

The results show that communication over RCCE Ruby binding is 2 times faster than communication over TCP or UDP/IP. RCCE transfers small message effectively. Messages below 256 bytes can be transferred 4 times faster than TCP or UDP/IP. Sending large messages in RCCE are 2 times faster than TCP/UDP. Large messages are sent by splitting into small messages in both cases, but RCCE allows larger chunk to be sent at once. RCCE can send message by 8160 byte chunks, but `rckmb` is limited by 1500 byte MTU.

Then, we measured remove method invocation overhead of DRb over TCP and RCCE (Table III). We measured time calling a remote method 100 times from core 0 to core 1. The remote method does nothing but return value 0. We also measured on core 0 to core 47, but could not find difference in result.

The results show that using RCCE speed up method calls with argument size under 64 bytes speed up slightly by 2%, but slows down on method calls with argument size 256 bytes. Comparing these values with latency measurements (Figure 4), the overhead of DRb library is 10-20 times higher than raw communication cost. The speed down on large arguments come from overhead of RCCE protocol implementation added to DRb, which performs more `String` operation on the received message headers as described in Section V.

VII. CONCLUSION

We developed a Ruby-based distributed objects environment for Intel single-chip cloud (SCC). A user could easily write programs for SCC by using the highly productive Ruby language and take advantage of the SCC communication hardware transparently in a distributed objects manipulation. We built a CRuby executable running on SCC Linux to run DRb, a distributed objects environment implementation in pure-Ruby. We developed the Ruby binding for RCCE to use SCC's inter-core messaging hardware. We extended RCCE to support communication primitives required for DRb and implemented an SCC-optimized DRb transport layer. The performance evaluation using micro-benchmarks show that the raw communication improved by 200% the remote method invocation via DRb improved by 2%.

As our future work, we plan to develop an optimize distributed objects implementation with lower overhead. Evaluation showed that the overhead of DRb library is 10-20 times higher than raw communication cost. We found that this is from high abstractions in the DRb implementation to support customizations. We expect that lower overhead can be achieved by creating minimal implementation that only support RCCE communication.

Also, we plan to explore various Ruby-based distributed applications on SCC. Examples are web applications and natural language processing. There are various frameworks for creating web applications on Ruby, such as Ruby on Rails[7]. The web common interface Rack[8] can be accelerated using SSC hardware in a similar way as by using DRb. Natural language processing is a region where implementation in a low-level language such as C is difficult. Ruby has a rich `String` class for manipulating text strings and has built-in support and external frameworks for tools such as regular expressions.

We expect that our alternative development environment using Ruby will open up various unexplored uses of SCC.

REFERENCES

[1] R. van der Wijngaart and T. Mattson, "RCCE: A small library for many-core communication." [Online]. Available: http://techresearch.intel.com/spaw2/uploads/files/RCCE_Specification.pdf
[2] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.
[3] A. Welc and R. L. Hudson, "Javascript farm on scc." [Online]. Available: http://communities.intel.com/docs/DOC-5646
[4] A. H. Keith Chapman, Ahmed Hussein, "X10 on the SCC." [Online]. Available: http://communities.intel.com/docs/DOC-6255
[5] Yukihiro Matsumoto, et al., "Ruby Programming Language." [Online]. Available: http://www.ruby-lang.org
[6] M. Seki, "dRuby." [Online]. Available: http://www.druby.org/ilikeruby/druby.en.html
[7] Rails Core Team, "Ruby on Rails." [Online]. Available: http://rubyonrails.org
[8] C. Neukirchen, "Rack: a ruby webserver interface." [Online]. Available: http://rack.rubyforge.org/

Characterization and analysis of pipelined applications on the Intel SCC

:

Tommaso Cucinotta,Vivek Subramanian
Real-Time Systems(ReTiS) Lab
Scuola Superiore Sant'Anna
Pisa, Italy.
cucinotta@sssup.it, vivek@retis.sssup.it

***Abstract*—Many-core computing platforms can be used to parallelize computations by dividing the data to be processed into smaller chunks and processing them simultaneously on different cores. One possible approach in such parallelization is to set up a pipeline such that each smaller chunk of data passes in turn through all the processors involved. In this paper we examine some approaches to set up such a pipeline on the Intel SCC. We use a combination of the message passing and the shared memory capability of the SCC hardware through the interfaces provided by the RCCE library for our implementation. We build a model to analyze and compare the performance of such pipelines by measuring the total time for computation. This model is used to illustrate the effects of type of memory scheme used, ordering of cores in the pipeline and caching.**

***Index Terms*—pipeline, real-time streaming, message passing, shared memory, SCC**

I. Introduction

THE parallelization of a computing task is a well-studied problem. There are several approaches and methods to achieve parallelization depending on the nature of the computing task. One of them is the pipeline approach that may be applicable in situations where input data may be divided into smaller fragments, each of which must have a certain set of operations carried out in a specified order. This is particularly effective whenever the data to be processed is available progressively, for example, in multimedia streaming applications, where pipelining the application results in a higher sustainable throughput.

In several instances the current operation to be carried out on a certain fragment is not dependent on either the result of the previous or the subsequent fragment. Also, the computing task maybe divided to be carried out at different processing elements. As an example; an audio-processing application that processes an input audio file for streaming, or DES encoding of an input file. The operations carried out at each processing element could be different (e.g the audio-processing application, where a different filter is used each time) or identical (e.g the cryptographic application). Setting up a pipeline for processing such tasks serves to reduce the amount of computing that a single processing element needs to perform before switching to the next data chunk, thus increasing the possible throughput, or reducing the computing requirements on a single processing element.

There are several factors that affect the performance of a pipeline such as the latency in memory accesses and the overheads involved in moving data between processing elements. If the processing elements communicate over an interconnect network, then simultaneous use of the network, by more than one processing elements results in contention for bandwidth which affects the performance of the pipeline.

In this work, we introduce a set of variables and equations to describe the pipeline. We use experimental implementations to help validate these models of the pipeline. The aim of these experiments is to have a method to build models of the various building blocks of the pipeline and of the pipeline itself. These may then be used to gauge the expected performance of pipeline and this, in turn, be used to guide the process of actual implementation and deployment that leads to gains in throughput and processing times.

II. Related Work

The availability of the fast message passing buffers on the SCC allow for using these for inter-core communications. The RCCE library [2] provides a framework to implement message-passing on the SCC, however a number of authors addressed the problem of efficient inter-core communications on the SCC. For example, Rotta [8] presents design options for message passing protocols and discusses them. Villa et al. [11] study the efficiency and scalability of barrier synchronization in NoC-based many-core systems. The NoC-based architecture of the SCC that uses the mesh-network to access the off-chip RAM presents challenges introduced by this additional latency. Verstraaten et al. [4] presents methods to implement memory copy mechanisms aimed at increasing the throughput. Abts et al. [1] explores issues in placement of memory-controllers and the effect on latency. Petrot et al. [6] present a software-based solution for cache coherency and memory consistency in NoC-based multiprocessors. Prell and Rauber [7] address methods for achieving task parallelism on the Intel SCC using runtime task schedulers. Kierstscher et al. [10] present the effects of MPI applications having knowledge of the topology, while Tol et al. [3] discuss the mapping of a distributed implementation of the S-Net on the SCC. Bo et al. [9] discusses the optimization of data-parallel operations in the context of many-core platforms. Papagiannis and Nikoloppoulos [5] examines bottlenecks in scalability of the MapReduce algorithm and presents an implementation of the same for the SCC.

III. Preliminaries

A. Modeling memory access

Consider the Intel SCC which uses a tile-based architecture with a mesh NoC that connects the tiles and the memory controllers. Each tile has two cores, their caches and a small local memory (the local memory buffer or the message passing buffer).Each core on the SCC is assigned space in the message passing buffer (MPB). The RCCE library uses this buffer to implement message passing between the cores.

Let $mpb(i, b)$ denote the time taken by a core i to write b bytes of data into its own MPB. Let $coord(i)$ be the coordinates of the tile that contains core i, such that $coord(i).x$ and $coord(i).y$ indicate, respectively, the x-coordinate and the y-coordinate. Let $dist(i, j)$ denote the routing distance between elements i and j. Note that, the elements may be either cores or memory-controllers. As the SCC uses dimension-ordered routing, we may write $dist(i, j)$as:

$$\begin{aligned} dist(i,j) &= |coord(i).x - coord(j).x| + \\ &\quad |coord(i).y - coord(j).y| \end{aligned} \tag{1}$$

If the data rate of the links of the NoC are denoted by μ, then the time taken to transfer b bytes from i to j can be expressed as t_t:

$$t_t(i, j, b) = \frac{dist(i, j) \cdot b}{\mu} \tag{2}$$

The above expression assumes that only a single transfer is happening over the set of links. We assume this simplistic way to model the memory access and further assume that it would be an upper bound on the time it takes to access memory in the worst scenario in this simple case. This might not always be the situation and there may be more than one core using the same links of the NoC. In this case, the effective data rate may be lower (see VII).

Define a function $mem(i)$ similar to $coord(i)$, but instead of indicating the coordinates of i, $mem(i)$, indicates the coordinates of the memory controller that has the private memory of i. Similarly define $shmem(i)$ to indicate the coordinates of the memory controller that has the shared memory that i is using.

B. Modeling message passing

The RCCE library provides synchronous blocking *send()* and *receive()* interfaces for transferring messages between cores. The *send()* method accepts the rank of the core that is the destination and the *receive()* method accepts the rank of the core that it expects to receive a message from. These calls have to be matched - for every send executed to j from i, j must execute a matching receive from i.

RCCE implements this mechanism such that the sending core writes the message from its private memory to the MPB, and signals the destination core. The destination core reads the message from the source's MPB (via the lookup table entries) and stores into its own private memory. Thus, a send and receive operation consists of one off-chip memory read by the source, one write by the source to its own MPB, one read of the source's MPB by the destination and one write to the off-chip memory by the destination. For a message of size b bytes, we may write this time taken as t_m:

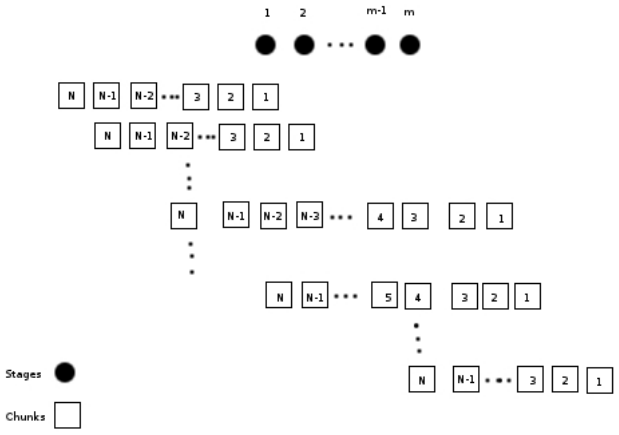


Figure 1. Representation of the pipeline

$$\begin{aligned} t_m(i,j,b) &= t_t(i, mem(i), b) + \\ &\quad mpb(i, b) + \\ &\quad t_t(i, j, b) + \\ &\quad t_t(j, mem(j), b) \end{aligned} \tag{3}$$

IV. Pipeline

The pipeline that we consider has several stages through which each chunk of data must be processed. For the purpose of this study, we have kept the operation performed at each stage to be identical. Also, a single core is mapped to exactly one stage in the pipeline. Each stage in the pipeline does the following:

- receive a single chunk of data from the previous stage
- perform the operation on that chunk
- send the chunk to the next stage in the pipeline

Since we use the RCCE library for message passing and synchronization, the send and receive steps are synchronous. Thus, all the cores are almost simultaneously doing one of the three steps described above. The first and the last stages of the pipeline are slightly different from the other intermediate stages - the first stage instead of receiving a chunk, reads a chunk from the input buffer and, the last stage instead of sending a chunk forward, writes to an output buffer. Figure 1 shows a representation of the pipeline.

The pipeline has a set of parameters associated with it:

- D is the total size of data (in bytes) to be processed by the pipeline.
- C is the size of each chunk (in bytes)
- $N = D/C$ is the number of chunks
- m is the number of stages in the pipeline
- Z is the size of a token
- $t_c(i, b)$ is the time take to compute b bytes at stage i - each compute step is a read from memory (private or shared), process and write to memory(private or shared).

- $T_{pipeline}(D, C, m)$denotes the time taken by a pipeline with m stages to process D bytes of input in chunks of C bytes.

We have implemented the pipeline using two of the memory types available on the Intel SCC. The following subsections describe each of these approaches.

A. Private memory

The private memory of a core is visible and accessible only to that core. In the implementation of the pipeline using private memory all the buffers that each stage uses are allocated in the private memory using the standard *malloc()*. The chunks are sent from one stage to the next using the *send()* and *receive()* methods of the RCCE library. Denote the time taken to process the nth chunk of size C bytes at stage i by $t_p(i, C)$, as:

$$\begin{aligned} t_p(i,C) &= t_m(i-1,i,C) + \\ &\quad t_c(i,C) + \\ &\quad t_m(i,i+1,C) \end{aligned} \tag{4}$$

For the pipeline to proceed, one chunk at a time must be processed and placed in the output buffer. For this to happen, $m-1$ messages have to be sent (or received) and since the messaging is synchronous we need to consider only either the time for sending or receiving - the time spent sending in at stage $i-1$ will be equal to the time spent in receiving at i. When the chunk from the last stage is placed in the output buffer, exactly one more new chunk may be admitted at the first stage. Thus, the maximum time spent at each step of the pipeline in processing is the time taken by the stage that has the maximum $t_c(i, C)$. That is, if the pipeline were to be stalled - some stage is in the processing step, while the stages before this stalled stage are waiting to send and the ones after the stalled stage are waiting to receive - then the pipeline would progress only when the stalled stage finishes processing and sends the chunk on to the next stage. In the time that this stage took to process the current chunk, all the other stages would have processed exactly only one chunk. Hence, we may write the time taken for pipeline to complete as:

$$\begin{aligned} T_{pipeline}(D,C,m) &= N \cdot \left[\sum_{i=1}^{m-1} t_m(i,i+1,C) \right] + \\ &\quad (N+m) \cdot max\left\{ t_c(i,C) \right\}_{i=1}^{m} \end{aligned} \tag{5}$$

$$\begin{aligned} &\leq N \cdot (m-1) \cdot max\left\{ t_m(i,i+1,C) \right\}_{i=1}^{m-1} + \\ &\quad (N+m) \cdot max\left\{ t_c(i,C) \right\}_{i=1}^{m} \end{aligned} \tag{6}$$

B. Shared memory

The SCC provides a shared memory area that can be accessed by all cores on the platform. We use this shared memory as one of the ways to implement the pipeline. In this case,the first core allocates the space in shared memory for the buffers and sends the offset from the start of the shared memory region to the other cores. One of the buffers that is created in the shared memory is a queue. Access to each slot in the queue is managed using tokens (denoted a Z) which the first stage initially generates. A stage may only access the slot to which it holds the token. The first stage reads data in chunks from the input buffer into the slots of the queue and the last stage reads out data from the queue into the output buffer. Unlike in the previous method where the entire chunk was transferred over the NoC, here only the token is transferred from stage to stage. Due to the synchronous nature of communication, the number of slots in the queue has an upper bound equal to $m-1$.

$$\begin{aligned} t_p(i,C) &= t_m(i-1,i,Z) + \\ &\quad t_c(i,C) + \\ &\quad t_m(i,i+1,Z) \end{aligned} \tag{7}$$

As reasoned for the private memory case, in this implementation as well a similar reasoning can be applied. The differences are that since the queue is of a circular nature (due to a ring created by the passing of tokens among the stages), the number of messages at each step of the pipeline is m (the last stage sends the token to the first stage). Also, the stages move along the queue of chunks as opposed to the chunks moving from one stage to another.

$$\begin{aligned} T_{pipeline}(D,C,m) &= N \cdot \left[\sum_{i=1}^{m-1} t_m(i,i+1,Z) \right] + \\ &\quad N \cdot t_m(m,1,Z) + \\ &\quad (N+m) \cdot max\left\{ t_c(i,C) \right\}_{i=1}^{m} \end{aligned} \tag{8}$$

$$\begin{aligned} &\leq N \cdot m \cdot max\left\{ t_m(i,i+1,Z)_{i=1}^{m-1}, t_m(m,1,Z) \right\} + \\ &\quad (N+m) \cdot max\left\{ t_c(i,C) \right\}_{i=1}^{m} \end{aligned} \tag{9}$$

1) Uncached and cached shared memory: Shared memory on the Intel SCC can be made uncached - bypasses both the L1 and L2 cache, or cached - cached in both L1 and L2, by setting the relevant bits appropriately for the corresponding page table entries. RCCE provides a mechanism to achieve this at the time of compiling the library. The uncached and the cached shared memory is exposed, respectively, through the */dev/rckncm* and */dev/rckdcm* devices. Depending on how the library was compiled the shared memory is mapped to one of these devices. The RCCE interfaces to handle this memory does not change significantly.

Although, using the cached shared memory has the advantage of reducing the number of memory accesses during computation, it currently comes with the overhead of having to flush the entire cache on the core to ensure consistency of the shared memory. A core must flush caches before begining to read from shared memory and must flush after modifying data in shared memory. As a result, the number of flush operations is proportional to the number of chunks that the input data is split into times the number of cores participating.

C. Effects of ordering of cores

The trival method to order cores would be in order of their physical core ID. Though on the average, the distance between tiles is about as small as it can get, the distance is quite large at certain points (for instance, from core 11 to core 12 and from core 23 to core 24). Some advantage could be gained if the ordering of cores and mapping of pipeline stages were done in such a manner so as to keep the routing distances as small as possible. One possible method is to start at some corner (say tile $(0, 0)$) and then move in the direction of increasing x-coordinate values, then step up to the next y-coordinate, and move in the direction of decreasing x-coordinate, and follow this method till the last core. The gains from following such reordering is only significant if the time spent in message passing itself is comparable to the time spent in processing, and further if the differnce in latency in messaging cores with different routing distances itself is significantly appreciable. Nevertheless, some small gains are to be expected by reducing the routing distances between stages of the pipeline.

V. Experimental Results

All experiments were performed on the SCC with 32GB of off-chip memory. The SCC system was configured with the cores running at *533Mhz*, the mesh at *800Mhz* and the DDR at *800Mhz*. The RCCE library was compiled with the non-gory interfaces and without the power-management options enabled. The -DSHMADD flag was enabled to for increasing the size of the available shared memory. The LUT entries corresponding to the shared memory were re-arranged such that the first 15 entries pointed to shared memory on the memory bank connected to tile (0,0), the next 15 to the bank connected to (5,0), the next 15 to the bank connected to (0,2) and the last 15 to the bank connected to (5,3). The processing done on each core was a placeholder operation that simply incremented the value read in the input by 1 and wrote it back. The applications on the core were run at real-time priority by using SCHED_FIFO with a priority of 20.This section presents sample results from the experiments we have performed on the described setup.

Figure 2 plots the time taken to access (read and write) 4MB of shared memory from each of the cores against the distance (in terms of dimension-ordered routing) of the core from the memory controller. We see that the latency varies almost linearly with increasing distance with respect to each of the four controllers.

Figure 3 and Figure 4 are for the private memory based implementation. We expect the time to process the input to be linearly dependent on the size from (6). The bumps in Figure 2 are at $C = 16KB$ are possibly due to interference from the L1 cache - since we would expect the previous chunk's data (and marked 'dirty') to be residing in the cache when the current chunk is being processed, thus accesses to the current chunk's data would cause evictions in the cache resulting in a write-back of the evicted data into memory.

Figure 5 and Figure 6 are using an uncached shared memory based model. As expected, there is a linear increase of total processing times but is independent of chunk sizes, since every

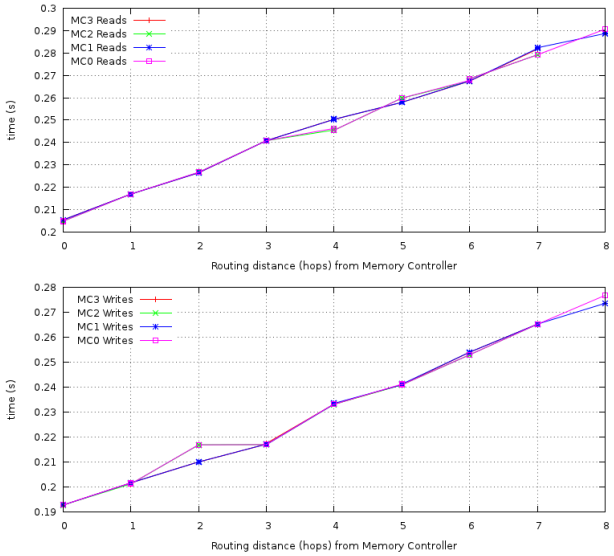


Figure 2. Read and write access times to shared memory for 4MB

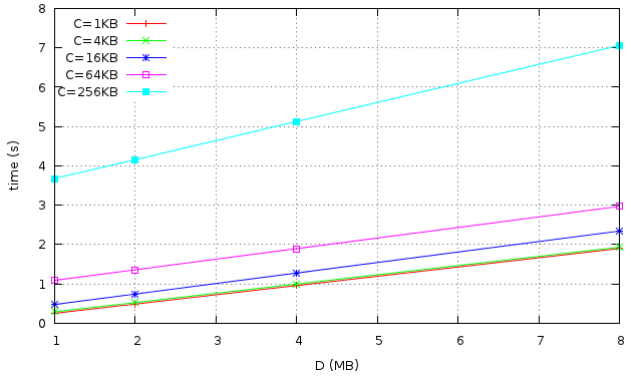


Figure 3. D vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a private-memory implementation and a trivial ordering of core by ascending physical ID

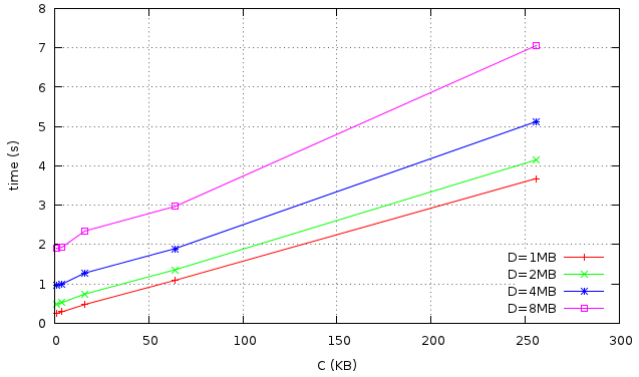


Figure 4. C vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a private-memory implementation and a trivial ordering of core by ascending physical ID

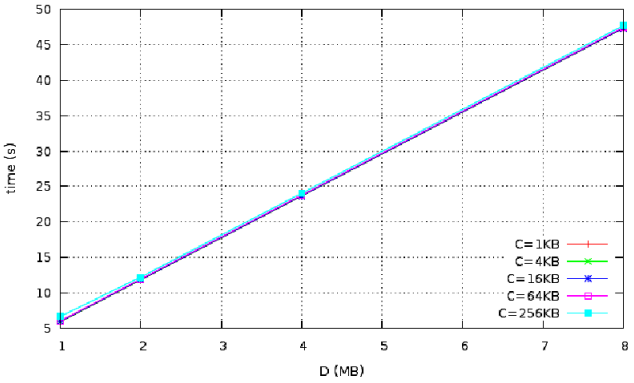


Figure 5. D vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a uncached shared memory implementation and a trivial ordering of core by ascending physical ID

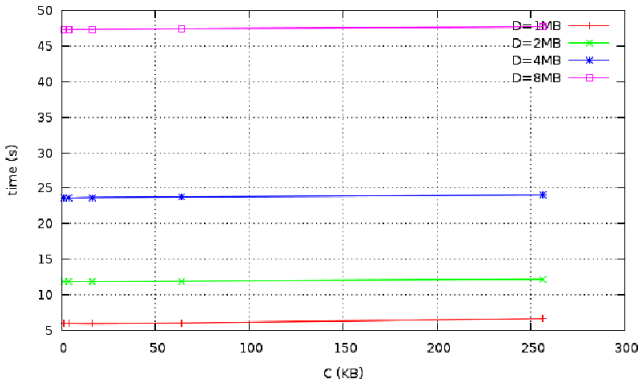


Figure 6. C vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a uncached shared memory implementation and a trivial ordering of core by ascending physical ID

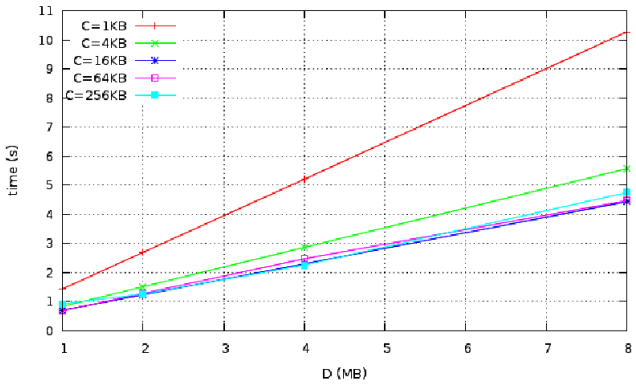


Figure 7. D vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a cached shared memory implementation and a trivial ordering of core by ascending physical ID

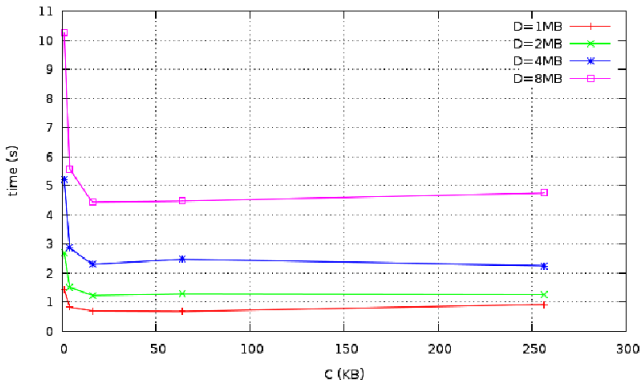


Figure 8. C vs $T_{pipeline}(D, C, m)$ with $m = 48$ for a cached shared memory implementation and a trivial ordering of core by ascending physical ID

access is from the off-chip memory. Also, in Figure 6 we see that the total time for processing doubles with doubling of input data size.

Figure 7 and Figure 8 refer to the cached shared memory based model. Here the nature of the plot is due to the additional time taken in explicity flushing caches before reading from shared memory to prevent reading stale data from the cache and after writing to shared memory to ensure changes are flushed from the cache to the memory. The flushing causes the entire cache to be flushed. For every chunk that a core processes, two flush operations are needed, hence, doubling the chunk size (hence, halving the number of chunks) halves the number of cache flushes needed.

Figure 9 and Figure 10 compare the performance of the three approches. The current limitation in the cache flushing causes the cached shared memory implementation to be somewhat worse than the private memory implementation for smaller chunk sizes, due to a large number of flushing operations. But, for larger chunk size it performs better.

We compared the times for two orderings of the cores which are:

1) A *trival* ordering in order of physical ID of cores - 0,1,2,3,4,....,45,46,47.
2) A reduced inter-core distance ordering denoted by *min-routing* - 0,1,2,...,9,10,11,23,22,21,...,14,13,12,24,25,26,...,33,34,35,47,46,45,...,38,37,36

Table I lists the comparison of total processing time for the three memory models based on the ordering of cores and with $C = 256KB$. Though there are gains, the effect is most noticeable in the case of the cached shared memory implementation.

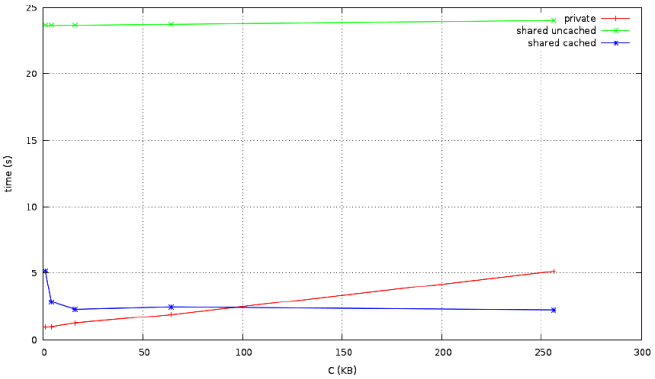


Figure 9. C vs $T_{pipeline}(D, C, m)$ with $m = 48$ and $D = 4MB$ - comparison

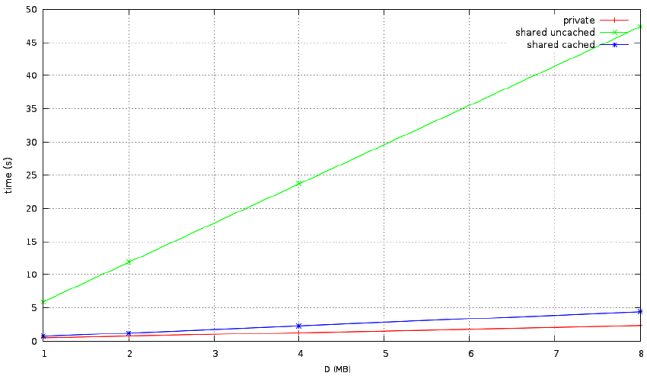


Figure 10. D vs $T_{pipeline}(D, C, m)$ with $m = 48$and $C = 16KB$ - comparison

$D(MB)$	$T_{pipeline}(D, 256KB, 48)$(sec)	
	Trivial ordering	Min-routing ordering
	Private memory	
1	3.678	3.670
2	4.155	4.151
4	5.128	5.117
8	7.060	7.054
	Shared uncached memory	
1	6.635	6.553
2	12.200	12.118
4	24.031	23.948
8	47.693	47.611
	Shared cached memory	
1	0.920	0.868
2	1.256	1.143
4	2.252	2.042
8	4.745	4.313

Table I
COMPARISON OF TOTAL PROCESSING TIMES AGAINST ORDERING AND MEMORY-MODEL

VI. CONCLUSIONS

From our experiments we concluded that in deploying pipelined applications factors of memory access latencies need to be accounted for improving the performance of computation. Further, it will help in improving performance by running data-intensive stages on cores that are closer to memory.

The ordering of the cores also shows effect on the total computation time. Ordering of the stages that such that stages that have a bulk of inter-stage communication requirements on cores that are close will lead to better computation times. In this work, in our experimental pipeline the communication pattern between stages is near-identical and quite predictable.

The simplistic assumptions we have used for memory and communication give us some estimation as to the performance of the pipeline, these need to be generalized further to improve the capabilities (See VII).

Though using messaging to transfer data between stages of the pipeline performs better than the shared memory (cached) approach, a finer method of just flushing only modified locations from the cache to the memory will greatly improve performance.

VII. FUTURE WORK

In the future we plan to investigate methods to dynamically map stages of a pipeline to cores based on constraints such as the maximum acceptable end-to-end delay or latency of the application. We will also measure how using the on-die TCP/IP communication affects the performance. Further we will explore the effect of the degree of parallelization of stages in the pipeline and methods to incorporate this factor into how stages are mapped onto cores.

In this work we assume a very simple model of the memory access, we plan to consider cases with concurrent accesses by different cores and the load this imposes on the NoC to be able to be more accurate in estimating performance. Also, the communication pattern between stages currently is near-identical and quite predictable. This is usually not the case in a general deployment and we will explore ways to model such general cases. This will enable us to create a more realistic model which will help further in the task of dynamic mapping of stages to cores.

VIII. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme under grant agreements n.248465, in the context of the S(o)OS Projects.

REFERENCES

[1] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *Proceedings of the 36th annual international symposium on Computer architecture*. ACM, 2009.

[2] Tim Mattson and Rob van der Wijngaart. *RCCE: a Small Library for Many-Core Communication*. Intel Corporation.

[3] Michiel Van Tol Roy Bakker Merijn Verstraaten, Clemens Grelck and Chris Jesshope. Mapping Distributed S-Net on the 48-core Intel SCC processor. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[4] Merijn Verstraaten Clemens Grelck Michiel W. Van Tol, Roy Bakker and Chris R. Jesshope. Efficient Memory Copy Operations on the 48-core Intel SCC Processor. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[5] Anastasios Papagiannis and Dimitrios S. Nikolopoulos. Scalable Runtime Support for Data-Intensive Applications on the Single-Chip Cloud Computer. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[6] F. Petrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in noc based shared memory multiprocessor soc architectures. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, 2006.

[7] Andreas Prell and Thomas Rauber. Task Parallelism on the SCC. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[8] Randolf Rotta. On Efficient Message Passing on the Intel SCC. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[9] Byoungro So, Anwar M. Ghuloum, and Youfeng Wu. Optimizing data parallel operations on many-core platforms. Intel Corporation.

[10] Simon Kiertscher Steffen Christgau and Bettina Schnor. The Benefit of Topology-Awareness of MPI Applications on the Intel SCC. In *3rd Many-core Applications Research Community (MARC) Symposium*, 2011.

[11] Oreste Villa, Gianluca Palermo, and Cristina Silvano. Efficiency and scalability of barrier synchronization on noc based many-core architectures. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2008.

Deterministic Execution on Many-Core Platforms: application to the SCC

:

Bruno d'Ausbourg, Marc Boyer, Eric Noulard, and Claire Pagetti

***Abstract*—The last decade has seen the emergence of multicore architectures, i.e. chips integrating several cores. The forthcoming generation of hardware components will implement the *many-core* architecture: numerous cores communicating over a *Network-on-Chip* (NoC) technology.**

The purpose of the paper is to overcome the main issues of those hardware platforms for executing safety critical embedded applications. Indeed, a many-core involves several non predictable mechanisms which make it hard to determine the worst case behaviors. We propose to use a programming model which consists of a set of execution rules that reduce the non predictable behaviors. We illustrate our solution on the Intel SCC research processor.

***Index Terms*—many-core, multi-core, real-time scheduling, deterministic execution, critical and embedded systems.**

I. Introduction

FOR cost reasons and because of the high level of integration, embedded systems manufacturers widely rely on commercial off-the-shelves hardware components (COTS). The trend of future processors is the many-core technology [1]: such chip includes numerous cores (more than 16) with distributed memories and a complex communication network based on a *Network-on-Chip* (NoC) technology. This promising emerging technology will be the next COTS hardware for time critical systems as well. The purpose of the paper is to examine the main issues and to propose adequate solutions for embedding this kind of hardware platforms for executing safety critical applications.

Many-core platform involves several non predictable mechanisms for managing the resource sharing which make it hard to ensure time predictability. Most of the works in the literature on many-core architectures are concerned with high performance: the idea is to extend and adapt current operating systems to the new architectural organisation [2], [3], [4]. The main concern of an embedded system designer is somehow different: he/she wants to determine the worst case behaviors in order to verify that the hard real-time requirements are always met, rather than to study the average performance.

For our first work, we propose to implement a simple execution model and verify that the system is analyzable, meaning that the *worst case execution time* (WCET) for any task is computable.

All authors are working at ONERA, the French Aerospace Labs, in the DTIM departement – http://www.onera.fr/dtim, E-mail addresses: firstname.lastname@onera.fr

This work has been funded by ONERA research program PR-SCC and supported by Intel through a research proposal for working with Intel SCC and the Many-core Applications Research Community (MARC).

A. Execution model

Our *system model* consists of a concurrent periodic tasks set $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$ communicating via variables and subjected to dependencies. The usage domain rules we propose are the following:

1) a unique task is statically allocated on a unique core. A global multi-threaded execution is not an alternative for hard real-time applications [5], this is the reason why we prefer a fully partitioned approach;
2) the periodicity of each task is handled by the local clock of the core. Indeed, the cost for accessing the local clock is much cheaper and conflict-less than the global clock one. Moreover, on a single chip multi-core all local clocks are derived from the same hardware clock and should therefore be running synchronously,
3) tasks only communicate via message passing. In uniprocessor or multi-core cases, the implementation of communications is usually done using global shared variables. Thus, tasks read values from local caches which should be maintained coherent with the RAM using some cache snooping feature. Such an approach implies unpredictable [5] time delay. Many-core architectures often propose message passing mechanisms between cores in order to overcome their distributed memory nature. In our deterministic quest, an explicit message is more suited because the user precisely knows the number of messages and within which time interval they occur. This leads to a reduction of the non predictability at the cost of modifications of the code in order to implement explicit communication using messages.

B. Automatic message passing programming

For helping the real-time system designer to use the execution model, we propose to derive the message passing code from a high-level specification. We should be able to do that for example with the PRELUDE language and compiler [6]. As a proof of concept we will first manually implement an example and then come up with the proper modification of the PRELUDE compiler afterwards. For the first implementation, we use point-to-point messages. So for instance, if τ_i produces a variable v which is consumed either by τ_j and τ_k, the resulting code sends two messages at each execution of τ_i: one for τ_j and one for τ_k. In fact it is a little more complicated than that because since the tasks are multi-periodic, a faster producer task will only send part of those messages to a slower consumer task.

Many-core architectures, like the SCC, are distributed memory architectures with sometimes hardware support for message passing, like the SCC MPB [7] which seems well suited for our message passing model. We will see in section III that we will use the widespread *single program multiple data (SPMD)* programming model because it can be easily used on the SCC even if our genuine need is a pure message passing model not its SPMD restriction.

C. Expected benefits: timing analysis

The final goal for using the execution model is to be able to compute the WCET of any task taking into account the memory accesses (for internal computation within a task) and the message WCTT (worst case traversal time) required by the task execution.

For our first work, we will not prove how to compute a WCET on the restricted architecture (by restricted architecture we mean a many-core compliant to the execution model defined above). We will simply provide experimental results and measures on the exchanged messages observed on the Intel SCC. Nonetheless, the execution model design should lead to easier and tighter WCET computation. In fact with our execution model the WCET computation is essentially equivalent to a classical mono-core one with the extra simplification of one task per core.

II. Target application

We target a subset of time-critical control command embedded applications such as the flight control of an aircraft, the guidance and navigation system of a vehicle. At ONERA, we use the formal language PRELUDE [8] designed for the specification of the software architecture of critical embedded control system. It belongs to the synchronous data-flow languages and focuses on dealing with the functional and real-time aspects of multi-periodic systems conjointly. PRELUDE enables the modeling of control application as the one described on Fig. 1.

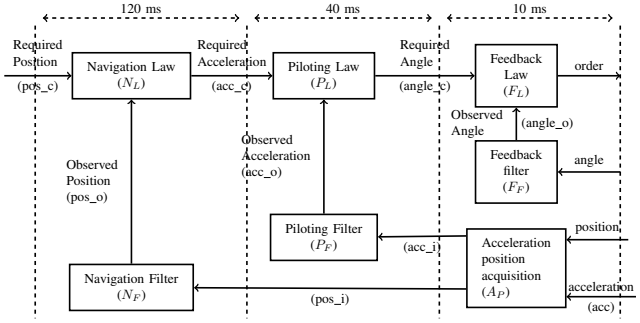


Fig. 1. Flight control system

From a PRELUDE program the compiler generates a periodic dependent task set $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$ exchanging via communication patterns $\mathcal{C}$ and subject to dependencies $\mathcal{R}$:

- Each task τ_i is characterized by the classical real-time parameters (T_i, C_i, O_i, D_i): T_i is the period, C_i the [worst case] execution time [estimation], O_i the release date or offset and D_i the relative deadline.
- The communication patterns $\mathcal{C}$ define precisely which values of an exchanged variable are consumed. For instance, there is communication $F_F \rightarrow F_L$ and both tasks execute at the same rate. In that case, the communication pattern is simply that F_L consumes each data produced by F_F. This is expressed in Fig. 2 by the fact that F_F writes every value of *angle_o* as (1): it is a periodic word that states for *write.write.*... In the same way, F_L reads every value which is expressed by (1). The communication $A_P \rightarrow P_F$ is more complex since P_F executes less often than A_P. In that case, P_F consumes every four produced values. This is expressed in Fig. 2 by the words (1000) for A_p which means *write.no.no.no.write.no.no.no.*... and (1) for P_F which means *read.read.*... PRELUDE involves numerous operators for expressing highly complex communication patterns.
- The dependencies $\mathcal{R}$ are the temporal translation of the communication patterns between the different tasks. Indeed, the task producing a data must be complete before the consumer starts. So for instance, the communication $F_F \rightarrow F_L$ imposes that F_F must always end before F_L. The communication $A_P \rightarrow P_F$ imposes that four jobs of A_P must always end before a new P_F jobs is started. This is expressed in Fig. 2 by the notation $A_P \xrightarrow{(0,0)} P_F$.

Example 1: A PRELUDE program for the example given fig. 1 may generate the task set given Fig. 2.

Task	T	C	O	D
N_L	120	20	0	120
N_F	120	10	0	120
P_L	40	5	0	40
P_F	40	5	0	40
F_L	10	2	0	10
F_F	10	1	0	10
A_P	10	1	0	10

$\mathcal{R}$
$F_F \rightarrow F_L$
$A_P \xrightarrow{(0,0)} P_F$
...

$\mathcal{C}$ - write
F_F: (1)
A_P: (1000)
...
$\mathcal{C}$ - read
F_L: (1)
P_F: (1)
...

Fig. 2. Generated task set

We have developed at ONERA an end-to-end framework, SCHEDMCORE [6], from the design in PRELUDE to the implementation on a uni-processor or a symmetric multi-core platform. The multi-threaded execution is ensured by the SCHEDMCORE runner: the communications are made via shared variables, the tasks can be scheduled by one of the four on-line policies (Fixed-Priority/FP, global Earliest Deadline First/gEDF, global Leats Laxity First/gLLF and Largest Local Remaining Execution First/LLREF) or an off-line sequencer, which sequences the task set from a static trace.

We want to extend the SCHEDMCORE tool set for the SCC, this entails: generating an allocation of the tasks on the cores and generating the message passing code between tasks, instead of the buffer-based communication protocol in the current version. Somehow this would transpose a real-time scheduling problem into a mapping real-time communication problem, i.e. we do not have classical multi-core scheduling problem anymore because every task has a dedicated processor.

III. Execution model on the SCC

We illustrate the implementation of the execution model on the SCC through an example. We first show a static allocation of the example 1 on the SCC and we then detail how to enforce the periodicity and the communication between tasks.

A. Static allocation

The execution model supposes to map statically a unique task on a unique core. This entails that there must be less tasks than cores. This hypothesis is not that restrictive since the current SCC has *48* cores and future many-core will integrate many more cores [1].

One may argue that this would be a waste of computing resources but: hard real-time software designer are first concerned with determinism and then with other constraints like energy consumption. Moreover on a SCC-like processor we may power off the unused processor(s) which would lower the energy waste, our idea with Many-core for real-time is than you do not necessarily want to use all the resource as long as you do not pay too high price (energy, weight, ...) for the unused resource. Unused computing resource may be used in order to implement software redundancy or considered as evolution margin, but this are not our current concerns. For now we just consider we would always have *enough* processor cores.

The illustrative case study is composed of 7 tasks which is much less than 48 cores. There are many possible allocations, exactly $A_{48}^7 = 48 \times 47 \times \ldots \times 42$, but there are not equivalent in terms of worst case response times. For example, the task A_P regularly acquires sensor data from the outside world (position, acceleration). This means that several communications with external (to NoC) I/O system will take place. A_P then sends data either to P_F and N_F, therefore an allocation should take into account the path length and bandwith of the message flow. The criteria for defining a "good" allocation is an on-going work and the experimental results obtained from several allocations will give us general rules to definition what "good" means. An example allocation is shown in Fig. 3, this allocation tries, in a simple manner, to minimise the network traffic. We know that the SCC NoC uses X-Y routing, thus in the mapping F_f is a direct neighbour of F_L and A_P is a direct neighbour of P_F and we know from figure 1 that thoses tasks are linked by a data dependency.

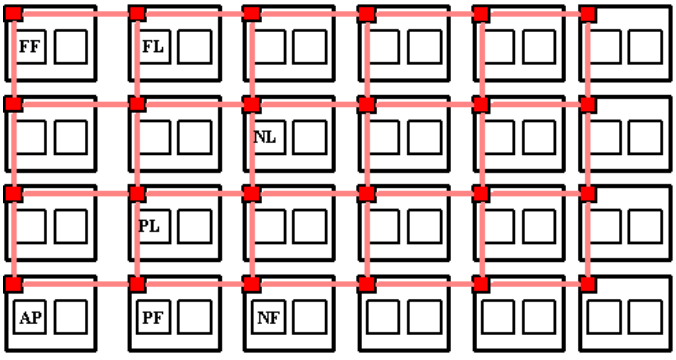


Fig. 3. Example of a static allocation

B. Required features of the SCC

In order to ensure better predictability and more deterministic execution we want to use the SCC in bare metal mode. That way we should be able to avoid any overhead and/or concurrent activities usually impaired by an operating system. Moreover since our execution is essentially to have a single core dedicated to a single task bare metal execution is the most appealing way of use. General purpose program designers may find shared memory programming model more easy to use but time critical systems standard like ARINC653 [9] usually enforce both temporal and spatial segregation/isolation so that the distributed nature of Many-Core may be interesting in the time critical domain. With the SCC the fact that each core may decide which part of its memory is shared is a great advantage. Our execution requires no sharing at all. Since every task has its own core, we need very few usually OS-provide services.

In practice, we are currently beginning to use the ETI Bare Metal Framework [10]. We will explain shortly that all the basic features we need are not currently supported by this framework but we think we should be able circumvent all these shortcomings. In the future we are willing to work with ETI and may be other SCC baremetal programming solution providers in order to define and implement our basic feature needs. Let us examine those needs:

1) Static allocation mapping: We have computed off-line a static allocation, the implementation need would be to be able to load each mapped function on a particular core as depicted on Fig. 3.

Such configuration mapping is not currently possible with current ETI bare metal framework. We can only launch a SPMD program on n cores and the launcher will then load it on core 0 to n. This does not strictly correspond to our need because in fact we do not want SPMD programming. In fact, we will emulate the 1 function on 1 core mapping by loading an SPMD program which will check its rank before running the appropriate function. Moreover if we want to do some testing with less than 48 core and with a particular mapping, e.g. our 7 tasks mapped on processor ID 0,2,4,14,28,36 and 38 we have no other choice than to load the SPMD program on all cores and then enter a `pause` loop on cores which are not supposed to be loaded.

2) Multi-periodicity management: We also have to implement the real-time behavior of each task: inputs must be available at the beginning of the execution, the outputs must be produced on-time (before the deadline) and the task repeats periodically. This means that we must have a way to create bare metal periodic program.

For that purpose, we use the TSC of each core which gives a local time reference. Since the TSCs of different cores are not synchronized, the cores do not have comparable time reference and we must find a way to reconstruct a global vision. We wanted to use the Global TSC introduced in recent sccKit 1.4.x [11] but this is not currently mapped into the memory of the process when using ETI bare metal. In any case, we do not want to generate unwanted traffic on the SCC NoC and since the global TSC (GTSC) is hosted on the FPGA it would not be wise to concurrently read the GTSC at high frequency from possibly 48 cores. We imagine that we could,

during a startup phase, compute the offset between GTSC and TSC of each core in order to use local TSC with a correction offset during run-time in order to have a global time reference with local (on core) access. An alternative would be to have some very low latency global barrier primitive which would makes it possible for each core to exchange their local counter right after such a barrier. The validation of these scheme is an on-going work. Another option would be to use the new 48 interrupt status registers. One core may be dedicated to the *schedule* of other periodic cores. This *scheduling* core would write into appropriate interrupt status registers in order to trigger the new periodic cycle of the concerned cores. Again we need to validate this scheme.

3) Communication management: We then must implement the communication patterns $\mathcal{C}$ with explicit messages. Therefore, our basic communication need is to have a send and receive function (which ensures no loss of data). Ideally both functions should be non-blocking in order to ensure that the periodic tasks do not overrun their period and WCET. Those send/receive function should be specified with a maximum message size for which we are ensured that the message will not be fragmented. This is necessary if we want to able to precisely compute WCTT.

ETI framework and their `streams` API should match part of this need, but the fact that send operation seems to be blocking and we do not know yet the maximum message size and how MPB and/or shared memory are used for messaging (e.g. RCCE is exclusively using MPB for small size and shared memory at some point).

We are currently thinking of directly using SCC MPB (using `-r` option of the ETI launcher) in order to handle raw communication ourselves. This way of work may be interesting in order to better exploit the tight MPB space knowing our static communication pattern. We can do that because we statically know the communication pattern so that we do not necessarily want to divided MPB evenly among the cores. We are currently studying RCCE implementation as an inspiring source of knowledge for that purpose.

4) Current limits and drawbacks: We are currently implementing some test drive programs in order to evaluate the execution model. We did send our feature request to ETI and are willing to work with other MARC people interested in bare metal solutions in order to have this minimal set of features for bare metal:

1) non SPMD bare metal execution with selected cores
2) globally comparable timestamp with local access
3) clean multi-periodic execution.
4) non-blocking send/receive functions with identified non fragmentation maximum message size.

IV. Road-map

Our road-map for the current work is the following.

*a) **Minimal hard real-time bare metal API**:* Our execution will lead us to the specification of a minimal bare metal API usable for the implementation of hard real-time multi-periodic application on many-core architecture. We should come up with such a specification shortly and work on implementation for SCC with SCC bare metal providers.

*b) **WCTT computation**:* Predictable communication is a common need of distributed real-time systems. In civil Aircraft like Airbus A380 or Boeing Dreamliner the ARINC664 (a.k.a. AFDX) is used in order to interconnect the generic computing modules. AFDX is a modified version of ethernet switched network tailored for the deterministic needs of the aerospace industry. Even if the network has been designed for determinism some off-line computation (sometimes huge) [12] must be done in order to compute a tight and secure WCTT. Various mathematical techniques, one of them being the network calculus, are used to compute such bound. In the NoC domain, there are very few available works [13].

We will define a possibly new mathematical model in order to be able to compute the WCTT we need in our application. Beside the reading of Intel SCC documents [7, §6 SCC Mesh] and other network performance related work [14] concerning the SCC NoC which will help us to define this model, we began to develop some [bare metal] test program in order to empirically evaluate some WCTT in the multi-periodic configuration we are interested in.

*c) **Integrate with** PRELUDE-SCHEDMCORE **tool set** :* Our first experiment will validate the proof of concept on hand-written example we will then pursue our work in order to integrate the work into our PRELUDE-SCHEDMCORE tool set. The overall goal would be to program hard real-time multi-periodic system using PRELUDE which would finally be mapped onto the SCC.

*d) **WCET computation**:* The SCC uses relatively old processor cores which may be more predictable w.r.t. to WCET. P54C is a in-order processor with relatively simple 2 level caches with on core L1 (16KB data, 16KB instruction) and unified 256KB L2. Every caches level are private to one core so that there is no concurrent access to the cache between cores. In fact, OTAWA [15] the research tool we usually use for WCET analysis does not currently support Intel processor. In a first raw approximation we will consider that each SCC core does not suffer too much execution time variation provided that we use it in bare metal mode. Moreover, it has distributed memory architecture with thus limited resource sharing, we can either assume that each core has a private memory (private memory ensures the spatial partitioning real-time systems usually requires) whose usage does not interfere with other core activities or that the local memory access is bounded in time, provided that we precisely know each core memory activities and RAM access performance model [14]. We will probably add cache miss instrumentation in our test drive code using Pentium hardware counter programmed with MSR.

V. Preliminary Results: WCTT experiments

We did some preliminary experiment on our Worst Case Traversal Time roadmap. We implemented a test drive SPMD C program using ETI baremetal framework. The program implements two main features:

- a rough time synchronization algorithm which makes it possible to use local TSC as a global clock
- a multi-periodic communicating tasks set with a configurable communication pattern which mimics an AFDX

network communication. The main purpose here is to measure the communication latency on each emulated AFDX Virtual Link.

Our main purpose in this test is to measure a one way communication latency between any core, the receiving core should be able to measure communication latency on his own using local time and a comparable time which has been put inside the packet by the sending core. A globally [to the SCC] accessible reference time like GTSC would have been nice but we want to avoid NoC traffic when measuring network latency moreover GTSC is not available with ETI baremetal. Therefore, we designed a simple synchronization protocol.

A. Time synchronization protocol

We have implemented a basic synchronization protocol which uses the TSC local to each P54C processor core. First we assume that every TSC in an SCC system is running synchronously; this should be true at the hardware level *by-design* as stated in SCC External Architecture Document [7, §7.1]: *The clock distribution is a fast synchronous global grid that gets divided down inside each tile inside the GCU*. Needless to say that in our case we assume that we do not play with the dynamic frequency feature of the SCC. Now even if all TSCs are running synchronously they may have varying offset values depending on the boot/reset time order of each core. Under the previous assumption we know that every TSC are running synchronously with varying offset. The purpose of our synchronization protocol is to find the best approximation of each core TSC offset with respect to *master core*. Note that this "synchronization protocol" has the property of being one shot. As soon as the local synchronization offset are computed, we do not need to communicate for synchronization purpose any-more.

Our current algorithm is the following, first we decide that some core is the master (we use core #0). Then, for each node, the Round-Trip Time (RTT) with the master is computed and the master send a sync message containing its local clock value. The correction is done such that the local reception instant (measured with the local clock of the receiving core) is equal to the master sending instant, plus half the RTT.

This procedure is run at the very beginning of our program, it is run several time in order to filter out possible cache effect. After that we have a global clock whose value may be obtained by each core by reading the TSC and adding it local offset which is a purely local operation. Using this procedure we obtain a global clock with a 20 micro-second precision.

B. Communicating task set description

Our test drive program aims at being similar to Avionics applications which communicates using AFDX network. The AFDX standard (a.k.a. ARINC664) defines the notion of Virtual Link (VL) which is a logical communication link with one source and one or more destinations. Each VL has a specification/configuration which says who is the source and who are the destinations plus temporal and bandwidth attributes. The attributes defines the amount of data conveyed by the VL and its so-called BAG which stands for Bandwidth Allocation Gap. This is the maximum rate data can be sent, and it is guaranteed to be sent at that interval. Our VL configuration on 48 cores is described on figure 4. On this figure each VL is represented by a set of arrows from source core to destination cores. The thickness of each arrow is proportional to the bandwidth. This configuration has an aggregate bandwidth consumption of 960 MBytes/s.

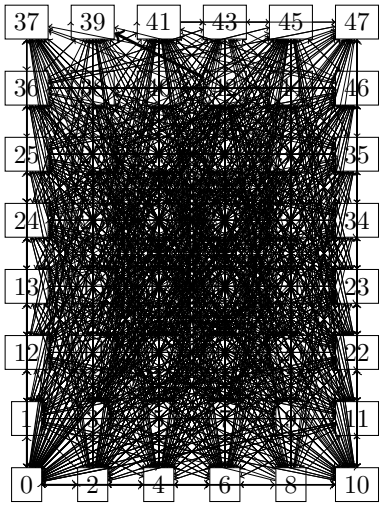


Fig. 4. Communication patterns

Our WCTT test drive program does just that. It loads a VL configuration and naively map all ends of VLs (source and destination) to an SCC core using basic modulo, i.e. VL_i is mapped to core i modulo 48. Notice that each message is less than 1.5Kb, which is quite small [16]. After the synchronization step described in previous paragraph each core uses its global clock to send time-stamped data through its VLs at instant corresponding to its specification. The receiver core finally measures the communication latency by reading its global clock right upon reception and then computes the difference with the time found in the received data.

The figure 5 shows the min, max and mean latency values on each receiving core for the nominal configuration. As we can see the maximum latency is just twice the mean latency and stays below 500 micro-seconds which is very low.

We then show at figure 6 the same configuration for which the bandwidth consumption has been scaled by 10. This scaling is reasonable since the AFDX configuration only models current computer to computer communications, ignoring memory accesses, and communication increase in next generations. As we can see some maximum latency pikes appear on some nodes, while the mean values remain quite the same. In a real-time context, the worst case is more important characteristic. It suggests that our naive mapping is not appropriate since we theoretically consume 9.6GBytes/s of bandwidth which is far less than the theoretical 2 TBytes/s network bandwidth.

VI. Conclusion

We want to evaluate the usage of Many-Core architecture for time critical software. We propose a simple one task to

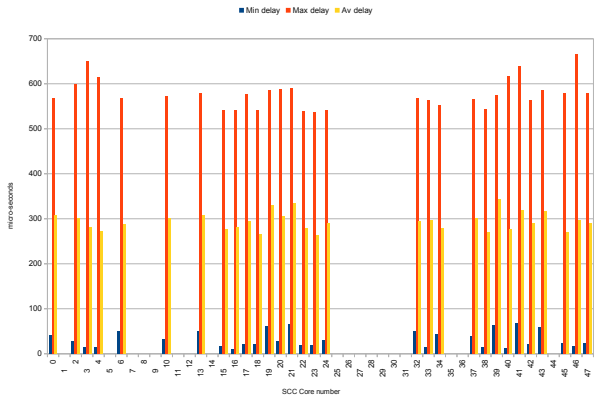


Fig. 5. Results

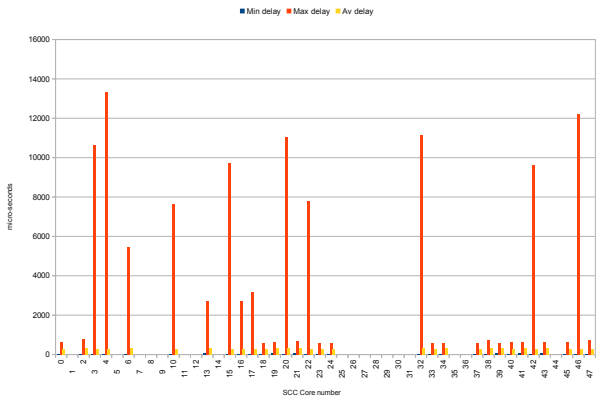


Fig. 6. Results

one core execution model and are currently developing the necessary test drive programs for the SCC. We encounter some limitation on the SCC current bare metal programming framework and suggest some improvements. We developed a test drive program which mimics the behavior of an avionic application using ETI baremetal in order to evaluate the SCC NoC latency performance which is central to our WCTT problem. For this experiment we proposed a simple global clock design which makes it possible to use the TSC as a global clock reference. We will continue to walk on our roadmap in order to map multi-periodic time-critical applications specified with the PRELUDE language on to the SCC using the proposed execution model.

ACKNOWLEDGMENT

The authors would like to thank ET International, Inc. for providing their bare metal framework for our experiment and Intel Labs for providing access to the SCC. This work has been funded by ONERA internal PR-SCC grant.

REFERENCES

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[2] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *8th Symposium on Operating Systems Design and Implementation*, December 2008, http://pdos.csail.mit.edu/corey/.

[3] D. Wentzlaff, C. G. III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "A unified operating system for clouds and manycore: fos," in *1st Workshop on Computer Architecture and Operating System co-design (CAOS),*, January 2010, http://groups.csail.mit.edu/carbon/docs/caos_final.pdf.

[4] S. Peter, A. Schpbach, D. Menzi, and T. Roscoe, "Early experience with the barrelfish os and the single-chip cloud computer," in *Proceedings of the 3rd Intel Multicore Applications Research Community Symposium (MARC)*, Ettlingen, Germany, July 2011.

[5] C. Rochange, "An overview of approaches towards the timing analysability of parallel architecture," in *PPES*, ser. OASICS, P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, Eds., vol. 18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011, pp. 32–41.

[6] M. Cordovilla, F. Boniol, J. Forget, E. Noulard, and C. Pagetti, "Developing critical embedded systems on multicore architectures: the prelude-schedmcore toolset," in *19th International Conference on Real-Time and Network Systems (RTNS 2011)*, IRCCyN lab. Nantes, France: IRCCyN lab, September 29-30 2011. [Online]. Available: http://rtns2011.irccyn.ec-nantes.fr/

[7] I. Labs, "SCC External Architecture Specification (EAS)," Intels, Tech. Rep. version 1.1, November 2010.

[8] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307–338, 2011.

[9] ARINC, "ARINC 653 avionics application software standard interface (part1, required services)," Aeronautical Radio Inc. (ARINC), Tech. Rep., 2005.

[10] *ETI SCC Bare Metal OS Development Framework*, version 1.1.1 ed., ET International, Inc., August 2011.

[11] I. Labs, *The SccKit 1.4.0 Users Guide*, (parts 1-7) revision 0.92 ed., March 2011.

[12] M. Boyer, N. Navet, X. Olive, and E. Thierry, "The pegase project: Precise and scalable temporal analysis for aerospace communication systems with network calculus," in *ISoLA (1)*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 6415. Springer, 2010, pp. 122–136.

[13] M. Tagel, P. Ellervee, and G. Jervan, "Scheduling framework for real-time dependable noc-based systems," in *System-on-Chip, 2009. SOC 2009. International Symposium on*, oct. 2009, pp. 095 –099.

[14] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the intel scc," in *IEEE International Conference on Cluster Computing*, September 2011.

[15] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: An open toolbox for adaptive wcet analysis," in *8th International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS'10)*, ser. LNCS, vol. 6399, 2010, pp. 35–46.

[16] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope, "Efficient memory copy operations on the 48-core intel scc processor," in *Proc. of the 3rd MARC Symposium*.

The SCC and the SICSA Multi-core Challenge

:

Paul Cockshott and Alexandros Koliousis

***Abstract*—Two phases of the SICSA Multi-core Challenge have gone past. The first challenge was to produce concordances of books for sequences of words up to length N; and the second to simulate the motion of N celestial bodies under gravity. We took both challenges on the SCC, using C and the Linux Shell. This paper is an account of the experiences gained. It also gives a shorter account of the performance of other systems on the same set of problems, as they provide benchmarks against which the SCC performance can be compared with.**

I. Introduction

THE SICSA Multi-Core Challenge[1] is an open competition called by the Scottish Informatics and Computer Science Alliance (SICSA) to develop multi-core implementations of a set of predefined problems. Its aim is to learn about the strengths and weaknesses of current systems for parallel programming by comparing them on common grounds.

So far, two phases (viz. Phase I and II) of the Challenge have been run, having attracted entries from teams across Europe. Each phase was announced with a problem specification, together with a candidate serial implementation for that problem. Participants then had to select a programming language, a host architecture, and a paralellisation system with the aim of achieving either the fastest implementation, or the best acceleration, relative to the performance of the serial implementation on that architecture. The results from Phase I were reported at a workshop at the Heriot-Watt University, on the 13th of December, 2010; results from Phase II were reported at a workshop at the University of Glasgow, on the 27th of May, 2011.

We have implemented both of the challenges posed by SICSA on the SCC. Our programming language of choice was C, and the parallelisation system was the Linux Shell – in particular, *Lino*, a process-algebra for chips like the SCC that translates into Linux Shell commands. This paper describes the problems, the SCC implementations, and then contrasts these with other reported implementations, both in terms of design and in terms of performance.

II. Phase I

The first phase of the SICSA Multi-core Challenge was to create concordances of books. The inputs to the problem were a file containing English text in ASCII encoding; and an integer N. The challenge was to find the number of occurrences of all sequences of words up to length N, together with a list of start indices. Optionally, sequences with only one occurrence could be omitted.

School of Computing Science, University of Glasgow, G12 8QQ. E-mail: {william.cockshott, alexandros.koliousis}@glasgow.ac.uk

With thanks to Intel's Many-core Applications Research Community.

[1]www.macs.hw.ac.uk/sicsawiki/index.php/MultiCoreChallenge

TABLE I
Serial benchmarks for Phase I on a twin-core 2.6GHz Intel platform

Language	OS	Print	File Size (bytes)	Time (sec)
Haskell	Windows	yes	4792092	$> 2h$
Haskell	Windows	yes	3580	0.824
C	Windows	no	3580	0.028
C	Windows	yes	3580	0.029
C	Windows	yes	4792091	3.673
C	Windows	no	4792091	0.961
C	Linux	yes	4792091	2.680
C (-O3)	Linux	yes	4792091	2.250
C	Linux	no	4792091	1.040
C (-O3)	Linux	no	4792091	0.899

In addition to the problem specification, a reference implementation in Haskell was provided, together with several reference texts. In practice, most work was done with the longest of texts, the *World English Bible*, which is approximately 4.79MB in size; and with Elizabeth Gaskell's *The Manchester Marriage*, a short story that is 3.58KB in size.

A. An Improved Serial Implementation

Prior to doing any parallelisation, it is advisable to initially set up a good sequential version. Intuitively, the concordance problem is of either linear or, at worst, log-linear complexity, and for such problems – especially ones involving text files – the time taken to read the file and print out the results can easily dominate the execution time. If a problem is I/O-bound, then there is little advantage in expending effort to run it on multiple cores. However, this hypothesis needed to be verified by experiment. In order to obtain an efficient and non-esoteric sequential implementation, C was chosen as the implementation language. The algorithm performed the following steps:[2]

1) read the entire text file into a buffer;
2) produce a tokenised version of the buffer;
3) build a hash table of phrases of up to N tokens and a prefix tree;
4) if the concordance is to be printed out, perform a traversal of the trees printing out the word sequences in the format specified;
5) if the results are to be sorted, pipe them through Linux `sort` command.

The complexity of this algorithm is dominated by the final printing of the results since the volume of output is proportional to the length of the input file, L, times N. The task of constructing prefix trees is of order L [1]. In practice, the printing takes longer than the tree construction.

[2]Available at www.dcs.gla.ac.uk/~wpc/reports/SICSA/concordance.c

TABLE II
PARALLEL BENCHMARKS FOR PHASE I

Programme	Time (sec)
concordance2.c	2.257
l1concordance × 2	2.120

Table I shows that the performance of the C implementation was very much faster than that obtained using the Haskell reference code. Our results also seemed to indicate that there was little practical benefit from parallelising the application since the greatest part of its time was spent formatting and printing the output.

B. The Parallel Implementation

The concordance problem is hard to parallelise efficiently. For example, one can not just split a book into two halves, prepare a concordance for each half, and then merge the results together; a repeated word might be missed if it was mentioned once in the first half and once in the second half. Thus, a more complicated approach was needed. The problem was parallelised by getting several threads to read the entire book, since reading turns out to be relatively fast. The words themselves are then divided into disjoint sets – one obvious split would be into 26 sets based on the first letter. Then, each thread could create the concordance for a disjoint subset of the words. A large part of the time is also taken up with output: the printed concordance can be 5 times as large as the input file. If distinct cores are producing results, there is an inevitable serial phase in which the outputs of the different cores are merged into a single serial file.

As a first parallel experiment, a dual core version of the C programme was produced using the Pthreads library (viz. `concordance2.c`). The programme was tested on the same dual-processor machine as the original serial version of the algorithm, running Linux. Table II shows the results for creating a concordance of the Bible (WEB.txt). There was a small gain in performance due to multi-threading – about 17% faster in elapsed time using 2 cores. Since a large part of the program execution is spent printing the results, this proved a challenge to improve using multiple cores.

The first parallel version allowed each thread to write its part of the results to a different file, which were later merged and sorted. A second parallel version (`l1concordance`) was similar, but used the Linux Shell, instead of Pthreads, to fork parallel processes. This latter parallel version communicates via files using the following set of commands:

```
./l1concordance WEB.txt 4 P 1 0 >WEB0.con &
./l1concordance WEB.txt 4 P 1 1 >WEB1.con
wait
sort WEB1.con WEB0.con
```

In this, $N = 4$ is the maximum number of words in a phrase, `P` indicates that printing is enabled, `1` is the mask to be applied to the hash-code of phrases, and `0` is the value that must result from this masked hash if the phrase is to be handled by this process. As shown in Table II, this version had the best performance of the lot.

TABLE III
SCC PERFORMANCE ON THE CONCORDANCE PROBLEM

Implementation	Time (sec)
1 core; full concordance	26.17
2 cores; $^1/_2$ concordance each	49
8 cores; $^1/_8$ concordance each	36
32 cores; $^1/_{32}$ concordance each	34
1 core on host processor; full concordance	1.03
2 cores on host processor and the Shell; full concordance	0.685

C. SCC Experiments

The SCC is configured with a host processor (MARC), a conventional modern Intel x86 chip. Attached to it is the experimental 48-core SCC chip, each of whose cores runs a discrete copy of Linux. A major worry here was the problem of file I/O for the multiple cores. The source file and the output files were placed (accessed) on (from) a shared NFS system.

Table III shows the results from the SCC experiments. Looking at the time it took one SCC core to complete the full concordance task, one can see that it is much slower than a single core on the host doing the same task. This slowdown is due to the slow access to files from the daughter copies of Linux and the slow performance of the individual cores on the SCC.

A model of the overall time taken on n cores, $n > 1$, is

$$t = \frac{P}{n} + Rn + W \tag{1}$$

where P is the time taken on one of the cores to process the file; R is the time taken to transmit the input file from disk to one of the SCC cores; and W is the time taken to write the results back to disk. Fitting this equation to lines 2, 3, and 4 of Table III, it is $P = 35$; $R = 0.055$; and $W = 31$ seconds. This verifies that writing of the results to disk dominates the total time in the case of 2, 8, and 32 SCC cores.

We dispatched the 32 tasks on the SCC cores using the `pssh` command as follows.

```
rm /shared/stdout/*
pssh -t 800 -h hosts32 -o /shared/stdout \
/shared/sccConcordance32
cat /shared/stdout/* |sort > WEB.con
```

The first line simply removes any temporary output from a previous run. We then use `pssh` to run the script `sccConcordance32` in a shared directory, sending the output to the `/shared/stdout` directory. When all tasks have finished, outputs are concatenated and sorted to yield the final concordance file. The script `sccConcordance32` invokes the actual concordance task:

```
cd /shared
./l1concordance WEB.txt 4 P 31 $(hostname)
```

The hostname command (returning `rck00`, or `rck01`, and so on) is used to derive a process ID, which is then used to select which words will be handled by each task. The 4^{th} parameter to `l1concordance` is the mask that is applied to give the number of significant bits in the process ID, 5 in this case.

TABLE IV
BEST TIMES REPORTED FOR PHASE I

Implementation	Tasks	N	Time (sec)
Java Fork/Join	1	4	134.5
Hadoop Map/Reduce	57 (Beowulf cluster)	10	36
Haskell	8	4	27
Groovy	12	4	61
Python	16	3	2
C on SCC at 0.533 Ghz	32	4	34
C on MARC Host	2	4	0.6

D. Other implementations

Phase I concluded with a workshop at the Heriot-Watt University, were a number of other implementations were presented. Singer reported on the use of Java Fork/Join primitives to implement a parallel version of the concordance problem [2]; Stewart reported on the use of Hadoop Map/Reduce [3]; Al Jabri reported on the use of parallel Haskell [4] and OpenMP [5]; Loidl reported on a parallel C# implementation [6]; Kerridge reported on the use of the new language Groovy in conjunction with JCSP [7]; and Sampson on the use of Python [8]. Apart from the results reported for the SCC, the other systems were run on 2.4GHz multi-core Xeon machines. The results are summarised in Table IV.

One problem with the analysis of these results is that, whilst a word sequence of length $N = 4$ is probably long enough to pick out unique phrases in the Bible, some participants used much longer word lengths, which must have made their output more verbose; some also used different input files, which again makes the results hard to interpret; and other participants gave only relative timings of their parallel and sequential implementations rather than absolute times. The summary of the results in Table IV shows only those implementations that are using the same text file (the Bible, i.e. WEB.txt). It was not always clear whether the reported results included the time to print the final concordance.

Nonetheless, the final conclusion with respect to the SCC is clear. Its performance falls roughly in the middle of the range, with its speed being of the same order as the Hadoop and Haskell implementations. The most successful, highly-parallel version was certainly the Python one, but by a small margin the C version on the MARC host beat its time using only 2 processes. Since the SCC experiments were using exactly the same C code as the version run on the host processor, it should have been fast. The fact that, even with 32 cores, it took approximately 50 times longer can be attributed to the cost of performing input/output from several SCC cores at once.

III. PHASE II

The second phase of the challenge was an N-body gravitational problem – a problem of predicting the motions of a large group of N bodies under gravity. This is inherently a problem of order N^2 on a sequential machine, since each body interacts with every other under gravity. As such, it makes a better candidate for parallelisation than the concordance problem.[3] There are many exemplar benchmark programmes that deal with the N-body problem.

[3]Recall that the latter was of order N and tended to be I/O bound.

SICSA took a C programme from the Computer Languages Benchmarks Game[4] as a reference implementation and modified it slightly so that it handled 1024 bodies. The starting positions, masses, and velocity vectors of bodies in three dimensions were provided as a text file. There were thus seven floating point numbers describing each body.

If we consider the general complexity of this problem under parallelism, one component of the execution time should shrink as the number of processors increases. During each round of the simulation, the program has to accumulate the gravitational forces imposed on each body by all other bodies. Since these calculations are independent, they can in principle be done using different processors in parallel. If p is the number of processors, this stage should have a cost $\alpha\frac{N^2}{p}$, for some constant $\alpha \in \mathbb{R}$. After this calculation has been done, all of the processors would have to ensure that all other processors have access to the same updated data on planetary positions. For a uni-processor this is unproblematic – there is only a single state vector in memory. For multi-processors, however, depending on their design, this communications phase can be an appreciable overhead. If the communications is done naively, the data-transfer cost is $\beta\frac{p^2N}{p}$, for $\beta \in \mathbb{R}$, because processor to processor messages will grow as p^2 and each message will have to send data on N/p planets. We can thus model the overall time taken per simulation step as

$$t = \alpha\frac{N^2}{p} + \beta Np + \gamma \tag{2}$$

where γ is the residual constant in the linear regression. For a shared memory multi-processor, the communications mechanism is effectively the memory bus in association with the cache coherency mechanism, since each processor will have updated its local cache copy of its own planets' positions in phase space, and these local cache copies will have to propagate to the other machines. But this work is also proportional to Np, since each of the p caches has to read a complete copy of the positions of each of the planets. Other communications architectures, including the one used on the SCC, have a similar cost structure.

A. Lino

The compiler group at the Glasgow University School of Computing Science has performed evaluations of the Phase II Challenge using a number of experimental parallelising compilers [9], [10], [11]. This section gives a detailed account of one of those, the Lino system.

Lino is a scripting language originally targeted at the SCC, but it also runs on other Linux machines. It allows Unix Shell commands to be placed on *tiles*, which represent individual processors in an array of available processors. A tile in Lino is represented as `[ cmd0; cmd1; ... ]` where *cmd0*, *cmd1*, etc. is some Shell command.

Tiles can be named, and can be laid out in a rectilinear grid using the '|' and '_' operators. The '|' operation can be used to form a horizontal pipeline of processors; and the '_' operator can be used to form a vertical pipeline.

[4]Cf. http://shootout.alioth.debian.org/

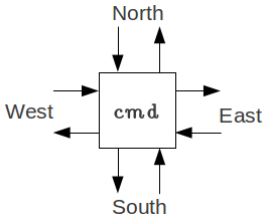


Fig. 1. A Lino tile.

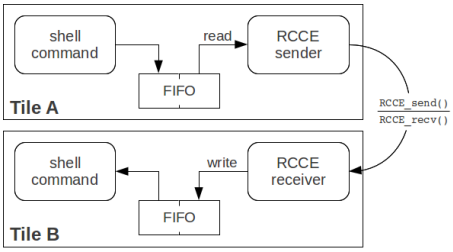


Fig. 2. On the SCC, channels pass via named pipes and RCCE relay processes.

Shell commands communicate with those on adjacent processes by using appropriately named channels, namely North, West, South, and East (Figure 1). Thus, the sequence

```
[ls >East] | [sort <West >file]
```

will cause the `ls` command to run on one tile, sending its output to the east, where it is read by the `sort` command on a right-adjacent tile, whose output goes to a sorted `file`. Geometric operations of 90° rotation and reflection are also supported on tiles or rectangular tile blocks. Tiles can be replicated horizontally or vertically. For more details on the Lino algebra, see [12].

The Lino compiler translates into standard Bash Shell scripts. In the case of the SCC, each tile is allocated a processor core; on other machines, each tile becomes a Linux process. In the latter case, the channels are mapped onto appropriately named Linux FIFO file. On the SCC, however, FIFO files do not work between cores so a five-stage communications process operates, as shown in Figure 2.

When data are passed down a FIFO, a RCCE relay process on the same core reads and sends them as RCCE messages to a corresponding relay process on another core, before being finally piped to another Shell command. This approach allows unmodified C and Shell programs to be linked up in the SCC multi-core environment without the programs having to know about the underlying communications mechanism. It also allows us to benchmark parallel applications both on the SCC and other Intel processors, using the same programmes on both machines.

Without further ado, here is a simple Lino script to run a potentially parallel version of the N-body problem:

```
controller = [./starter >East <East];
worker = [./nbody >West <West];
main = controller | worker;
```

The corresponding layout is shown in Figure 3. For the N-body problem, consider two types of tiles, worker and controller. Tiles are connected in a circle. The controller

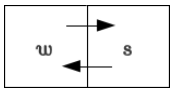


Fig. 3. A 2-core Lino layout for the N-body problem, with one starter (tile s) and one worker (tile w). See Algorithms 1 and 2.

communicates with p workers by messages. A message starts with the character "D", or "U", or "S" to instruct the workers to advance, or update, or terminate the celestial motion, respectively; followed by the number of workers p; followed by the current number of hops the messages has traversed; followed by the positions, velocities, and masses of N bodies – all in all, a 65KB message. A controller tile runs the C programme `starter` which goes through the following sequence:

1) read in the initial position of the planets from a file;
2) request from the worker(s) to perform one simulation step on the data by:
   a) writing the planet data, preceded by a "D" character, on standard output, and
   b) waiting for the corresponding "D" message to arrive on standard input;
3) request the worker(s) to update the data by:
   a) writing the planet data preceded by a "U" character, on standard output; and
   b) waiting for the corresponding "U" message to arrive on standard input, and then storing the new planet positions.
4) If the required number of simulation steps have finished, send the worker(s) an "S" message on standard output and terminate, otherwise go to step 2.

The N-body worker programme itself (referred to as `nbody`) is a slightly modified version of the reference single processor implementation in C, waiting (in a loop) to read messages on standard input. The N-body programme branches on the first character of the message as follows:

- if the header starts with a "D", then increment the number of hops and write the message to standard output. Then, simulate the dynamics of N/p planets for one time-step, and remember their new positions in phase space;
- if the header starts with a "U", then increment the number of hops, and copy into the message the updated positions of the planets for which the worker is responsible for, before writing the message to standard output;
- if the message starts with an "S", terminate.

This approach allows us to vary the number of workers associated with each controller without changing the C code. For example to have 4 workers we use the Lino script:

```
nwcorner = [./nbody >East <South];
swcorner = [./nbody >North <East];
scorner = [./starter4.sh >South <West];
corner = [cat >West <North];
passright = [./nbody >East <West];
passleft = [./nbody >West <East];
top = [nwcorner | passright | scorner];
bottom = [swcorner | passleft | corner];
main = top _ bottom;
```

TABLE V
TIME/SIMULATION STEP OF THE N-BODY PROBLEM IN LINO ON THE SCC AND ON AN 8-CORE INTEL XEON E5620 @ 2.4GHZ

N-body workers (cores)	Time on Xeon (ms)	Time on SCC (ms)
16 (20)	8.1	2032
8 (10)	7.8	1025
4 (6)	9.9	702
2 (4)	17.1	648
1 (2)	30.5	967

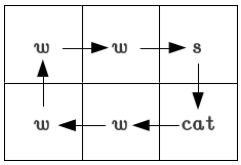


Fig. 4. A layout with 4 worker cores.

This gives the layout shown in Figure 4. We have tested layouts for 1, 2, 4, 8, and 16 worker cores, both on the SCC and on an 8-core 2.4GHz Xeon E5620. On both machines, the same C and Lino code was used. Table V shows the results.

The SCC is almost two orders of magnitude slower than the Xeon. Some of this may be attributed to the earlier version of GCC used on the SCC, some to the slower clock used and some of it to the earlied Pentium design used. But one might have hoped that these disadvantages would have been offset by the opportunity too use more parallelism. On the contrary, we find that the SCC implementation peaks at 2 worker processes, whilst the Xeon peaks at 8. That this slowdown is due to the inter-core communication mechanism on the SCC, rather than to the use of Linux FIFOs, is proven by the fact that the Xeon Lino implementation which also used FIFOs but which did not use RCCE was faster.

Fitting Equation 2 to the data in Table V, we obtain for the Xeon $\alpha = 27$ns and $\beta = 223$ns, whereas for the SCC $\alpha = 677$ns and $\beta = 94\mu$s. Recall that α is the time to compute the interaction between two planets and β the time taken to communicate one planet data between two workers. The SCC is slower on both counts, but is much slower on communications. This means that the level of parallelism that can be supported before the costs of communications comes to dominate is lower on the SCC.

B. Other Implementations

Similar to the first phase of the SICSA Multi-core Challenge, Phase II concluded with a workshop at the University of Glasgow. The results are summarised in Table VI, ordered by their overall performance. Where multiple results were reported for a language/processor pair, the fastest time is given.

Thomas Horstmeyer [13] reported on an implementation using Eden [14]. As Table VI shows, this had a relatively poor performance, being slower than the single thread C reference version, and about half the speed of Lino on the same hardware. The C# implementation reported by Loidl had similar performance [15]. Sampson, whose Phase I entry was very fast, reported on an impressive implementation using SSE vector intrinsics and Threading Building Blocks (TBB) [16]. This appears to have one of the fastest performances of all,

TABLE VI
BEST TIMES REPORTED FOR PHASE II ON 8-CORE XEON MACHINES

Implementation	Threads	Time (ms)	Clock (Ghz)
Glasgow Pascal (SSE) (a)	16	1.75	2.4
C++ (SSE) (a)	12	2.05	2.27
Glasgow Pascal, AVX (b)	4	2.12	3.1
Lino on Xeon (a)	10	7.8	2.4
Go (a)	16	8	2.4
C sequential (a)	1	14	2.4
Eden (a)	8	16.6	2.5
C# (a)	12	18.2	2.33
Glasgow Fortran (E#) on Cell	12	23	3.2
GCC on Cell	1	45	3.2
Glasgow Pascal on Cell	4	48	3.2
Gnu Fortran on Cell	2	82	3.2
Lino on SCC	2	648	0.533

(a) 8-core Intel Xeon E5620; (b) 4-core Intel i5-2400

which is a credit to the efficiency of the TBB and the gains to be had from SSE intrinsics.

The Glasgow results [17], [18], [19] are polarised according to the processor and type of language used. Lino and Go are slower than Pascal; the Cell is slower than conventional Intel machines; and the SCC is slower than the Cell. This ranking of machines is born out across all results reported at the workshop, although the lower clock speed of the SCC is clearly a factor that has to be taken into account here. Indeed, if we normalise for clock speed, the Lino on the SCC falls into the same range of performance as GNU Fortran on the Cell.

IV. CONCLUSIONS

The SCC is described as a Single Chip Cloud. The performances we have observed for it indicate that this may be an accurate description. On the concordance application the SCC performance most closely resembled that of Hadoop on a Beowulf cluster – a more classic cloud configuration. Compared, however, with other multi-core chips (e.g. Nehalem, Sandybridge or the CellBE), the SCC performs poorly on both applications. The SCC experiments, particularly those for Phase II, indicate that the underlying cause for the uncompetitive performance of the SCC is the inefficiency of the inter-core communications system. Unlike the CellBE which performs inter-core communications using high speed DMA, or the Nehalem which uses cache coherence hardware, the SCC relies on software message passing in small shared buffers. We conclude that if tessellation processors like the SCC are to be viable, they will require high performance DMA hardware.

A separate conclusion from our experiments is that the old Unix Shell model of parallelism – C programmes communicating via files and pipes – is still remarkably effective. It gave the highest performance for the Phase I problem and for Phase II, it was only beaten by compilers that made explicit or implicit use of SIMD parallelism.

REFERENCES

[1] E. Ukkonen, "On-Line Construction of Suffix Trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

Algorithm 1 The single worker N-body example compiled for the SCC.

```
#!/bin/sh
shift `expr $1 + 2`
[ ! -d $1 ] && exit 1
cd $1
case `hostname` in
rck00)
mkfifo fifos/East0_0
mkfifo fifos/East0_0in
# 'hello' is the RCCE relay process between cores 00 and 02
./apps/HELLO/hello 2 0.533 00 02 --from fifos/East0_0 --to /dev/null &
./apps/HELLO/hello 2 0.533 02 00 --from /dev/null --to fifos/East0_0in &
./starter.sh  > fifos/East0_0 < fifos/East0_0in  &
wait
rm fifos/East0_0
rm fifos/East0_0in
;;
rck02)
mkfifo fifos/West0_1
mkfifo fifos/West0_1in
./apps/HELLO/hello 2 0.533 02 00 --from fifos/West0_1 --to /dev/null &
./apps/HELLO/hello 2 0.533 00 02 --from /dev/null --to fifos/West0_1in &
./nbody > fifos/West0_1 < fifos/West0_1in &
wait
rm fifos/West0_1
rm fifos/West0_1in
;;
esac
```

Algorithm 2 The single worker N-body example compiled for a shared memory Linux machine.

```
rm fifos/*
mkfifo fifos/East0_0
mkfifo fifos/West0_1
./starter1.sh >fifos/East0_0 <fifos/West0_1 &
./nbody <fifos/East0_0 >fifos/West0_1 &
wait
```


[2] J. Singer. (2010, December) Java Fork/Join Implementation. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.dcs.gla.ac.uk/~jsinger/pdfs/sicsa_concord_101213.pdf

[3] R. Stewart. (2010, December) Hadoop MapReduce Concordance. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/~rs46/multicore_challenge1/Hadoop_concordance.pdf

[4] M. Aljabri. (2010, December) A Parallel Concordance Benchmark, Haskell Implementation. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/events/MultiCoreChallenge/slides/aljabri_mcc10.pdf

[5] ——. (2010, December) A Parallel Concordance Benchmark, OpenMP Implementation. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/events/MultiCoreChallenge/slides/aljabri_mcc10.pdf

[6] H.-W. Loidl. (2010, December) Parallel Concordance in C#. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/events/MultiCoreChallenge/slides/hawo_mcc10.pdf

[7] J. Kerridge. (2010, December) SICSA Concordance Challenge: Using Groovy and the JCSP Library. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/events/MultiCoreChallenge/slides/jon_mcc10.pptx

[8] A. Sampson. (2010, December) "This is a parallel parrot". SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://offog.org/publications/mcc201012-python-slides.pdf

[9] P. Keir, W. Cockshott, and A. Richards, "Mainstream parallel array programming on cell," in *5th Workshop on Highly Parallel Processing on a Chip (HPPC 2011)*, 2011. [Online]. Available: http://eprints.gla.ac.uk/54875/

[10] W. Cockshott and G. Michaelson, "Orthogonal parallel processing in vector pascal," *Computer Languages, Systems and Structures.*, vol. 32, no. 1, pp. 2–41, April 2006. [Online]. Available: http://eprints.gla.ac.uk/3451/

[11] Y. Gdura and W. Cockshott, "A virtual simd machine approach for abstracting heterogeneous multi-core," in *ICT 2011 18th International Conference on Telecommunications*, 2011. [Online]. Available: http://eprints.gla.ac.uk/56324/

[12] P. Cockshott and G. Michaelson. (2010, April) Lino: a tiling language for arrays of processors. University of Glasgow, School of Computing Science. [Online]. Available: http://www.dcs.gla.ac.uk/~wpc/reports/linopaper.pdf

[13] T. Sauerwein, T. Horstmeyer, and M. Dieterle. (2011, May) N-Body in Eden - A skeletal approach in a distributed memory setting. SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.mathematik.uni-marburg.de/~horstmey/sicsa/NBodyEdenSlides.pdf

[14] A. Black and U. of Washington. Dept. of Computer Science, *The Eden programming language*. Dept. of Computer Science, University of Washington, 1985.

[15] H.-W. Loidl. (2011, May) A C# implementation of the n-body problem . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/events/MultiCoreChallenge/slides/mcc11.pdf

[16] A. Sampson. (2011, May) Colliding Blobs with Threading Building Blocks . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.mathematik.uni-marburg.de/ horstmey/sicsa/NBodyEdenSlides.pdf

[17] Y. G. P. Cockshott. (2011, May) Vector Pascal implementations running on Nehalem and Cell processors . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.dcs.gla.ac.uk/~jsinger/ pdfs/wpc_multicore.pdf

[18] P. Keir. (2011, May) All-pairs n-body in Fortran for CellBE . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.dcs.gla.ac.uk/people/personal/pkeir/mcore2.pdf

[19] I. McGinniss. (2011, May) Naive approaches to n-body parallelism using Google Go . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://prezi.com/qrgmjzexqvgp/naive-approaches-to-n-body-parallelism-with-google-go/

Experiences in porting the SVP concurrency model to the 48-core Intel SCC using dedicated copy cores

Roy Bakker and Michiel W. van Tol
Informatics Institute, University of Amsterdam
Sciencepark 904, 1098 XH Amsterdam, The Netherlands

***Abstract*—The Single-chip Cloud Computer (SCC) is a 48-core experimental processor created by Intel Labs targeting the many-core research community. It has hardware support for sending short messages between cores, while large messages have to go through off-chip shared memory. In this paper we discuss our implementation of the SVP model of concurrency on this architecture, and how we deal with its distributed memory design and communication bottlenecks. We employ our previously developed *copy core* technique and show which approaches show scalable performance against our original implementation.**

I. Introduction

The Single-chip Cloud Computer (SCC) experimental processor [1] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. It provides an on-chip message passing network, a non cache-coherent off-chip shared memory and dynamic frequency and voltage scaling. In this paper we discuss our implementation of SVP on this platform, a hierarchical concurrent execution model [2]. In future work, we will use this implementation and exploit its dataflow-style execution to provide us with a handle for adaptive power management.

The Self-adaptive Virtual Processor, or SVP, is an abstract concurrent programming and machine model, which evolved from the earlier work on the Microthread CMP architecture [3] which implemented SVP in hardware [4]. The model can be used to express concurrency at many levels of granularity for multi- or manycore systems, and uses shared memory semantics with a weak consistency model. As the SCC has a distributed shared memory architecture without cache coherency, this suits the consistency model of SVP very well. As SVP actions can be trivially translated into messages in a distributed environment, this maps well onto the message passing communication infrastructure of the SCC.

Effectively, the SCC is an on-chip distributed system, and therefore we can already run the available distributed implementation of SVP [5] without any modifications. As this is based on the coarse grained communication primitives of TCP/IP sockets, we experimented with different approaches to more efficiently use the hardware messaging support on the SCC. However, we have already shown in previous work [6] that using the on-chip message passing buffers with RCCE [7] or iRCCE [8] are not sufficient for such an implementation. In this paper we will employ several of the techniques that we investigated in our earlier work to efficiently copy memory on the SCC, for example by using dedicated copy cores.

In this paper we will discuss our experiences with porting the distributed SVP runtime to the SCC. We assume sufficient knowledge of the SCC architecture and its memory system as this is broadly covered by both related work [1], [9] as well as our previous [6] work. First we will discuss the SVP model of concurrency and its consistency model in Section II, and discuss which approaches we considered for the implementation on the SCC in Section III. In Section IV we evaluate these different approaches and we conclude with a discussion and future work in Section V.

II. SVP

SVP is a generic concurrent programming and machine model which has a separation of concerns between the expression and management of concurrency. The SVP model defines a set of actions to express concurrency on groups (*families*) of indexed asynchronous activities (*threads*).

Each thread can execute a *create* action to start a new concurrent child family of threads, making the model hierarchical, and later on use the *sync* action to wait for its termination. The *create* action has a set of parameters to control the number and sequence of created threads, as well as a reference to the thread function that the threads will execute. This thread function can have a set of communication channels defined that are explained later on.

Besides these two basic constructs, there is the *kill* action to asynchronously terminate an execution. Programs for SVP based architectures or run-times are written in a dialect of the C language which has extensions to explicitly support these SVP actions and thread definitions.

A. Resources

While SVP code has no notion of what a resource physically is or how code is scheduled, an abstract resource identifier, a *place*, is provided. On a *create* action a *place* can be specified where the new family should be created, binding the execution onto a certain resource, similar to sending an Active Message [10]. What this *place* physically maps to, is left up to the SVP implementation; for example, on our implementation on the SCC it will be a single core, but on the Microgrid CMP [11], it is a group of cores. On other implementations it could, for example, be a reserved piece of FPGA fabric, an ASIC, or some time-sliced execution slot on a single- or multi- processor system.

As long as the underlying implementation supports it, multiple *places* can be virtualized onto a single physical resource. Mutual exclusion is supported through places; families delegated to an *exclusive place* are guaranteed to be sequentialized so that only one family can be executing on such a place at a time.

B. Communication and Synchronization

Each family has a set of synchronized communication channels that link up the threads and the parent context. There are two types of unidirectional write-once channels; *global* and *shared* of which multiple can be present. These channels have non-blocking writes and blocking reads. A *global* channel allows vertical communication from the parent thread to all threads in the family. A *shared* channel allows horizontal communication, as it daisy-chains through the sequence of threads in the family, connecting the parent to the first thread and the last thread back to the parent. These channels are defined as arguments of a thread function and identify the data dependencies between the threads.

Due to this restricted definition, and under restricted use of exclusive places, we can guarantee that the model is composable and free of communication deadlock [12], and that there is always a well defined sequential schedule if parallel execution is infeasible.

C. Memory Consistency

The model assumes a shared memory with a restricted consistency model. It is seen as asynchronous and therefore it is not suitable for synchronizations, and no explicit memory barriers or atomic operations are provided. The consistency model is described by the following three rules:

- A child family is guaranteed to see the same memory state as the parent thread saw at the point of *create*.
- The parent thread is only guaranteed to see the memory changed by a child after *sync* on the child has completed.
- A family on an *exclusive place* is guaranteed to see the changes to memory by earlier families on that place.

The memory consistency relationship between parent and child threads is similar to the well-known *release consistency* model [13]. The *create* resembles an *acquire*, and *sync* resembles the *release*. We should note that the third rule is a very important property as it can be used to implement communication between two arbitrary threads, but it can also be used to implement a service; state is resident at the *exclusive place* and instances of the functions implementing that service are created on the *place* by its clients.

D. Distributed SVP

Distributed SVP, or DSVP, is an extension to SVP to handle distributed memory [5]. The implementation of DSVP was our starting point for an SVP implementation on the SCC. The DSVP extension introduced the idea of a *data description function* which tells the implementation which parts of memory need to be sent/received when a thread function is started remotely with a create or completes with a sync, similar to how in- and outputs are annotated in CellSs [14] and Sequoia [15]. This is based on the premise that a thread needs to receive a reference to any data it will access through its communication channels, and therefore this identifies which data needs to be communicated to adhere to the consistency model.

```
thread fibonacci(shared int p1, shared int p2, int* result)
{
    index i;
    result[i] = p1 + p2;
    p2 = p1;
    p1 = result[i];
}
DISTRIBUTABLE_THREAD(fibonacci)(int p1, int p2, int* result, int N)
{
    INPUT(p1);
    INPUT(p2);
    ARRAY_SIZE(result, N);
    for(int i = 2; i < N; i++)
        OUTPUT(result[i]);
}
main()
{
    family fid;
    int    result[N];
    int a = result[1] = 1;
    int b = result[0] = 0;

    create(fid;;2;N;;) fibonacci(a, b, result);
    sync(fid);
}
```

Figure 1: Fibonacci code example

An example code is given in Figure 1, showing a program that stores the Fibonacci sequence up to N into a result array. Threads 2 to N are created for the corresponding iterations and they communicate their dependent values through their shared channels p1 and p2. The data description function takes the two initial values for p1 and p2 as input, and returns the resulting fibonacci array as output. Please note that some parameters for create are omitted, for example one to set the *place* where the computation is executed and others to control more complexing indexing.

III. Implementation

The distributed SVP implementation that uses TCP/IP for communication between places [5] runs on the SCC without any modifications. However, it supports heterogeneous platforms with different data representations, which adds additional overhead. All data that needs to be communicated (indicated by a *data description function*) is serialized to a platform independent representation using XDR [16] before it is sent to the other *place*, where it needs to be deserialized again. On the SCC we can avoid this step, as it is a homogeneous system with the potential to use shared memory.

Our initial optimization was to skip the XDR step and send each data element that is part of the data description function directly through the socket. For scalar values this causes a large communication overhead, since they are now all pushed separately through the channel. For (large) arrays this means a great reduction in the overhead of encoding and copying, especially on the SCC where memory operations are expensive. The data description function was altered to support sending arrays in a single shot. To send arrays, we no longer need to explicitly touch each single element of the array, but

just provide a pointer to the first element, and the number of elements in that array.

A. Using (i)RCCE

Our first approach was to modify the communication layer of the existing implementation to make use of the RCCE and iRCCE libraries. We could easily replace all send and receive calls with the appropriate iRCCE functions. For the connection establishment we used an iRCCE waitlist with a receive request for each possible sending core. However, the (i)RCCE implementation only matches requests on core identifier. As a result, we cannot have multiple outstanding receive requests for a single sending core, and the first message that fits the size will complete. Therefore, we cannot issue another receive request in the connection establishment function while there is already a connection active. The messages in the connection establishment function are rather small, and therefore match any other larger sized message. We see that messages sent through a previously established connection now initiate a new connection, and fail thereafter. iRCCE does not support virtual channels, and therefore was not suitable for our SVP implementation in its current form.

B. Memory Remapping

The SCC allows us to share memory from one core with an other by using the programmable *look-up tables* (LUT), which means that the communication of large data chunks through a channel can be avoided. However, the virtual memory system of Linux makes this difficult as the virtual addresses seen by a process are not the same as the physical addresses. A function in the special SCC Linux memory kernel driver provides a virtual to physical address translation. The sccLinux virtual memory system uses 4KB pages, and chunks of contiguous virtual memory that span more than one page, therefore do not necessarily map to contiguous core-physical memory. However, as sccLinux does not support the use of swap space, the virtual to physical address mapping of a page is stable. Once the core-physical memory address is known, the real-physical address can easily be obtained from the LUT.

To make effective use of the memory remapping approach, we need to manage our own virtual to core-physical memory mappings, avoiding the fragmentation induced by the sccLinux virtual memory system. We use an sccLinux image that has only 320MB of private memory configured, which leaves about the same amount of memory for the application to manage as there is 656MB of private memory reserved for each core. We *mmap()* this region so that we have a contiguous mapping of virtual to core-physical addresses, and within that use our own memory allocator with a simple first-fit algorithm.

When using memory remapping, we can either use cached memory or non-cached memory. Cached memory has the downside that the L2 cache is write back and therefore needs to be flushed, which is an expensive [6] operation, to make sure all data will be in physical memory on the sending side. We can avoid the L2 flush at the receiving side by mapping the memory with the MPBT tag on, so that the L2 cache is bypassed, but multiple accesses within the same cache line will hit in the L1 cache. Then, it is enough to issue the cheap CL1INVMB instruction that invalidates all data in the L1 cache with the MPBT tag. The alternative, the use of non-cached memory, was not considered; it is too expensive as every individual access needs to go to main memory.

In the memory remapping implementation, the sending core will send the core-physical address through the socket to the receiving core. The receiving core checks the LUT of the sending core to obtain the real-physical address. It will dynamically map this address range on free LUT pages and start a memory copy operation to the receive buffer. As the initial socket communication, lut writing procedure and cache flushing will cause overhead, this approach is only efficient when the message is large enough to compensate for this overhead. To allow faster writing, the target area is remapped with the MPBT flag on, which enables the WCB. As these addresses are independent from the L2 cache perspective, and MPBT bypasses the L2 cache, no additional flushes are required in this approach.

The remapping approach is similar to the *Privately Owned Public Shared Memory* (POPSHM) approach proposed by Intel. However, POPSHM requires a copy of the data into the shared memory region on the sending side, and a copy out of the shared memory region on the receiving side. In contrast, we map the memory locations directly on demand at the receiving side, therefore only requiring a single copy operation which uses specialized memory flags for the fastest possible reading and writing.

C. Copy Cores

Instead of performing the memory copy operation at the receiving core, we can also choose to use our earlier proposed *copy cores* [6] to copy memory regions. Copy cores are dedicated cores that run a memory copy service; when data needs to be copied between cores, multiple copy cores can be employed to copy the data, similar to DMA engines, which due to the limited memory throughput of a single core should be able to deliver a better performance. Copy cores use the same approach to copy memory as the remapping implementation, using specialized flags for reading and writing. In the current implementation, all copy tasks are issued round robin to a set of copy cores.

IV. Evaluation

A. Benchmarks

1) Ping Pong: The first benchmark that we use is an SVP based Ping-Pong application which creates a computation on the remote node which terminates immediately, but using the data description function sends chunks of data back and forth with incremental size. We use this benchmark to measure the latency and throughput achieved by our different approaches. The sizes we measured range from 4 bytes up to 16MB, and are transferred between cores 0 and 1.

2) Matrix Multiplication: A benchmark that fits the distributed implementation of SVP with the potential to copy lots of data is matrix multiplication. We implemented a recursive decomposition algorithm that splits a matrix in sub matrices

and performs the calculations on sub matrices only. Figure 2 shows the decomposition algorithm. We can apply the decomposition recursively as long as the square matrix size is still dividable by two. Each step splits the calculation in eight parts that can execute concurrently, followed by four additions that can also execute concurrently. Note that the addition can be performed on the individual sub matrices, or on the combined larger matrices. In this benchmark we perform the addition on the sub matrices, since this exposes more concurrency without changing the representation again. This implementation works on square matrices and operates on double precision floating point values.

$$A \times B \rightarrow \begin{vmatrix} a1 & a2 \\ a3 & a4 \end{vmatrix} \times \begin{vmatrix} b1 & b2 \\ b3 & b4 \end{vmatrix} =$$
$$\begin{vmatrix} a1 \times b1 & a1 \times b2 \\ a3 \times b1 & a3 \times b2 \end{vmatrix} + \begin{vmatrix} a2 \times b3 & a2 \times b4 \\ a4 \times b3 & a4 \times b4 \end{vmatrix} = \begin{vmatrix} c1 & c2 \\ c3 & c4 \end{vmatrix} \rightarrow C$$

Figure 2: Matrix decomposition: Matrices A and B (both $N \times N$ are split into four $\frac{N}{2} \times \frac{N}{2}$ matrices each. Eight matrix multiplications and four matrix additions are performed on the sub matrices.

In order to make the decomposition more time and space efficient, the matrix representation in memory is a column of pointers that all index a row in the matrix. All matrix rows together form one contiguous block of memory, both virtual and physical, guaranteed by our own memory allocator. This representation is visualized in Figure 3. The normal lines indicate a pointer to an element in memory, while the dashed lines refer to the same element in the corresponding matrix. Pa, Pb, Pc and Pd are pointers to arrays with pointers to sub matrix rows. This allows us to do the decomposition by creating a new array of pointers and assign the pointers to elements in the original matrix rows, without the need of copying data.

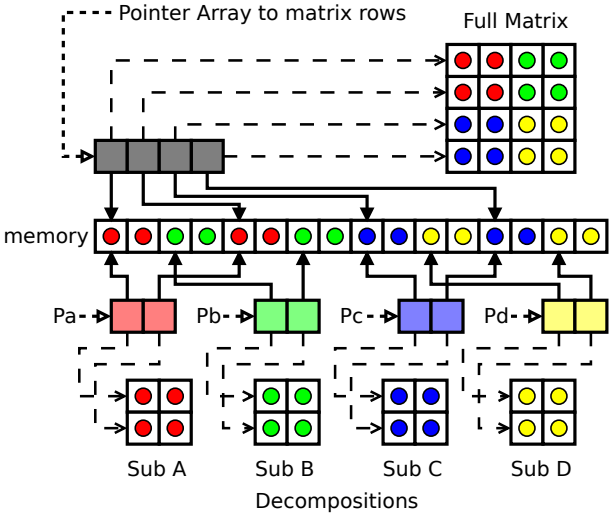


Figure 3: Representation of the original and decomposed matrices in memory

We have run two versions of the matrix multiplication benchmark with different distribution strategies. The first has only one master node that decomposes the matrices and sends the sub matrices to worker nodes. Initial experiments have shown that only a single master node can not keep the other cores busy when we use two decomposition steps creating $8 \times 8 = 64$ concurrent multiplications on only 47 (or 48, when the master is included) nodes. The overhead for the decomposition and communication is too large compared to the computation performed by the worker nodes. In the second version the master node does a single recursion step, and then delegates the work to eight nodes which in turn do the second recursion step to create a total of 64 tasks. Using this method, we divide the communication overhead over multiple nodes, but the total amount of required communication is higher.

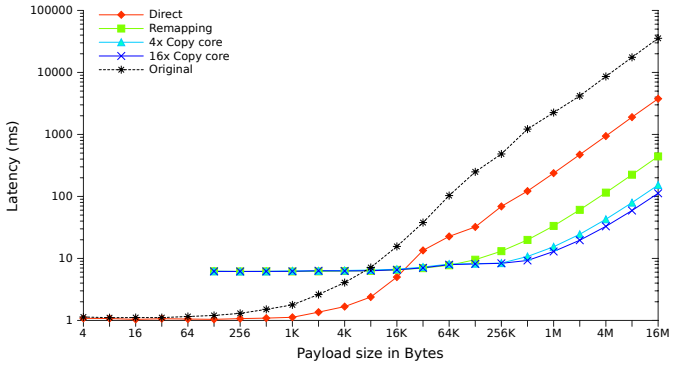
B. Results

1) Ping Pong: In Figure 4, the results of the Ping Pong benchmark are visualized in two graphs. The first graph shows the best achieved latency of creating a remote computation, followed by a synchronization directly thereafter with different data payloads. All results are the minimum over 10 measurements, to compensate for outliers generated by TCP/IP timeouts that would have a large impact on an average. We show the results for the previously discussed implementations; *Direct* is the same implementation as the original but without the XDR encoding and decoding steps, *Remap* is the approach that remaps and copies the memory on the receiving side, and *Copy core* is spreading the copy operation over 4 or 16 dedicated copy cores.

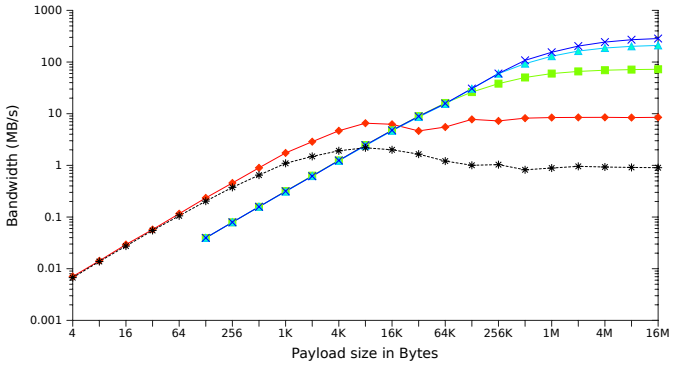
The new approaches have a higher latency, around 6 ms instead of 1 ms, for a small payload, due to the required L2 cache flush, but have a much better latency when communication a lot of data. This is further shown in the second graph, which shows the corresponding throughput, note that this graph is also on a log to log scale. As the initial communication of the addresses still goes through TCP/IP, we set a threshold for using remapping or copy cores to 128 bytes, however they only become faster then the direct approach when transferring more then 16 KB.

The direct communication approach reduces the execution time by almost an order of magnitude compared to the original implementation. The speedup peaks at a factor of 9 for messages larger than 512KB. Remapping memory is about two orders of magnitude faster than the original implementation. The copy core approach clearly improves on the memory remapping approach, as it can aggregate more bandwith by using multiple cores, as described in [6]. It is three times as fast as remapping when using 4 copy cores, and four times as fast with 16 copy cores, where you start to notice the delegation and synchronization overheads to send the Copy cores their work requests.

2) Matrix multiply using one decomposition step: In this benchmark we run our matrix multiplication application using one decomposition step, resulting in eight remote creates. We ran the benchmark for square matrix sizes of 128, 256, 512 and 1024 elements, resulting in sub matrices of half that square size. The amount of communication is order $O(n^2)$ while the computation is in the order $O(n^3)$, which results in better scalability for larger matrices due to a better computation to



(a) Latencies for the different implementations



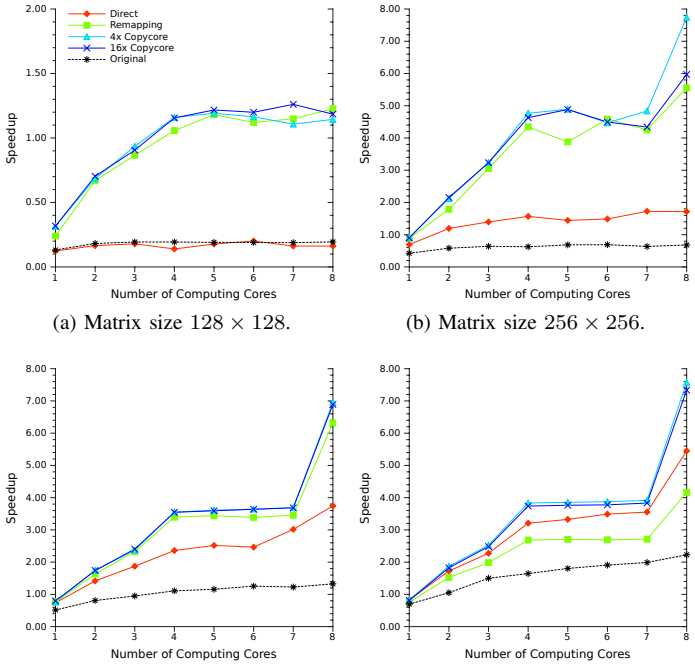
(b) Throughput for the different implementations

Figure 4: Results for the SVP based Ping Pong benchmark

communication ratio. The master node initializes the matrices, performs the decomposition, and distributes the work over 1 to 8 remote places. The results averaged over three runs and are shown in Figure 5. We benchmark again our four implementations; the original, the direct implementation, remapping and copy cores with 4 or 16 copy cores which are placed at the edges of the SCC chip around the memory controllers. All speedups are measured against a baseline of a non-threaded local matrix multiplication using the same computation kernel but without distribution or decomposition.

For a small matrix size of 128×128 elements (Figure 5a), we see the impact of the large overhead of communication. The original and direct implementation perform about the same but do not scale at all. In this case, the messages are rather small due to the memory layout of the matrices. Every row of each matrix has to be sent separately, consisting of 64 double precision floating point elements of 8 bytes each, resulting in a message size of 512 bytes. Using the original and direct implementations, each of these messages will be sent separately, resulting in a lot of TCP/IP overhead. The remapping and copy core implementations show some scalability as they only receive a list of addresses that they have to copy their data from. The L2 cache also does not have to be flushed for every message, as it recognizes that all these messages together are part of the same remote computation.

For size 256, (Figure 5b), we still do not see a lot of speedup for the original and direct implementation. The data size per message has increased to 1KB, which is a clear advantage to the remapping and copy core approaches, where the measurement with 4 copy cores manages to nearly gain a perfect speedup of 8. For size 512, (Figure 5c), the direct implementation starts to show some scalability for multiple cores, scaling up to a speedup of 4. The remapping and copy core approaches scale well and perform roughly the same, though the latter shows some superlinear speedup for 4 cores, probably as the data fits well into the L2 cache. The last graph (Figure 5d) shows similar results, except that the remapping implementation is now clearly outperformed by the copy cores as they provide more communication bandwidth. The original implementation again scales poorly, which is caused by the different ratio between communication bandwidth and computational power, compared to a cluster environment.



(a) Matrix size 128×128.

(b) Matrix size 256×256.

(c) Matrix size 512×512.

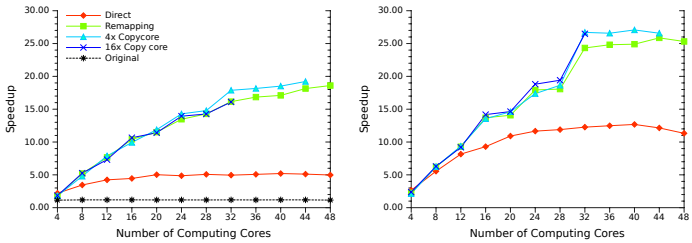
(d) Matrix size 1024×1024.

Figure 5: Matrix multiply with 1 decomposition step

3) Matrix multiply using two decomposition steps: The matrix multiplication benchmark exposes eight times the concurrency for each decomposition step that is performed. However, the additional recursion step leads to more communication overhead due to increased number of messages. We ran this benchmark for sizes 1024×1024, and 2048×2048, but the original implementation could not run on the latter size due to memory constraints.

In Figure 6, we see no speedup for the original implementation when using additional cores. It fails to scale as the master node is fully occupied with the distribution of tasks while most of the workers are idle waiting for work. This also limits the scalability of the direct approach to about a factor of 5 at 20 cores, but this is not the case for remapping or copy cores. As these two approaches fetch the memory, more concurrency in the communication is exposed when the number of workers is increased, resulting in more scalable communication and corresponding speedups, peaking at 27 with the copy core approach. The master node only needs

to send a set of addresses which reduces the communication time for the master so it can distribute tasks faster. The copy core approach performs slightly better then remapping with more clients as the master node still becomes a bottleneck on receiving back the result of the computations. Note that computations using the copy core approaches can not be run on all 48 cores as some cores are reserved for the copy tasks.

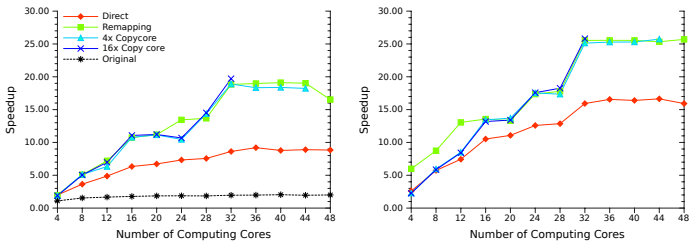


(a) Matrix size 1024×1024. (b) Matrix size 2048×2048.

Figure 6: Matrix multiply with 2 decomposition steps

4) Recursive decomposition over multiple nodes: A solution that decreases the load of a single master node, is to split the recursion steps over multiple nodes. The master node performs one decomposition step and delegates the next decomposition to other nodes. These nodes then perform the second step and distribute the work over even more nodes, using a round robin algorithm that guarantees an as much even distribution as possible. This approach introduces a lot of additional communication, but not much computational overhead as the decomposition on a single node can be done without additional copy operations due to the way we structure our matrices in memory.

The benchmark results are shown in Figure 7, where again the original implementation was unable to run the 2048 configuration. The original and direct approaches clearly benefit from the different communication pattern, while the remapping and copy core approaches perform about the same as with the other communication pattern, peaking at a factor 25 speedup.



(a) Matrix size 1024×1024. (b) Matrix size 2048×2048.

Figure 7: Matrix multiply with distributed decomposition.

V. Conclusion

We have shown our initial results of porting our implementation of the SVP model of concurrency to the Intel SCC. One of the biggest problems was the efficient communication of data; it is difficult to keep all the cores busy and to find a good communication to computation ratio.

We have discussed several approaches on how we improved our communication bottleneck; removing XDR encoding, remapping and copying data directly at the receiving core, and employing our copy core techniques. The latter showed a two orders of magnitude improvement in throughput, and has the potential to scale up by employing multiple copy cores. However, the matrix multiply benchmarks that we used were not able to effectively use the large bandwidth provided by the copy core techniques compared to the remapping approach.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108–109, February 2010.

[2] C. R. Jesshope, "A model for the design and programming of multi-cores," *Advances in Parallel Computing*, vol. High Performance Computing and Grids in Action, no. 16, pp. 37–55, 2008.

[3] K. Bousias, N. Hasasneh, and C. Jesshope, "Instruction level parallelism through microthreading—a scalable approach to chip multiprocessors," *Comput. J.*, vol. 49, pp. 211–233, March 2006.

[4] J. Sykora, L. Kafka, M. Danek, and L. Kohout, "Analysis of execution efficiency in the microthreaded processor UTLEON3," in *Proceedings of the 2011 Conference on Architecture of Computing Systems (ARCS 2011)*, vol. 6566 of *Lecture Notes in Computer Science*, pp. 110–121, Springer, 2011.

[5] M. W. van Tol and J. Koivisto, "Extending and implementing the self-adaptive virtual processor for distributed memory architectures," *CoRR*, vol. abs/1104.3876, April 2011.

[6] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope, "Efficient memory copy operations on the 48-core intel scc processor," in *3rd Many-core Applications Research Community (MARC) Symposium*, KIT Scientific Publishing, September 2011.

[7] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on Intel's Single-chip Cloud Computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.

[8] 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 International Conference on High Performance Computing and Simulation (HPCS2011) – to appear, Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, (Istanbul, Turkey), July 2011.

[9] Intel Labs, *SCC External Architecture Specification*, revision 1.1 ed., November 2010.

[10] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ISCA '92: Proc. of the 19th annual Int. Symp. on Computer architecture*, (New York, NY), pp. 256–266, ACM, 1992.

[11] C. R. Jesshope, M. Lankamp, and L. Zhang, "Evaluating CMPs and their memory architecture," in *Proc. Architecture of Computing Systems* (M. Berekovic, C. Muller-Schoer, C. Hochberger, and S. Wong, eds.), pp. 246–257, 2009.

[12] T. D. Vu and C. R. Jesshope, "Formalizing sane virtual processor in thread algebra," in *ICFEM*, pp. 345–365, 2007.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, (New York, NY, USA), pp. 15–26, ACM, 1990.

[14] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "Cellss: a programming model for the cell be architecture," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 86, ACM, 2006.

[15] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), p. 83, ACM, 2006.

[16] M. Eisler, "XDR: External Data Representation Standard." RFC 4506 (Standard), May 2006.

Caching Strategies and Access Path Optimizations for a Distributed Runtime System in SCC Clusters

Björn Saballus, Stephan-Alexander Posselt and Thomas Fuhrmann
Technische Universität München
Boltzmannstrasse 3, 85748 Garching/Munich, Germany
{bjoern.saballus|stephan.posselt|thomas.fuhrmann}@tum.de

***Abstract*—In the context of our J-Cell project, we propose a novel approach for locating objects in a fully decentralized system. In our system migrations are caused by processor nodes modifying shared objects. Hence our system allows for a high migration rate. Migrating objects leave behind a chain of proxies that forward access requests until they eventually reach the object.**

In this paper, we study the effect of caching and a mechanism to reduce the proxy chain length. We show that caching works well with red-black trees, but not with AVL trees. We also show that the benefits of caching are not degraded when the cache size is increased beyond the required minimum. Furthermore, we analytically derive a formula to determine the optimal parameters for our proxy chain reduction algorithm; and we demonstrate that our algorithm works as expected.

***Index Terms*—Distributed Runtime Environment, Cache Strategy, Distributed Object Access, SCC Cluster**

I. Introduction

The physical limits to sequential processing performance force computer architectures to become increasingly parallel. Well known examples include general purpose graphics processing units such as Intel's Larrabee, heterogeneous many-core processors with explicit memory management such as the IBM Cell BE, and homogeneous many-core processors with non-coherent caches such as Intel's Single-chip Cloud Computer (SCC). This trend challenges software developers who have to work with legacy code that was originally targeted at uniprocessors. Also, most developers seem to be well accustomed to sequential algorithms, but they have a hard time programming with concurrency in mind, especially when the algorithms need to scale to a vast number of cores.

We believe that concepts like distributed shared memory (DSM) and transactional memory (TM) can help to alleviate these problems: DSM keeps up the traditional system model and is better suited for irregular applications than message passing; TM [1], [2] better hides latency than locking, and it is easier to implement correctly [3].

Our research project J-Cell[1] follows this line of argumentation. The J-Cell runtime system provides a *single system image* (SSI) [4] across all cores and all processors in a compute cluster. Thereby, it hides the heterogeneous and distributed nature of clusters of many-core processors from the software developer.

[1]J-Cell is supported by the German Ministry of Education and Research under grant number 01IH08011. We are also grateful to Intel for granting us access to their SCC platform.

We described our system in more detail in [5], where the focus is on the decentralized access to mobile objects on SCC clusters. The SCC is an experimental 48-core processor [6] that Intel Labs have created as a "concept vehicle" for many-core software research. We have shown that the most suitable approach for locating and retrieving mobile objects in a decentralized scenario is the use of forwarding proxies, and not the use of update messages that are sent to the referencing objects.

This paper extends this work and gives an overview of our current work on path optimization algorithms for long chains of forwarding proxies. Moreover, we examine the influence of object caching on distributed applications, which we examine with regard to our TM algorithm. We simulate a distributed environment where an application uses a tree data structure. We compare an AVL and a Red-Black tree to study the effect of different degrees of contention on the elements of a shared data structure.

The remainder of the paper is organized as follows. Section II briefly introduces our distributed runtime environment. Section III describes our proxy forwarding approach and our optimizations that speed up access to migrating objects. Section IV illustrates our caching approach, and Section V evaluates our approach with the help of simulations. Section VI gives an overview over the related work, before we draw our conclusion in Section VII.

II. Runtime Environment/System Overview

The J-Cell runtime environment completely eliminates all centralized components: Its memory access sub-system references objects that can be scattered across all participating nodes; a fully decentralized object location and retrieval algorithm enables the location-transparent access to the objects [7]; a multi-version software transactional memory system (DecentSTM) handles parallel access to these objects with the help of a fully decentralized consensus protocol [8]. To deal with node failures, the runtime uses a fully decentralized recovery mechanism, which uses outdated object versions as implicit checkpoints [9]. Thereby, when a processor or a memory chip fails, the runtime can automatically roll-back to the most recent consistent version. On each core in a cluster of many-core processors runs one instance of the runtime system; and together, all instances collaborate to provide the SSI.

If the runtime system is used in form of our C/C++ library, the application has to call the appropriate functions that create

and access objects. In particular, these function calls indicate when pointers are de-referenced, so that the runtime can ensure that a copy of the respective object is available in the respective processor's memory.

If the runtime system is used with Java applications, it directly interfaces to our distributed virtual machine DecentVM [10], so that applications do not need to be modified. The VM automatically partitions programs into a sequence of transactions during execution; it does so by taking accesses to monitors and volatile variables as boundaries between the transactions [11]. Optionally, the application programmer may annotate the source code to indicate which code blocks shall form transactions.

The memory model of the J-Cell runtime system is derived from a *non-uniform memory access* (NUMA) architecture. It distinguishes between *logical* (private or global) and *physical* (private, local, or remote) memory. All objects that reside in global memory are globally accessible from all nodes. All objects that reside in private memory are either local variables, or private copies of globally accessible objects. Logically, the application operates on mutable objects, while for the runtime system each object is represented by a list of immutable object versions. Whenever a node writes to an object, a new object version is created, and prepended to the corresponding list of object versions during a successful commit. Thus, each object consists of a current head version and a chain of multiple, outdated versions (see Figure 1).

On each core, threads can only operate on object versions that reside in that core's private memory. Thus, all operations on global objects require that the runtime environment first retrieves a copy of the head version of the object. When the transaction has finished, the DecentSTM algorithm propagates all changes back into the global memory. Thus, the DecentSTM algorithm mediates between private and global memory. We describe this memory model in more detail in [5].

III. Chains of Forwarding Proxies

With the reactive location update protocol presented in [5], migrating objects leave behind a trail of proxies. In our system, an object typically migrates when another thread creates a new version of that object. In that case, DecentSTM implicitly creates a forward and backward reference between the object's head version and its previous version.

A proxy does not know by itself if it is still referenced by other objects and thus, can not be deleted. However, if it is never deleted, the length of a proxy chain is unbounded. So, a *Distributed Garbage Collector (DGC)* periodically removes all proxies: It follows all references and all proxy forwarding pointers to all reachable objects. When the DGC reaches the referenced object, the DGC implicitly updates all outdated object locations and marks all visited proxies as garbage. Afterwards, all proxies are un-referenced and marked for deletion, and the next DGC run can remove them.

The other mechanism that updates the object location of a migrated object is an ordinary access operation to the migrated object. The first access request message that is sent to an outdated object location traverses the chain of proxies, and eventually reaches the head version. The node where the head version resides sends its response or acknowledgment directly back to the requesting node. Because the response implicitly contains the current location of the object, the requesting node can cache this new location and use it for all subsequent accesses.

Each proxy p_i in a chain of proxies holds a forwarding pointer to the next proxy p_{i+1} in the proxy chain. Additionally, each proxy p_{i+1} also knows its predecessor, and thus the node that stores proxy p_i. The reactive location update protocol does not need these backward pointers, but the *DecentSTM* protocol has to be able to walk the history of an object.

The work presented in this paper uses these backward pointers for a novel proxy chain optimization protocol that propagates updated object location information backwards along the reference chain, from the target to the sources. Thereby, the update propagation depth k decreases the number of hops along a proxy chain of length ℓ from $\ell+1$ to $\lceil \frac{\ell}{k} \rceil + 1$, cf. Figure 1.

Note that our proxy forwarding algorithm inherently cuts out loops in a proxy chain whenever an object returns to a previously visited node, because the next migration overwrites the previous forwarding pointer with the new one.

The optimal propagation depth k depends on the access characteristics of the corresponding object. Namely, on the ratio of the number of read accesses R and the number of write accesses W.

To update the forwarding information of all previous proxies, the system needs to send ℓ messages, one for each step along the chain. If only k update messages are sent, each access request for an object requires $\lceil \frac{\ell}{k} \rceil$ hops along the proxy chain. Here, ℓ is the average number of object migrations that took place since the reference to the migrated object was created. Therefore, ℓ describes how deeply in the proxy chain the accessing node begins its way up to the current location of the object, cf. Figure 1. To optimize the costs of an object access, the number of messages per write plus the number of messages per read, or $kW + \frac{\ell}{k}R$, must be minimal, i. e. $W - R\ell/k^2 = 0$ or

$$k = \sqrt{\frac{R \cdot \ell}{W}} \tag{1}$$

As a result of the update propagation, each proxy p_i holds k forwarding pointers that point to the successive proxies p_{i+x}, with $1 \leq x \leq k$, within the proxy chain, where the special case of $k = 1$ represents a proxy chain with only 1-hop forwarding pointers.

Figure 1 gives an example for an update message propagation depth of $k = 2$. Note that the length of the proxy chain between referencing object and referenced object depends on the entry point into the proxy chain. From object 9 the total length ℓ of the proxy chain is $\ell = 2$, while from object 14 the total length is $\ell = 4$. The propagation depth of $k = 2$ decreases the number of message hops for object 9 and 14 from 3, respectively 5, hops to 2, respectively 3 hops.

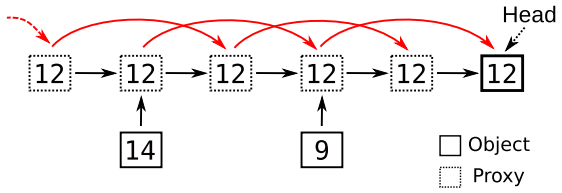


Fig. 1. Update propagation with depth $k = 2$

Besides decreasing the number of hops, these additional forwarding pointers add additional paths along which the referenced object can be accessed, and thus harden the proxy chain against node failures. With a propagation depth of k, a proxy chain can tolerate the loss of at least $k - 1$ proxies without getting fragmented. When e. g. one of the proxies p_i between object 14 and 12 fails, one of the $k - 1$ remaining forwarding pointers of the previous proxy p_{i-1} can be used.

In the same way, the system can harden object references, and store not only the latest object location, but additionally $k - 1$ previous locations. Then, the response message from object 12 does not only contain the current location of the home node, but also the locations of the last $k - 1$ proxies if there are that many.

In the DecentSTM context, this additional information about previous proxies (object versions) allows transactions to work with older object versions. This information is also required by the redundancy and recovery mechanism to restore all necessary transactions that have been lost due to a node failure.

IV. Object Version Caching

DecentSTM can execute transactions optimistically without first checking that a cached version actually is the most recent version. If it is or if using an outdated version does not lead to an inconsistency during the commit phase, the optimistic approach speeds up execution. If however the transaction has to roll back and restart using current versions, this approach wastes time. Thus, we now explore the probability that the optimistic approach succeeds. We assume a cache using the *least-recently-used (LRU)* policy. We distinguish three cases when accessing an object:

- *cache miss*: The accessed object is not cached, and the thread has to stall until the object's head version has been retrieved.
- *good cache hit*: The accessed object is cached, and the cached version is the current head version of the object. The thread continues execution immediately (without knowing that it works with the head version). It has a good chance to succeed when committing.
- *bad cache hit*: The accessed object is cached, but the cached version is outdated. The thread continues execution immediately (because it cannot know that the version is outdated). It is likely to fail when committing.

V. Evaluation

We simulated the described system with two example applications: a *Red-Black (RB) tree* and an *AVL tree*. The AVL tree is a balanced binary search tree with the invariant that the height of any two branches in the tree differ by at most 1. To guarantee this invariant, each insert or delete operation may have to re-balance the whole tree, starting from the root.

In contrast to the AVL tree, a RB tree has weaker balancing constraints. Among others, one invariant of the RB tree is that the longest branch of the tree can be at most twice as long as the shortest branch of the tree. Re-balancing operations start at the node in the tree where the element has been inserted or deleted, and continue towards the root until the tree invariants are restored. As a result, the balancing operations of an RB tree modify less objects than those of the AVL tree.

Our simulator consists of a given number of nodes, where each node executes one thread that operates on the given data structure. Each thread randomly inserts and deletes elements into/from the tree. Each node is equipped with a local cache of a given size that implements the *least-recently-used (LRU)* policy.

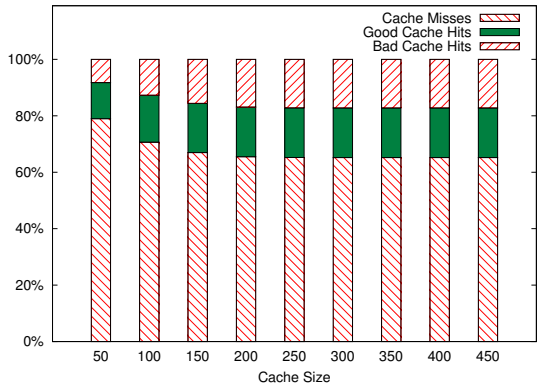
A single simulation run executes one million transactions that manipulate the given tree. Each transaction draws two random numbers between 1 and 10 000, which are used as object identifiers r_1 and r_2. Each transaction first inserts the object r_1 if the tree does not yet contain this element, and re-balances the tree if necessary. Then, it searches for object r_2 in the tree, deletes it if it was present, and again re-balances the tree.

Note that we are not concerned with an underlying routing algorithm or a particular network topology. We only consider proxy forwards.

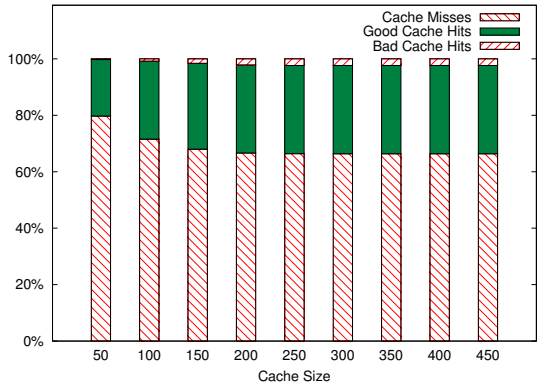
We ran simulations with 1 000 nodes and various cache sizes, ranging from 50 to 450 objects. The numbers shown are averages over 100 runs each. Each simulation run made in total 25 114 763 initial cache accesses for the AVL tree and 23 998 597 initial cache accesses for the RB tree. Note that the total number of initial cache accesses is equal for all cache sizes. Here, "initial access" denotes the first access to an object within a transaction. To prevent the transactions' memory snapshot from becoming inconsistent, our TM system requires that subsequent accesses use the same cached object version.

Figure 2 shows the percentage of *cache misses*, *good cache hits*, and *bad cache hits* for the different cache sizes. As expected the good cache hit rate is higher for the RB tree than the AVL tree. It is 12.8% for the AVL tree and 20.0% for the RB tree with a cache size of 50 objects, and 17.6% and 31.3% for a cache with 450 objects. Furthermore, the bad cache hit rate is lower. It is 8.23% for the AVL tree and 0.26% for the RB tree with a cache size of 50 objects, and 17.23% and 2.36% for a cache with 450 objects. Compared to the total cache hits, the AVL tree has 39.16% outdated objects and the RB tree 1.28% outdated objects in a cache with 50 objects, and 49.51% and 7.02% for a cache size of 450 objects. We can see that for the AVL tree the good cache hit rates set in at a cache size of 100, whereas for the RB tree the cache size should be 150. Larger caches do not (significantly) increase the system's performance.

Figure 3 shows the probability that a cache hit is good as a function of the number of intermediate transactions, i.e. the

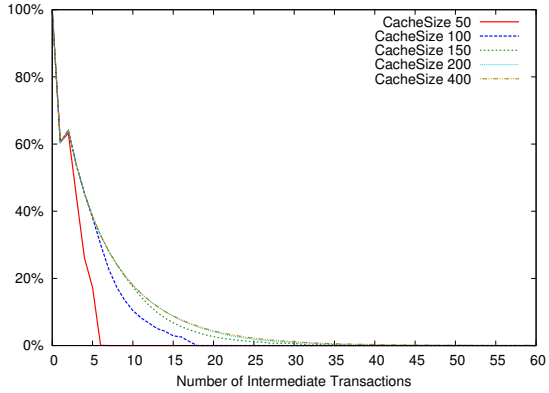


(a) Cache Accesses for AVL Tree

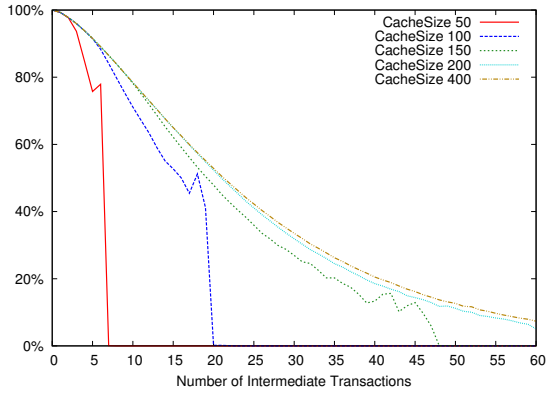


(b) Cache Accesses for RB Tree

Fig. 2. RB and AVL Tree Cache Accesses in %.



(a) AVL Tree: Good Hit/Total Hit probability.



(b) RB Tree: Good Hit/Total Hit probability.

Fig. 3. Probability of Good Hit/Total Hit after x intermediate Transactions.

number of transactions that the *accessing* node has processed since the cache entry has been used previously. The smaller the cache the more quickly the good cache hit rate drops. Just before the probability reaches zero, the number of total hits is so small that the probability becomes erratic. Therefore, we removed those data points from the plots for which the number of total hits is less than 0.3 hits on average.

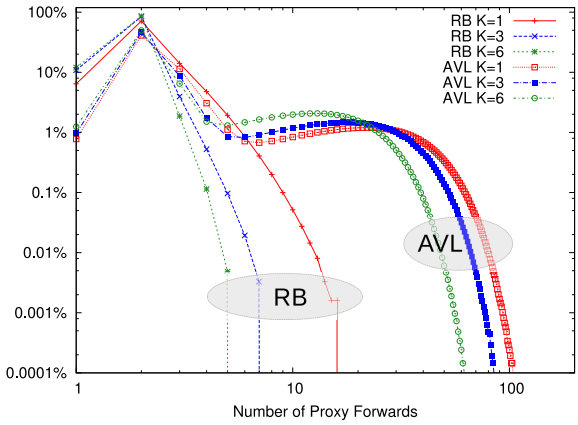
Figure 3 confirms our finding from Figure 2 that large caches are not worthwhile. Even though a large cache can increase the overall number of cache hits, it also decreases the percentage of good cache hits. (Since the number of total hits decreases with the number of intermediate transactions, this effect is not seen in Figure 2.)

Comparing the AVL tree with the RB tree, we again see that the RB tree has a much better performance in our system. In an AVL tree, the probability of a good cache hit, and thus the probability that an object was not modified after two intermediate transactions, is only about 60%. In an RB tree, the probability of a good cache hit is 97.7% after two intermediate transactions with a cache size of 50 objects. Overall, we conclude that 100 objects is the optimal cache size for the AVL tree, whereas it is 150 objects for the RB tree.

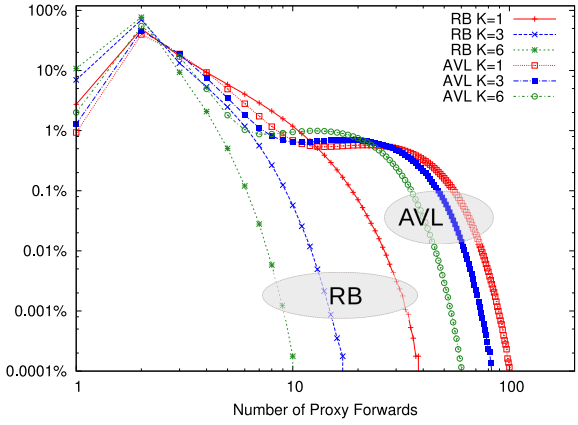
Upon a cache miss, the thread must stall until it has retrieved the respective object. However, the proxy chain that leads to the current version is the longer the more transactions have created new versions of the object, and thus proxies. The longer the proxy chain that must be traversed, the longer a thread has to stall before the request completes. Figure 4 shows the number of proxy forwards that are needed to retrieve the head version of an object. (Note that the figures are plotted with a double-logarithmic scale for the two cache sizes of 50 and 450 objects.) To create the figure, we ran our simulation with various update propagation depths k. (We only show $k = 1, 3, 6$ because the plots for $k > 6$ are not significantly different.) In addition, Table I shows the length of the traversed proxy chain, i.e. the probability that the object's head version was reached after the first, second, third, and fifth proxy forward.

Both AVL tree and RB tree have their maximum at a proxy chain length of 2 hops, but for the AVL tree, this maximum does not peak as highly as for the RB tree. Especially, the AVL tree has a non-negligible probability to produce very long proxy chains. For the RB tree, the maximal proxy chain length we found varies between 5 ($k = 6$) and 23 ($k = 1$) for a cache with 50 objects, and between 14 and 56 for a cache size of 450 objects. The increase in the proxy chain length is a result of the increased number of outdated objects in the cache: The larger the cache, the more objects are cached that remain in the cache for a longer time period before they are evicted by the LRU policy. And the longer the time an object stays in the cache, the more new object versions are created on other nodes and the longer the proxy chain grows.

For the AVL tree, the proxy chains in our simulations grow to 78 hops (cache size 50) and 177 hops (cache size 450).



(a) AVL and RB Tree, Cache Size: 50.



(b) AVL and RB Tree, Cache Size: 450.

Fig. 4. Number of Proxy Forwards per Cache Miss.

	Cache Size	k	1-hop	2-hop	3-hop	5-hop
AVL	50	1	0.49	28.38	16.28	3.22
		3	0.96	45.99	8.73	0.85
		6	1.24	49.64	6.47	1.31
	450	1	0.53	26.22	19.76	7.42
		3	1.29	46.13	18.90	3.47
		6	2.00	54.86	16.91	1.79
RB	50	1	6.43	71.13	14.08	1.92
		3	10.83	84.56	3.95	0.09
		6	11.99	86.01	1.87	0.01
	450	1	2.74	49.18	17.40	5.88
		3	6.98	69.70	13.25	2.47
		6	10.79	77.15	9.31	0.50

TABLE I
PROXY FORWARDS FOR AVL AND RB TREE IN %.

The reason for these high values are the frequent tree rotations that are necessary to balance the tree. Each such rotation creates new object versions, and thereby increases the proxy chain length. These frequent rotations also explain the quickly decreasing good cache hit probability in Figure 3 and the plateau at about 1% in the AVL plot. Only the finite number of nodes in our simulated system keeps the proxy chains from growing even larger. This is because the longer a chain, the higher the probability that adding a proxy introduces a loop, which our algorithm then cuts out automatically.

In Section III, we introduced the update propagation mechanism to reduce the number of proxy forwards. The optimal propagation depth k is determined by the average number of newly created head versions W and the number of cache accesses R. Both values can be determined at run time. In our simulation, we thereby arrive at an optimal propagation depth k_{sim}. Additionally, we also measured the average proxy chain length ℓ_k. With these numbers we can compute the average message cost c as

$$c = W \cdot k_{sim} + R \cdot \ell_k \tag{2}$$

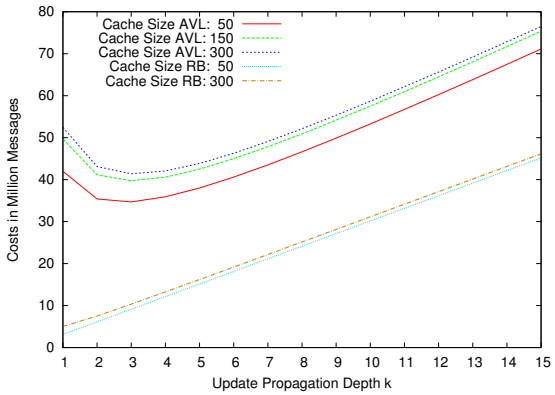


Fig. 5. AVL Tree Message Costs depending on k_{sim}, in Million Messages.

	Cache Size	W	R	$l_{k=1}$	k_{opt}
AVL	50	3 955 611	2 066 916	18.40	3.10
	450	3 955 611	4 326 585	11.21	3.50
RB	50	3 004 185	62 151	2.31	0.22
	450	3 004 185	566 184	3.59	0.82

TABLE II
THEORETICAL OPTIMAL PROPAGATION DEPTH k_{opt}.

Figure 5 shows the results our simulations: The two linear curves at the bottom show the results obtained from the RB tree; the three U-shaped curves are from the AVL tree. With a cache size of 300 objects, the message costs are already close to the maximal message costs of both data structures. A larger cache does not increase the message costs any more, because a larger cache would only contain objects that will never be used again. For the AVL tree, a smaller cache slightly reduces the message cost, because it leads to shorter proxy chains. In accordance with our analytical result in equation (1), we see that the minima of the message costs are at $k = 3$ for the AVL tree, and at $k = 1$ for the RB tree. Also, compare the experimental results of the graphs with Table II, where we computed k_{opt} with Equation (1) for cache sizes of 50 and 450 objects.

VI. RELATED WORK

Fowler [12] was the first to introduce forwarding addresses to locate mobile objects in distributed systems. By now, the use of forwarding pointers is a common approach to ensure the reachability of migrating objects, e. g. [13]–[17].

All these approaches leave a proxy behind whenever an object or mobile agent moves from one node to another.

However, to our best knowledge, none of these approaches deals with the removal of old proxies, and none updates the forwarding pointers within the proxy chain.

Our approach uses multiple forwarding pointers and a backwards pointer to update these forwarding pointers, and is closest to the approaches of Moreau and Ribbens [18] and Fowler [12].

Moreau and Ribbens developed a middleware for mobile agents that uses chains of proxies as well. The authors describe an *Eager Acknowledgments* mechanism that propagates the new location of a mobile agent to all previous proxies. Thus, their approach can be seen as a special case of our approach with $k = \infty$. However, Moreau and Ribbens do not consider the access characteristics of a mobile agent at all.

Unlike us, Fowler always update all proxies in the proxy chain after each object access that traversed a chain of proxies. This approach can be seen as a special case of our approach with $k = \infty$, as well. Furthermore, Fowler updates the proxies upon object access, whereas we update the proxies upon object migration, i.e. as part of the DecentSTM commit protocol. Thereby, we avoid that the first access after a number of migrations has to traverse the whole chain of proxies, and we avoid the case where multiple nodes traverse and try to update the chain of proxies at the same time.

VII. Conclusion

The J-Cell runtime system stores application-level objects in form of immutable object versions. A processor node that modifies an object creates a new version; typically, the newly created version resides on the node that has created that version.

In this paper, we have studied two problems that arise in this system: Cached object versions might have become outdated, and a referenced remote object version might not be the object's head version any more.

To study the effect that different cache sizes have on the performance of our system, we simulated a 1 000 node cluster where 1 000 threads concurrently access a shared data structure. Using a Red-Black tree and an AVL tree as examples, we showed that our speculative execution approach is viable when there is not too much contention. In particular, we found that with an RB tree, only 5% of the cached objects in a cache with 150 objects are bad in the sense that the cached version has become outdated and is thus likely to cause a roll-back when used.

Our findings also show that the choice of data structure has a large impact on the performance of the application. We found the AVL tree to not be suited for our envisioned system, because the tree balancing operations modify a large portion of the tree. A node with 100 cached objects, for example, has only a 60% chance that a cached object is still up to date.

Besides analyzing the cache efficiency, we also studied the number of proxy forwards that our system requires. Such proxies are created when a processor node creates a new object version. Upon access, a referencing node must follow the proxy chain to retrieve the object's head version.

We found that with the RB tree, proxy chains are typically short, whereas the AVL tree can lead to very long proxy chains. For the latter case, we thus propose to use an update propagation mechanism that creates shortcuts in the proxy chain. Our simulations show that our analytically derived formula for the recommended update depth minimizes the system's message overhead.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. of the 20th Annual Int'l Symposium on Computer Architecture(ISCA'93)*, May 16 – 19, 1993, pp. 289–300.

[2] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. of the 14th Annual ACM Symposium on Principles of DistributedComputing (PODC'95)*, Aug. 20 – 23, 1995, pp. 204–213.

[3] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" *ACM SIGPLAN Notices - PPoPP '10*, vol. 45, pp. 47–56, Jan. 2010.

[4] R. Buyya, T. Cortes, and H. Jin, "Single system image," *International Journal of High Performance Computing Applications*, vol. 15, no. 2, pp. 124–135, 2001.

[5] B. Saballus, S.-A. Posselt, and T. Fuhrmann, "A scalable and robust runtime environment for SCC clusters," in *Proc. of the 3rd MARC Symposium*, Jul. 05 – 06, 2011.

[6] Intel Corporation, "The SCC platform overview, rev. 0.7," May 2010.

[7] B. Saballus, S.-A. Posselt, and T. Fuhrmann, "Brief announcement: Fault-tolerant object location in large compute clusters," in *Proc. of the 13th Int'l Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, Oct. 10 – 12, 2011.

[8] A. Bieniusa and T. Fuhrmann, "Consistency in hindsight, a fully decentralized STM algorithm," in *Proc. of the IEEE Int. Symposium on Parallel Distributed Processing (IPDPS'10)*, Atlanta, Georgia, USA, Apr. 19 – 23, 2010.

[9] S.-A. Posselt, "Design of a reliable, fully decentralized software transactional memory protocol," Diploma thesis, Technische Universität München, Munich, Germany, Aug. 2010.

[10] A. Bieniusa, J. Eickhold, and T. Fuhrmann, "The architecture of the DecentVM – towards a decentralized virtual machine for many-core computing," in *Proc. of the 4th Workshop on Virtual Machines and Intermediate Languages (VMIL'10)*, Reno, Nevada, USA, Oct. 17, 2010.

[11] A. Bieniusa and T. Fuhrmann, "Lifting the barriers - reducing latencies with transparent transactional memory," in *Proc. of the 13th Int'l Conf. on Distributed Computing and Networking (ICDCN'12)*, Jan. 03 – 06, 2012.

[12] R. J. Fowler, "The complexity of using forwarding addresses for decentralized object finding," in *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*, Aug. 11 – 13, 1986, pp. 108–120.

[13] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

[14] B. Steensgaard and E. Jul, "Object and native code thread mobility among heterogeneous computers," in *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP'95)*, Dec. 3 – 6, 1995, pp. 68–77.

[15] M. Philippsen and M. Zenger, "JavaParty transparent remote objects in Java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1225–1242, Nov. 1997.

[16] W. Fang, C.-L. Wang, and F. C. M. Lau, "On the design of global object space for efficient multi-threading Java computing on clusters," *Parallel Computing*, vol. 29, no. 11-12, pp. 1563–1587, 2003.

[17] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer, "Design and implementation of a single system image operating system for ad hoc networks," in *Proc. of the 3rd Int'l Conf. on Mobile Systems, Applications, and Services (MobiSys'05)*, 2005, pp. 149–162.

[18] L. Moreau and D. Ribbens, "Mobile objects in Java," *Scientific Programming*, vol. 10, no. 1, pp. 91–100, 2002.

Flexible Sharing and Replication Mechanisms for Hybrid Memory Architectures

:

Thomas Prescher, Randolf Rotta, Jörg Nolte
{tpresche,rrotta,jon}@informatik.tu-cottbus.de

***Abstract*—The SCC (Single-Chip Cloud Computer) is an experimental 48-core concept vehicle created by Intel Labs and does, deliberately, not provide hardware-implemented cache coherence. It can be treated like a distributed memory system by implementing data replication and consistency control based on message passing. However, the SCC still is a shared memory system with shared access to parts of the memory. Thus, replicating data for every core wastes memory and is inefficient as well, because most replica updates go from and to the same memory device. This paper presents a framework for memory efficient sharing on such distributed systems with shared memory subsystems. It is adaptable in respect to the underlying memory architecture (with and without hardware cache coherence) as well as the employed sharing models (e.g. central instance, replicas with various consistency models, and hybrids of both). This is achieved by dividing the replica management into storage containers (for shared data on each memory device) and control containers (for consistency and cache control on each core). The framework's design shows that it is possible to combine different sharing and communication paradigms to exploit hybrid many-core architectures up to clusters of SCCs.**

***Index Terms*—many-core, cache coherence, distributed shared memory, C++**

I. Introduction

SOME of the already existing many-core chips are the 100-core Tilera [1], the IBM Cyclops64 with 160 thread engines [2], and the 48-core Intel Single-Chip Cloud Computer (SCC) [3]. The SCC is an experimental *concept vehicle* created by Intel Labs as a platform for many-core software research and does, deliberately, not provide hardware-implemented cache coherence [4]. Soon, it should be possible to integrate even more cores into a processor [5], but cache coherent shared memory across hundreds of cores faces significant scaling issues compared to message passing approaches [6].

The SCC is a hybrid system that combines aspects of a distributed memory system (low latency message passing for small messages) with aspects of a shared memory system (access to shared memory). Thus, replicating large amounts of data for every core wastes memory: Most of the replicas would be located on the same memory device, while several cores could access a single shared replica on each memory device. It is inefficient as well, because during updates the data is passed from main memory through the caches of the communicating cores and is written back to the very same memory device where it originally resided, while large portions of the involved caches are evicted. On the SCC, at most one replica per memory controller is sufficient to achieve optimal access bandwidth and the necessary cache control can be achieved by extending the replica management code. This approach works as well on other systems with hardware cache coherence by just skipping the manual cache flushing. Further, in the presence of shared caches, one replica per shared cache reduces the competition for space in that cache.

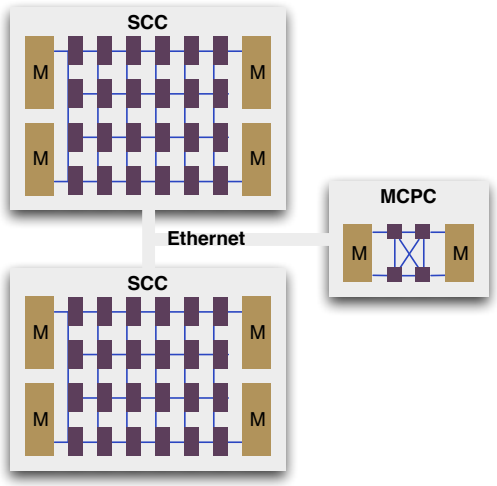


Fig. 1. A distributed system with hybrid memory consisting of two 48-core SCCs with non-coherent and one 4-core management computer with cache-coherent shared memory.

Consider a system like in Figure 1 performing N-particle force simulations based on the popular Barnes-Hut algorithm [7]. In each simulation step a tree is created and then traversed for each particle. The traversal phase can be easily parallelized on a large number of worker cores. After the forces on a particle are calculated, the result is written into a force vector. Because the workers do not modify the tree, its data can be shared by all workers. The force vector is never read by the workers and, thus, the written data can be sent to a central storage in the background.

This paper presents MESH, a framework for Memory Efficient SHaring, to manage shared objects like the above-mentioned Barnes-Hut trees and particle force vectors. Function shipping (based on small messages) is used as fundamental means of communication. On top of this, a Distributed Shared Memory is implemented with configurable sharing spaces behind a unified interface. Inside its implementation, the concept of storage containers and control containers is introduced. The proposed storage containers provide access to the heterogeneous memory architecture through a homogeneous interface and are flexible enough to support even clusters of multiple SCC processors mixed with cache-coherent multi-

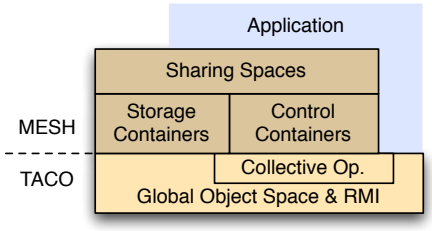


Fig. 2. Software architecture for the Memory Efficient SHaring.

TABLE I
REMOTE METHOD INVOCATION MECHANISMS

	synchronous	deferred synchronous	asynchronous
w/o result	`call(f)`	`call(f, future)`	`apply(f)`
with result	`invoke(f)`	`apply(f, future)`	—

core processors as suggested in Figure 1. The control containers simplify the implementation of consistency protocols, cache control, and critical sections.

Figure 2 summarizes the architecture of the MESH framework. The next section summarizes briefly the *Global Object Space* and Remote Method Invocation (RMI) mechanisms as the most basic level of sharing, that is communication with a resource by knowing where it is. Section III presents our *Sharing Spaces*, describes their interface and some exemplary sharing models. Furthermore, it provides implementation notes on the new essential control and storage containers. The paper concludes with preliminary performance estimates, a short discussion of related work, and closing remarks.

II. THE GLOBAL OBJECT SPACE

Using the SCC as a distributed memory system usually means running one process with private memory per core. These processes communicate through a messaging system in order to coordinate their work and transfer data between each other. The basic way to share a resource in this setting is to know on which core it is and how to address it on that core.

The Global Object Space is such an addressing scheme for objects. Global pointers identify the destination core together with the object's address in the core's logical memory—a concept also known as Partitioned Global Address Space (PGAS). The messaging system is used to remotely perform actions on these objects, for example invoking/applying method calls on them. Basically any PGAS framework can be used to implement a Global Object Space and advanced sharing paradigms on top of it. We chose TACO [8], because it provides global object pointers (`ObjectPtr<T>`) as well as flexible remote method invocation mechanisms, without requiring any special compiler support besides standard C++. Like many PGAS frameworks, TACO also relies on one-sided communication: Method invocations are explicitly sent, but their execution is performed implicitly by the framework.

In TACO, new objects can be created in the local and in remote memory and are made accessible from other cores simply by sending an object pointer to them. For remote method invocations, method calls are wrapped with `m2f` into function objects and these are sent to the object's location for execution. The following example creates a remote object and calls some methods on it. Table I summarizes the basic RMI mechanism; more details can be found for instance in [8].

```
ObjectPtr<BHTree> p =
  allocate<BHTree>(coreID)(init-args...);
Node n = p->invoke( m2f(&BHTree::newNode) );
p->apply( m2f(&BHTree::setWeight, n, 5.0) );
```

III. SHARING SPACES

For many applications the Global Object Space alone is not sufficient. For example, all workers in the Barnes-Hut algorithm have to traverse the tree nodes, which involves many short method calls to the tree object. Thus, they would have to communicate in every small traversal step with the one core that created the data. In order to remove the communication latencies the data should be accessed directly over the memory without any remote communication, which implies to replicate the data as sketched in Figure 3. At the same time for other data, the access may be shared while the transmission costs for replication are not worth the effort. For example, a vector for collecting result values does not benefit from replication because the contents are never read by the workers.

This section introduces *Sharing Spaces* to organize this sharing and replication behind a unified interface. The programmer allocates objects in a Sharing Space and then can pass access to this *shared object* to any core by special *sharing pointers*. The space defines the sharing model for the objects created inside it, for example a migration, replication, and consistency strategy. Three basic sharing models are apparent: In a migration space each shared object has just one central instance and all cores communicate with it over RMI, but this instance can be migrated to another core. In a replication space every core has access to a replica of the shared object directly in its memory—even if some cores share the replica over shared memory. In this case the replicas are managed by the sharing space according to a consistency model like, for example, entry consistency [9]. Hybrid sharing spaces are possible as well, where a shared object has just a few replicas and cores communicate with the nearest replica over RMI.

The framework provides unified *access objects* to work on shared objects and these implement three things: They communicate with the object's controller to trigger consistency protocol actions, they acquire an object pointer to their replica, and they can enforce exclusive access for critical sections.

The following subsections describe the programming interface and introduce distributed containers as means to implement cache control and to exploit the sharing capabilities of heterogeneous memory architectures. The last subsection provides implementation details of the aforementioned migration and entry-consistent replication spaces.

A. User Interface

Pointers to shared objects are implemented by a special pointer type and new shared objects are allocated inside a sharing space in order to specify the sharing model. In the

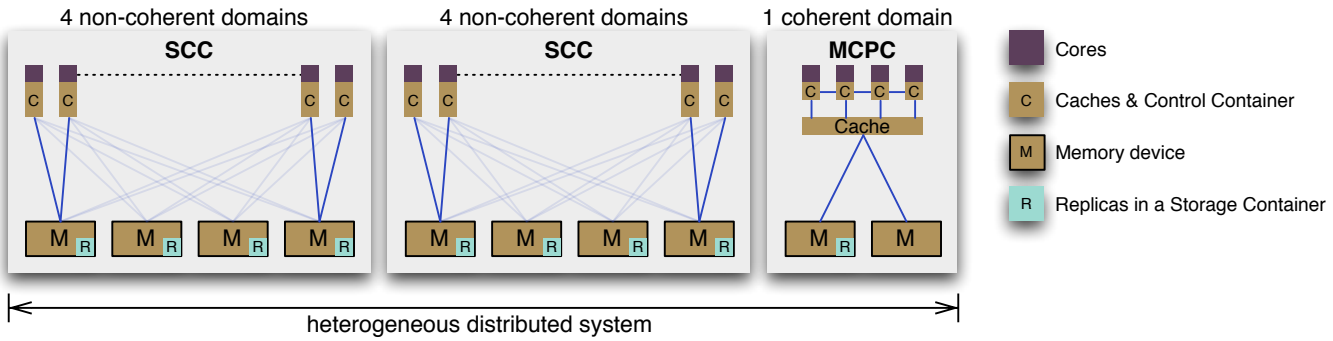


Fig. 3. Memory view to the heterogeneous system from Figure 1. On the SCC groups of 12 cores share a single replica in their nearest memory device. Because the MCPC has a shared last level cache, just one replica is sufficient. Note that all cores are required to have the same Instruction Set Architecture as prerequisite for TACO.

following example two shared objects are created: A Barnes-Hut tree with replication managed by entry consistency and a shared particle vector with a single central instance, initially located at the configured default location.

```
EntryConsistencySpace entrySpace;
Shared<BHTree> tree =
  allocate<BHTree>(entrySpace)(init-args...);
... more allocations

MigrationSpace mSpace(default-core);
Shared<FVector> results =
  allocate<FVector>(mSpace)(init-args...);
... more allocations
```

An instance of the desired sharing space is provided to the allocator. This approach allows to create shared objects of any type in any sharing space, while the sharing pointer hides all implementation details of the space. For example, it is possible for a space to manage replication of its objects collectively instead of separately for each object. Then, unrelated objects can be allocated in separate spaces by using several instances of the sharing space class.

Access to the shared object is granted to other cores by passing them a sharing pointer as argument in a remote method invocation. The object is accessed by creating a temporary access object, which triggers consistency management and can acquire and release locks to protect critical sections. Our first implementation provides three access types: non-exclusive non-modifying (`Reader`), exclusive modifying (`Writer`), and non-exclusive modifying (`MultipleWriter`)—but the framework can be extended by more specific access types. The example below acquires read access to the replicated Barnes-Hut tree and non-exclusive write access to the central force vector. The method call on the tree will be executed on the local replica, while the second call on the vector will be sent to the central instance.

```
{
  Reader<BHTree> t(tree);
  MultipleWriter<FVector> v(results);

  Node n = t->invoke( m2f(&BHTree::nextNode) );
  ... compute forces of particle i ...
  v->apply( m2f(&FVector::put, i, forces) );
}  // destroys access object, releases locks
```

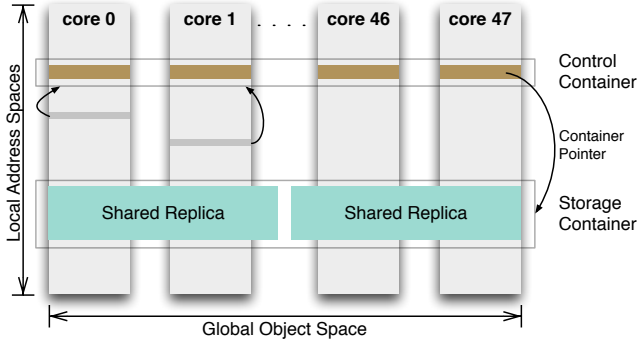


Fig. 4. A Control container and a Storage Container embedded into the Global Object Space. The cores in a domain share one data replica but have individual controllers. Other objects can point to these containers.

After the access object is acquired, the replica is accessed using TACO's method invocations. The invocations may be executed locally or are redirected to another core depending on the sharing model. The critical section is left and locks are released when the access object is destroyed. In the above example this happens automatically in the last line by leaving the scope in which the access objects was created.

B. Distributed Containers as Implementation Vehicle

A container contains several object instances, called members, that are distributed throughout the system, for example with one member on each core. A special pointer class for containers provides access the core-local member, any other member, and all members collectively. These pointers can be passed between cores and immediately allow other cores to access all of the members.

As depicted in Figure 4, two implementations are particularly interesting. Control containers have one member per core and are useful to implement, for instance, the core's cache control. Storage containers reflect the heterogeneous memory architecture by grouping the cores into sharing domains and have a member in each domain. In the following paragraphs, we introduce the interface to control containers and then extend it with the additional aspects of storage containers.

TABLE II
BASIC COLLECTIVE OPERATIONS FOR DISTRIBUTED CONTAINERS.

interface	semantics
`map(f)`	apply `f` asynchronously on each member
`step(f)`	call `f` synchronously on each member
`step(f, future)`	as above, but deferred synchronous
`reduce(f, op)`	invoke `f` and merge results by `op`
`reduce(f, op, future)`	as above, but deferred synchronous

All members of a container are aligned across the individual address spaces of the cores, which means they have the same local address everywhere. Communication with members is greatly simplified by this address alignment: Global object pointers to individual members can be created on demand by combining the local memory address with the target core. Then, members can be accessed through TACO's RMIs with the mechanism summarized in Table I.

Collective operations on the members are provided as well and the operations can be restricted to selected members by providing a boolean predicate. TACO already provides convenient collective operations on groups of objects based on global object pointers, which are summarized in Table II. In the MESH implementation, all containers share a single TACO group for the propagation of their collective operations. Internally, the method invocations and predicates are wrapped in order to operate on the actual container's members instead of the helper group's members. Details of the efficient parallel implementation of collective operations are presented in [10].

In the following example, first a method is called on the local member, followed by a method call on a remote member. Finally, a collective step operation calls the `invalidate()` method on each member of the container.

```
ControlPtr<MC> p = ...;
int i = p.local()->invoke( m2f(&MC::mgrCore) );
p.other(i)->call( m2f(...) );
p.each()->step( m2f(&MC::invalidate) );
```

Control and storage containers share this interface. While control containers have exactly one member per core, storage containers have fewer members that are shared between cores. Their main purpose will be to hold the replica of a shared object. Note, that although their main purpose will be the replication of a shared object, the storage container itself does not know anything about the real object—it just contains images of it [11]. The cores are partitioned into sharing domains with one replica per domain. Each domain has a leading core that is responsible for management tasks and each core has a pointer to his leader, which also allows to check if two cores are in the same domain. Storage containers extend the collective operations interface to operations on all leaders and on all cores of the own domain.

On the SCC, POPSHM is used to allocate physical shared memory and the cores are grouped into four domains according to the four DRAM devices. On other systems, POSIX shared memory is used and the domains are discovered automatically. The access to a replica can go through conventional coherent caches, SCC's non-coherent caches, or circumvent the caches. Special care is necessary on the SCC, because the replica in the memory device can be in a valid or invalid state,

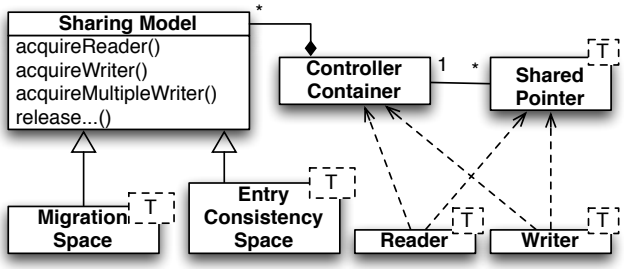


Fig. 5. Interaction and dependencies between spaces, controllers and access objects.

while the cache can contain either no data, some unmodified valid data, some unmodified outdated data, or modified data. These results in six possible states, because a valid replica with modified cache and an invalid replica with valid data in the cache is not possible. Consistency protocols must ensure that two cores in the same domain never have modified cached data for the same replica, because their caches will write back lines at any moment leading to inconsistent memory contents.

Replica data should not be transmitted between cores inside a sharing domain, because explicit flushing parts of the cache is sufficient. Between sharing domains, data transmissions over the network are necessary. Note that this changes the caching states of the sender (to unmodified data in the cache) and receiver (to invalid replica with modified data in the cache). Thus, consistency protocols must perform additional cache control around such transmissions. Inside domains with hardware coherent caches, the cache flushing actions are simply ignored, although the consistency protocols may still track the replica and cache state.

C. Implementing Sharing Spaces

Pointers to replicated objects point to a control container as shown in Figures 4 and 5. The container's members, called *controllers*, implement a common interface to communicate between access objects and the sharing model. Virtual methods are used to hide implementation details about the sharing model. An alternative implementation without virtual methods is possible, but would increase the code complexity on the application side, while the performance win is small as these methods are called only at the construction and destruction of access objects.

Our implementation currently provides two different sharing models (migration spaces and replication spaces with per-object entry consistency). Other models can be integrated by implementing the respective controllers.

The allocator of migration spaces creates a storage container for the shared object and initializes it only on one core, called the central instance. The controllers have a global object pointer to this central instance and implement a shared-exclusive lock for the critical sections. The lock is distributed in a way so non-exclusive acquisitions are performed locally, while exclusive acquisitions perform a collective operation. Migration of the central instance to another core is performed

by acquiring exclusive access, transmitting the replica, updating the controller's object pointer collectively, and releasing the exclusive access. Because no access object can be created during this operation, the migration is not visible to the application

The allocator for entry consistency spaces creates a storage container for the replicas and a control container. The controllers contain the shared-exclusive lock, a pointer to the storage container, the local consistency state (see Section III-B) and a pointer to the core that modified the object last.

When acquiring read access on a core with an invalid replica state, the domain's leader is asked to update the shared replica. This ensures, that the data is transmitted just once from the last writer to this domain: In case the leader knows that the replica is already valid, the transmission is skipped.

Our implementation simplifies the consistency protocol by avoiding three of the possible states: Sooner or later, modified data in a cache has to be written back and, thus, we do it as early as possible to eliminate the modified cache state. Because the writer flushes his cache immediately, any core in the same domain already knows, that his replica is valid. Thus, there is no need to ask the leader for updates in case the last writer was in the same domain. Outdated data in a cache is produced by data transmissions from the last writer to a new writer. This state is eliminated by flushing the cache after such transmissions.

We focused on entry consistency, because it allows to use the shared memory very efficiently. Object-based release consistency can be implemented as well, but would require separate data copies in each modifying access object in order to compute the update messages.

IV. Performance Estimates

Our work-in-progress implementation already runs on cache-coherent systems and is able to trigger the manual cache flushing on the SCC. However, it is in a too early state for meaningful performance benchmarks on the SCC.

We performed LogP parameter [12] benchmarks for TACO's communication protocol. Based on these numbers, it is possible to estimate, for example, the management overhead for replication spaces with entry consistency. Here, we consider a single 48-core SCC and an controller implementation using a distributed shared-exclusive lock, that is non-exclusive locking is done locally, but exclusive locking requires collective operations over all cores in the sharing space.

For a non-exclusive read access, the lock and release operations perform no communication and thus have just a very small overhead. To acquire exclusive write access, the locks have to be acquired by a collective operation over all cores (one `step`). The same operation is also used to invalidate all other replicas and inform the cores about the owner of the new valid replica. Finally, when the write section is left, a collective one-way operation releases all locks (one `map`) Before updating an invalid replica with data from the last writer, it is more efficient to check, whether the replica was already updated in the own domain. This is achieved by asking the domain leader (one `invoke`). Otherwise the data is fetched from the last writer.

TABLE III
Micro-Benchmark results for individual protocol actions.

Action	Local Overhead	min. Completion Time
`step`	2600 cycles	8000 cycles
`invoke`	600 cycles	1500 cycles
`apply`	470 cycles	750 cycles
`map`	2100 cycles	5600 cycles
read acquire	—	—
read + update	**1070** cycles	2250 cycles
write acquire	**4700** cycles	13600 cycles
write + update	**5770** cycles	15850 cycles

Table III summarizes pessimistic estimates for the four possible situations (read vs. write, valid vs. update) based on the 800 MHz core and 1600 MHz mesh frequency configuration. The local overhead counts the send and receive overheads at the core, and the completion time counts the time until the operation is finished on all involved cores. In practice, the operations will take longer, because the involved cores have better work to do than idle polling. However, only the computing overhead really matters as the cores cannot do any useful computations in that time.

The consistency protocol has to transmit data copies over the mesh network and to flush the L1 and L2 caches. Due to current hardware limitations, the latter is implemented in software by a Linux kernel module. Flushing a individual line takes 280 to 580 cycles (clean vs. dirty) plus the system call overhead.[1] Flushing larger memory ranges is up to four times more efficient due to the cache's architecture.

V. Related Work

In the past, frameworks for Distributed Shared Memory (DSM) were unlikely confronted with non-cache-coherent shared memory. The hardware either was entirely distributed memory (clusters), or cache-coherent shared memory (multiple cores), or a mix of both (clusters of multiple cores). The SCC probably is one of the first systems available to researchers that does not provide hardware-implemented cache coherence. Thus, until now, the necessary distinction between control and storage containers, as discussed by this paper, was not immediately apparent.

The Multigrain Shared Memory system [13] is very similar to the presented approach. It implements page-based release consistency with many optimizations, particularly it uses the cache-coherent shared memory of multi-core processors to share replicas. As a pure DSM system, it does not integrate sharing by function shipping.

In [4] a DSM implementation for the SCC is presented. It is page-based and transparent to the applications running on it, while our approach is embedded into the programming language. Coherency domains are used to define which applications/cores have access to the shared data and the employed release consistency creates local copies of accessed pages. In contrast, our entry consistency approach eliminates the local copies and the domains are used to share replicas between nearby cores.

[1] Based on measurements reported by Michiel W. van Tol.

The PSHM [14] implementation of GASNet provides a PGAS optimized for clusters of shared memory machines. The PSHM processes on a cluster node share some cache-coherent memory and use it for faster messaging and of course for direct access (instead of RDMA mechanism) to the nearby partitions of the global address space.

Baumann et al. [6] compared the performance of concurrent access over cache-coherent shared memory versus concurrent access over function shipping to a central server. Their experiments show, that the latter message-based approach scales better on current multi-core systems. The migration space presented in Section III are equivalent to this central server approach, but the presented framework enables also hybrid solution between central servers and replication.

DSM implementations, like Midway [9] and Orca [15], have separate synchronization variables to manage the consistency. Our architecture is not different in this respect. The replicated object pointers internally actually point to the synchronization variable and just through this variable access to the shared data is possible.

We used TACO to supply a global address space and remote method invocation. However, the presented replication spaces can be implemented on top of almost any framework that supplies a global address space and function shipping. For instance the well known GASNet platform [16] could be used as well, but requires considerably more effort.

VI. Conclusions

No single sharing paradigm can serve all use cases equally well and, thus, hybrid paradigms based on function shipping, data replication, and local sharing of replicas are necessary. Many-core systems like the Intel SCC combine aspects of a distributed system with aspects of a (non-cache-coherent) shared memory system. Thus, they support hybrid sharing paradigms very well and applications on such systems also benefit from hybrid sharing compared to conventional implementation approaches.

Conventional sharing approaches either consider cache control without replication, full per-core replication without exploiting shared memory, or communication with central instances. The presented MESH framework is truly hybrid and flexible by exploiting the available shared memory, performing cache control just where necessary, and performing message-based communication where sufficient. By layering various control and storage containers as well as sharing spaces on top of a common global address space, the different sharing paradigms can be used together in applications.

Acknowledgments

We express our gratitude to Sandra Beyer and Robert Zimmermann, who provided us with valuable experience from their Bachelor's thesis projects. Furthermore, we thank Intel for the access to the SCC and the opportunity to contribute to its MARC (Many-core Applications Research Community) program. In particular, we thank Michiel W. van Tol (University of Amsterdam), Werner Haas (Intel Research Braunschweig), and Jan-Arne Sobania (HPI Potsdam) for tremendous insights into SCC's non-coherent memory and implementing the software-based L2 cache flushing.

References

[1] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, 2007.

[2] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao, "A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 64–64.

[3] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE, 2010, pp. 108–109.

[4] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A Case for Software Managed Coherence in Many-core Processors," Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10, 2010.

[5] J. D. Owens, W. J. Dally, R. Ho, D. N. J. Jayasimha, S. W. Keckler, and L.-S. Peh, "Research challenges for on-chip interconnection networks," *IEEE Micro*, vol. 27, pp. 96–108, September 2007.

[6] 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 *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44.

[7] J. Barnes and P. Hut, "A hierarchical O (N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[8] J. Nolte, Y. Ishikawa, and M. Sato, "TACO – Prototyping High-Level Object-Oriented Programming Constructs by Means of Template Based Programming Techniques," *ACM Sigplan, Special Section, Intriguing Technology from OOPSLA*, vol. 36, no. 12, December 2001.

[9] B. Bershad, M. Zekauskas, and W. Sawdon, "The midway distributed shared memory system," in *Compcon Spring'93, Digest of Papers.* IEEE, 1993, pp. 528–537.

[10] J. Nolte, M. Sato, and Y. Ishikawa, "TACO — Exploiting Cluster Networks for High-Level Collective Operations," in *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001), Brisbane, Australia*. IEEE Computer Society Press, May 2001.

[11] Plato and F. Cornford, "Allegory of the cave," in *The republic*. Oxford University Press, 1951.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '93. New York, NY, USA: ACM, 1993, pp. 1–12.

[13] D. Yeung, J. Kubiatowicz, and A. Agarwal, "Multigrain shared memory," *ACM Trans. Comput. Syst.*, vol. 18, pp. 154–196, May 2000.

[14] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick, "Hybrid PGAS runtime support for multicore nodes," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, pp. 3:1–3:10.

[15] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A language for parallel programming of distributed systems," *IEEE Trans. Softw. Eng.*, vol. 18, pp. 190–205, March 1992.

[16] D. Bonachea, "Gasnet specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

Towards Symmetric Multi-Processing Support for Operating Systems on the SCC

:

Jan-Arne Sobania, Peter Tröger and Andreas Polze

***Abstract*—The *Intel Single-Chip Cloud Computer (SCC)* is an experimental many-core system created for research purposes. By default, it is operated as 48-node cluster-on-a-chip with one operating system instance per core.**

In this paper, we analyze the hardware capabilities expected by a standard operating system for symmetric multi-processing support. We discuss how the SCC lacks some of these mandatory capabilities, and present a technique for overcoming the differences through virtualization. Our new Linux hypervisor *RockyVisor* emulates missing SMP capabilities for the SCC hardware, which allows the execution of para-virtualized SMP operating systems on the SCC.

I. Introduction

THE *Single-Chip Cloud Computer (SCC)* is a 48-core experimental processor [1] created by Intel Labs. It is intended to act as a hardware platform for many-core software research on different system levels. Highlights of the SCC architecture are the on-die mesh network for communication between cores and memory controllers, flexible power management and frequency scaling capabilities, and a reconfigurable shared memory hardware.

Due to differences in peripheral device and interrupt handling, standard unmodified operating system kernels do not run on the SCC, even on a single processor core. In [2], we have analyzed necessary modifications for the Linux 2.6 kernel to support its execution on a single SCC core. To utilize the entire chip, it is still needed to run 48 independent instances of the operating system and use the resources as cluster execution environment. With such a setup, applications need to rely on a distributed parallel programming model, e.g. with the *Message Passing Interface (MPI)* or the SCC-specific RCCE [3] messaging facility. There is no default support for parallel shared-memory based applications spanning multiple SCC cores.

In this paper, we present a novel approach for not only running parallel messaging-based *applications*, but also an entire SMP *operating system* on the SCC. This would allow concurrent shared-memory applications to use the SCC hardware resources as a whole. Our approach relies on a new hypervisor with specialized support for the unique SCC hardware environment – the *RockyVisor*. It implements a virtualized provisioning of SMP capabilities, which allows the guest operating system kernel to experience the hardware as a traditional SMP system, so existing software can run without modification. By moving the handling of the shared-memory SMP issues into the hypervisor layer, we reduce the amount of necessary changes in both the host and the guest operating system kernel.

In the following text, we first analyze the interface between an x86 operating system and standard SMP hardware (Section II), and contrast it to the Intel SCC (Section III) hardware prototype interfaces. Based on this analysis, we present an architecture for the RockyVisor to encapsulate hardware limitations of the SCC (Section IV). Section V proposes an implementation of this architecture, and Section VII discusses related work.

The authors are with Hasso Plattner Institute for Software Systems Engineering, Potsdam, Germany – `http://www.hpi.uni-potsdam.de`. E-Mail: [jan-arne.sobania/peter.troeger/andreas.polze]@hpi.uni-potsdam.de

Manuscript received October 22, 2011.

II. x86 Operating Systems on SMP Hardware

x86 processors have been used in multi-processor machines for a long time. An accepted standard for *symmetric multi-processing (SMP)* hardware support in x86 systems is the *Intel MultiProcessor specification* [4]. It has the goal of "[extending] the performance of the existing PC/AT platform beyond the traditional single processor limit, while maintaining 100% PC/AT binary compatibility". Although superseded by today's standards such as ACPI [5], this specification still acts as a baseline for SMP support in the x86 architecture. Compliance of SMP hardware to the specification requires special attention in the following areas:

- *PC/AT Backwards Compatibility*. If an existing, non-SMP operating system is executed on the hardware, it must still function as if it was running on a single processor system.
- *Memory Subsystem*. All processors must have access to the same memory locations that must be mapped at the same physical address. Caching attributes must be consistent and caches must be coherent. Interlocked (commonly known as *atomic*) memory operations must be honored by the hardware at least on aligned accesses. Further details on the memory system are discussed later in Section II-A.
- *Interrupt Logic*. Processors must be able to receive interrupts from peripherals in a configurable manner, as well as interrupt each other individually. This is further discussed later in Section II-B.
- *Reset Support*. As part of backwards compatibility, both a software and hardware reset is required to act on all processors. Furthermore, the specification requires that SMP-capable operating systems must be given a means to reset processors individually.
- *Configuration Information*. Information about specific hardware details, like the number and identifiers of

available processors and I/O buses, are reported by the hardware via an in-memory structure known as the *MP Configuration Table*.

Notably, higher-level functionality is not expected to work synchronously between CPUs, such as the stateful operation of the memory-management unit (MMU), the handling of translation lookaside buffers (TLB) or model-specific register provisioning (timestamp counter, system call vector). Those resources are maintained individually by each processor, which makes the operating system responsible for all required consistency management activities.

A. Memory Subsystem

In the default case on x86 SMP systems, all processors have access to the same memory locations with the same physical addresses. This includes all locations in main memory, devices mapped to I/O space, as well as memory-mapped devices like controllers on peripheral extension cards. The only exception are devices that are completely processor-specific, like the local interrupt controllers (see Section II-B).

Most memory accesses are expected to work just as if the program was running on a single-processor system, so the MP specification lists the following requirements [4, pg.3-4]:

- Memory attributes, like whether or not a region is *cachable*, are identical across all processors.
- Cache coherency is guaranteed by hardware. There is no need for a software coherency mechanism.
- Caches support flushing. If a processor issues a *flush* call (i.e., the *WBINVD* instruction), only its own caches are guaranteed to be flushed.
- Atomic operations (i.e., instructions having the *LOCK* prefix) are visible to all processors. However, atomicity is guaranteed *only* on aligned accesses; caches may ignore the *LOCK* prefix for unaligned operations.
- Memory write operations are observable by other processors in the order they appear in the program.

B. Interrupt Logic

The CPU In single-processor systems is the only target for interrupt requests from peripheral devices. Traditionally, x86 CPUs have relied on an external Intel-8237-alike interrupt controller to gather interrupt requests from devices and dispatch them to the CPU. This scheme has been extended for SMPs in two major ways.

First, *Interrupt Routing* from peripheral devices to processors is configurable, thus allowing interrupts from certain devices or buses to be handled by a subset of installed processors. This can be used by administrators to link certain devices with processors for increased system throughput.

Second, *Inter-Processor-Interrupts (IPIs)* must be additionally supported in the system. They are the primary means of the operating system to trigger other processors to perform work. Similar to device interrupts, the interrupt vector number is transferred with the request. If additional information is needed by the recipient, it needs to be communicated via other channels like shared memory.

1) The Advanced Programmable Interrupt Controller: Starting with the MP specification, the traditional IBM PC-style programmable interrupt controller (Intel 8259 PIC) has been superseded by a set of *Advanced Programmable Interrupt Controllers (APICs)*. The older PIC (or a compatible device) still needs to be present for backwards compatibility in single-processor mode, but as soon as multiple processors are running, the APICs are the primary means of managing interrupts.

In the APIC system, there is one *Local APIC (LAPIC)* for each processor, as well as at set of *I/O APICs* serving interrupts from peripherals; typically one IOAPIC per device bus, but other topologies could be used by manufacturers as well. IOAPICs are mapped into global memory space, whereas local APICs are mapped for their respective processor only.

2) IPIs and Processor Reset: A special case of signaling other processors to perform work is a processor reset or initialization request. Due to backwards compatibility, the MP specification requires that only a single *Bootstrap Processor (BSP)* is active when the BIOS code is executed. If the machine is used by a single-processor operating system, the BSP will act as this single processor and no other processor will be active until a system restart. The BSP need not be predetermined by the hardware, though; it is also possible to start all processors at power-on, then run an agreement protocol as part of the BIOS startup sequence to determine a BSP and stop all other processors afterward.

For a multi-processor operating system, its startup code is responsible for detecting the presence of other processors and starting them. These other processors, named *Application Processors (APs)* in the Intel MP specification, are identified via the identifiers of their local APIC on the APIC bus. Startup is accomplished by sending special IPIs; the type of IPIs depends on whether the system uses external (Intel 82489DX) or on-die local APICs.

The INIT IPI causes the remote processor's local APIC to reset the processor state and begin the normal bootstrap sequence, just as if power had been turned on for the processor. In comparison, the STARTUP IPI causes the instruction pointer to change to an address specified by the vector number in the IPI message; all other CPU state remains unchanged.

III. Why the SCC is no x86 SMP

The SCC processor design consists of 48 *GaussLake* cores that are organized in 24 dual-processor tiles, each having:

- its own independent clock generator,
- a set of core configuration registers,
- a scratch-pad memory called the *Message-Passing Buffer (MPB)*
- a message router that interfaces the tile to the on-die communication network [1].

The prototype platform does not have a "chipset", but instead contains an FPGA that is connected directly to the on-die network [6]. Depending on the firmware version, the FPGA contains a set of core-specific devices like queues for the Ethernet ports. Furthermore, it acts as a communication bridge, forwarding packets between the on-die network and the *Management Console PC (MCPC)* connected via PCI-Express.

From an overall architecture perspective, each of the 48 processor cores conforms to the standard 32-Bit x86 architecture. However, the whole SCC cannot be treated as x86 SMP system, since it does not conform to the MP specification in several areas, as described in the next sections. While some of these differences could be alleviated by software running either on the GaussLake cores itself (like a modified BIOS), or the FPGA or the MCPC (for device emulation), certain others like differences in the memory subsystem are inherent for the hardware and would need a new silicon revision to be changed. We discuss each of the categories below, providing information on what could and cannot be changed in software.

A. BIOS Support

The MP specification requires the hardware to be in a specific state after the BIOS has performed its startup processing: only one processor (the Bootstrap Processor, BSP) shall be running, while each other (Application Processor, AP) shall be placed in a state where it is inactive and waits for an INIT or STARTUP IPI.

On the SCC, there does not exist a BIOS up to and including sccKit 1.4.1.3. We submitted our minimal SCC BIOS introduced in [2] for inclusion in sccKit 1.4.2, but it also does not include support for the MP specification or the warm restart. Instead, cores are reset directly from the MCPC, and the initial memory contents that would be constructed from a bootloader on a standard x86 system are transferred directly via the MCPC to the memory controllers [2].

However, these differences could easily be overcome by implementing a full BIOS; e.g., by providing the warm restart vector as well as MP or ACPI tables.

B. Peripheral devices

In the current sccKit releases, no peripheral devices known from a standard PC are implemented for the SCC. Instead, if a core performs an I/O operation (via either the *IN* or *OUT* x86 instruction), the corresponding network packet is sent to the MCPC and potentially handled there. Implementing these devices relates to an appropriate device emulator on the MCPC side – the authors have prototyped this approach by implementing virtual 16550A UARTs at standard PC addresses, which could then be handled by build-in Linux device drivers. Real-world systems based on SCC could then contain the according true implementations of these devices.

C. Memory

As written in the *SCC External Architecture Specification (EAS)* [6], the memory subsystem of the SCC is considerably different than that of a standard x86 SMP. The main differences are as follows:

1) *Additional memory mapping layer*. The mapping of a core's physical addresses to system addresses is controlled via *Look-Up Tables (LUTs)*; these allow each core to have a completely different view of the global memory space. For SMP operation, the LUTs could be configured to map all participating cores to the same region in main memory.
2) *Cache coherency* is not maintained by the hardware, so there is no single view of a global memory space if caches are enabled.
3) Processor cores do not have a means to communicate the *LOCK* signal to memory controllers, even with disabled caches.

A formal workaround for incoherent caches is to disable them completely, which is also clearly allowed by the MP specification – but may not be desirable because of the expected performance impact. However, there is no such workaround for the missing *LOCK* signal: the corresponding line from the GaussLake is not connected in hardware, and the processor itself does not allow to emulate or trap on instructions that use the prefix. Therefore, atomic operations do not work on the SCC, as *LOCK* is silently discarded. According to the checklist [4] in the MP specification, this makes the SCC non-compliant, which prevents the operation as standard x86 SMP system.

As part of our solution, we propose to solve this issue by using a combination of a software coherency layer and virtualization. When caches are enabled, the following conditions are sufficient to prevent data corruption in a non-coherent hardware environment:

1) At most one 'owner' core has a physical page mapped for write access at each point in time
2) When a write access has to be performed by another core, the 'owner' must perform a flush

The initial idea relies now on the possibility to prevent a page write access by setting the corresponding page table protection bits [7]. Furthermore, for guaranteeing sequential consistency, it is necessary to disallow reading of a page on a core if *any* other core has mapped it for writing. In our concept, the *RockyVisor* distributed hypervisor fulfills the role of such a memory access coordination entity.

D. Interrupt Handling

Similar to the memory subsystem, the SCC's interrupt handling support also differs significantly from traditional x86 systems. The GaussLake cores contain a local APIC, just like the original P54C; however, these APICs are not connected. Instead, the corresponding lines to the processor core are exposed directly via the tile's configuration registers [6].

Although one end of the bus is exposed, there is no means to actually *send* a message over to a local APIC, as the signaling protocol is not publicly documented. Furthermore, even if it would be known, *receiving* messages is not guaranteed to work, as it would require polling the bus wires at a high speed, with no means to prioritize such traffic on the on-die mesh or the link to the MCPC in the SCC system.

As one possible solution, we chose to simulate the APIC operations at a higher abstraction level, by sending corresponding IPI messages between hypervisor instances. In combination with the memory management interception of a hypervisor described above, we end up in a solution where a distributed hypervisor adds the missing SMP capabilities for the SCC platform.

IV. SMP via Virtualization

Although the discussion so far is limited to one shared-memory SMP system architecture, our general discussion is strongly related to what is known as a *Single-System Image (SSI)* view. In SSI systems, any process – no matter on which physical processor or system of the cluster it runs – has always the same view of the multi-computer. The term "SSI" can refer to various layers of either software or hardware [8, pp350], though, but we are discussing it here only in the context of shared-memory SMP systems.

In traditional operating systems, there are two main operational modes for executing code. Applications run in non-privileged (user) mode, whereas the kernel runs in privileged (kernel) mode to control the hardware. When simulating an SMP system through a hypervisor, we deliberately extend this model. A new layer below the kernel is now responsible for converting the existing hardware interface to a virtual one the kernel understands. For traditional virtualization, the VMM just provides the same interface (or a subset) to the kernel as it would find on real physical hardware, probably with some para-virtualized devices for increasing performance. In our architecture, the VMM layer is responsible for simulating all aspects of a shared-memory SMP the real hardware lacks, which let's the VMM transparently add new hardware capabilities.

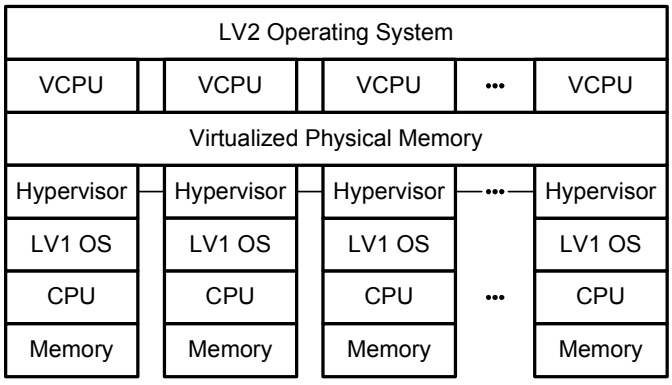


Fig. 1. RockyVisor with Guest OS

For the given approach, it is not relevant whether the VMM itself runs directly on the physical hardware (Type 1 VMM) or if it runs as a process in another host operating system (Type 2 VMM). We also do not distinguish implementation techniques of VMMs; e.g., whether it uses hardware-assisted virtualization on a trap-and-emulate-virtualizable instruction set, or a software technique such as binary translation or para-virtualization [9]. As discussed in Section V below, we chose a Type 2 VMM approach (Hosted VM) for our prototype to allow for reusing code from an open-source hypervisor (lguest). Figure 1 shows the resulting architecture.

We assume each processor to run a separate "Level 1" operating system instance, in order to manage low-level hardware resources like communication devices. On top of this operating system, our RockyVisor process runs as a regular application. Multiple RockyVisor instances cooperate to provide a single, coherent memory space; in addition, each RockyVisor supplies a single virtual CPU. Inside the resulting virtual machine, the "Level 2" operating system is executed as a standard SMP operating system.

To realize the proposed software architecture on the SCC, there are two fundamental problems to solve: The SCC CPU cores must virtualized, and the overall virtual hardware used by the LV2 operating system must show behavior similar or equal to a real x86 SMP.

V. The RockyVisor: Virtualizing the SCC

As the GaussLake processor cores in the SCC are based on the P54C [6], they do not contain virtualization support instructions. Classical trap-and-emulate virtualization is therefore not possible [10]. We decided to use para-virtualization for our prototype, since the *lguest* hypervisor is already providing the capability with the standard distribution of the Linux kernel. At the same time, this hypervisor is compact enough to be easily understandable for research purposes. *lguest* has been developed by Rusty Russell and is part of Linux since version 2.6.23. It is a minimal, yet fully-functional hypervisor that is implemented as a loadable kernel module. It supports para-virtualization only, so the guest kernel must be changed accordingly. For Linux, a corresponding sub-architecture supplying necessary callbacks on the kernel's *paravirt* interface is provided.

The hypervisor itself has basic support for MMU virtualization using *shadow page tables* [9], as well as device emulation via Linux' build-in *virtio* framework. Unlike Xen or KVM, *lguest* is not meant as a commercial-grade solution, but as a platform for research and experimentation. Specifically, lguest favors readability of the hypervisor code over performance wherever possible.

The current *lguest* hypervisor does not support multi-processor guest systems. Specifically, it does not fully distinguish *per-processor* from *machine* state. Although the *lg* kernel module contains structures for both hypervisor and virtual CPU (vcpu) state, the separation is incomplete: some fields are still present in the wrong structure, and the paravirt layer installed in the guest does not have a notion of more than processor.

In order to implement the RockyVisor, we first developed an SMP extension of *lguest*. For this, we implemented support for more than one virtual processor in the guest's *paravirt* layer, separating the hypervisor state into global and per-virtual-CPU structures. The global state now consists of just the virtual interrupt controller; virtual processor state includes the interrupt enable flag, mask of pending interrupts, as well as information required by the virtual MMU.

We also added new hypercalls to support emulation of local APICs. As discussed in section II-B, local APICs provide two basic operations not required in single-processor mode: startup of APs (secondary processors), and signaling between processors. In our implementation, we added corresponding hypercalls that forward these operations to the *lguest* launcher process, which then sends to other virtual CPUs via sockets.

VI. SMP MMU Virtualization

As briefly mentioned in section II, the MMUs of all processors act independently in an x86 SMP. Unlike cache coherency

or the software-visible *LOCK* prefix, there is no implicit communication between processors, so all MMU state changes need to be explicitly requested by the operating system.

However, virtualization also provides opportunities for optimization, especially in regards to the MMU [9]. Some of these optimizations can have a major impact on hypervisor design, which we are going to discuss in the remainder of this section.

On processors that support *nested page tables*, MMU virtualization can be trivially implemented. This is not the case for the SCC, as the GaussLake cores are based on the P54C that predates any virtualization assists in hardware. Therefore, MMU virtualization requires other techniques that allow the guest operating system to safely change the real page tables used by the physical hardware.

An example is Xen, which allows the guest to manipulate its page table *directly*, giving the guest direct knowledge of the physical page frame numbers it uses for its mappings. Before such a page table is used by the processor, the hypervisor just needs to check that the guest does not install any page numbers it is not allowed to access. Other implementations use *shadow* structures that are maintained by the hypervisor and fully hidden from the guest operating system.

A. Emulated TLB

Another technique, traditionally implemented by VMware and (in a modified form) by *lguest*, is called *emulated transaction look-aside buffer (TLB)* or (in optimized form) *shadow page tables* [9]. For both techniques, the guest manipulates its own page tables, and does not need any knowledge of real (physical) memory addresses. The page table that is used by the MMU to translate guest-virtual to physical addresses is manipulated only by the hypervisor.

For the emulated TLB, the real page table represents the TLB of the virtual CPU. On a guest page fault, the hypervisor interprets the guest's page tables. If it does not find a mapping or its attributes do not provide sufficient access, the page fault is reflected into the VM. Otherwise, the hypervisor retrieves the guest-physical page number, performs the translation to a host-physical page number itself, then install a mapping from the *guest-virtual* to *host-physical* address in the page table. On any later access, the processor just uses the already installed entry directly.

The software TLB has a major drawback: loading *any* TLB entry is considerably more expensive than the corresponding hardware operation in a non-virtualized environment, as it requires both a software pagetable walk to fill the software TLB, then another hardware pagetable walk to fill the real TLB. Furthermore, the real TLB needs to be flushed more often, as the hypervisor's page fault handler runs in hypervisor space instead of guest space. Any page fault results in two TLB flushes, whereas real hardware does not need to perform any flushes in this case. Similarly, reloading the guest's virtual CR3 (e.g., on a context switch in LV2) also leads to clearing the entire software TLB, thus resulting in a storm of hidden page faults afterward. We are currently investigating these issues for further improvement.

B. Shadow Page Tables

As an optimization to the software TLB, hypervisors can use *shadow page tables*. These use multiple page table hierarchies, one for each value of the guest's CR3 register: if the guest operating system switches to another process that has already been active before, it can just switch to the cached page table corresponding to this CR3 value.

One major problem with shadow page tables is to decide when to remove entries from inactive, but still cached shadows. For the hypervisor to notice changes to page tables, it can make use of *traces*: once a shadow page table has been constructed for a page in guest memory, the guest is transparently disallowed to perform any further changes. If the guest performs a write, the hypervisor recognizes that the page being changed has an associated shadow page table and invalidates it.The lguest hypervisor also implements shadow page tables, but the invalidation problem is solved in a different way: it relies on the guest operating system to notify the hypervisor of any changes to page tables. Therefore, whenever an entry is removed or changed – in *any* page table – the hypervisor is informed and purges corresponding entries from shadow page tables.

C. RockyVisor Page Tables

As the *emulated TLB* is fully consistent with documented behavior of real hardware, it was our first choice when designing page table support for the cooperative RockyVisor. The existing implementation of shadow page tables by the underlying *lguest* has a major drawback here: as the guest needs to communicate changes of the page tables to the hypervisors, the *RockyVisor* would need to communicate all changes to page tables across the entire system, leading to a potentially huge number of useless simulated IPIs.

We implemented an optimization that is based on the following observation: Linux maintains a bitmask for each page table, noting which processors reference the page table. If the kernel modifies a page table in memory, it automatically sends a corresponding IPI to each CPU listed in the bitmask, requesting it to perform the necessary TLB flush. The abstraction level of this operation is fairly high, so more detailed information is available: for example, the affected virtual addresses are known, so a subset of TLB entries could be flushed if the processor supported such a "selective" TLB flush.

The page table indication bits are normally changed during context switch. In our architecture, though, the guest *just sets* the bit when it activates a page table. The bit of the old page table's mask is not cleared, because it may still be cached by the hypervisor. Instead, the address of the bitmask is communicated to the hypervisor as part of context switch, allowing it to clear the bit for that virtual CPU when the shadow page table is no longer available.

D. RockyVisor Memory Coherency

As caches on the SCC are not coherent, the RockyVisor cannot simply map one physical page on more than one core

at a time. Instead, when enabling caches, it needs to obey the criteria for cache coherency as discussed in section III-C.

The *lguest* implementation of shadow paging allows the hypervisor to keep track of all pages it has mapped for a virtual CPU, and transparently change these mappings (e.g., to make a page read-only) without the need to communicate with the guest operating system. If a guest later requires to access the page again, it will encounter another hidden page fault. Therefore, on our architecture, memory coherency can be implemented as follows:

- If a guest page fault is encountered, the corresponding page is requested from the memory coherency driver. The page fault handler completes once the page is available.
- If a shadow page table is torn down, all referenced pages are released to the coherency driver.
- If the coherency driver is requested for a page that violates above conditions, it informs the hypervisors that still have the page mapped, requesting them to release the page.

We are currently working on the according extension of the prototype implementation.

VII. Related Work

As precondition for the work presented here, we have demonstrated a modified Linux kernel before that works on a single core of the SCC [2]. This kernel only works on a single SCC core at a time and does not provide SMP operation.

A completely different approach for multiprocessor operation is the Barrelfish operating system [11]. Barrelfish uses satellite kernels and demands tailored high-level applications. There is no backward compatibility to any existing application.

vNUMA [12], developed by Matthew Chapman as part of his Ph.D. thesis, is a distributed hypervisor for IA-64 processors. It simulates an SMP system on networked workstations, using Gigabit Ethernet as node interconnect. NEX [13] by Xiao Wang *et al.* is a similar effort based on the open-source XEN hypervisor, but requires hardware extensions for virtualization. Versatile SMP [14] by ScaleMP is a commercial product that claims to support up to 1024 processors (with up to 8192 cores) on standard computers, interconnected via Infiniband. All these projects rely on the availability of a standard x86 processor, especially with respect to memory coherency and device handling.

Finally, *MetalSVM* [15] is another project for hypervisor-based SMP on the SCC. In contrast to the RockyVisor, MetalSVM implements a Type 1 VMM that runs directly on the hardware, instead of another operating system instance.

VIII. Conclusion and Future Work

In this paper, we analyzed the properties of an x86 SMP system and denoted in which points the Intel SCC research hardware differs. Based on this analysis, we presented the architecture and first steps for the *RockyVisor* that is intended to run a para-virtualized SMP operating system on the SCC.

We are currently in the process of finalizing this architecture, using the modified Linux kernel presented in [2] as a unified LV1 and LV2 kernel. In the current state of our prototype, a single GaussLake core can simulate a 2-way SMP VM. The modified shadow page table mechanism, as well as the reverse mapping for finding which shadows contain references to specific physical pages have also been implemented, but our implementation still lacks the cache coherency layer. Future work will focus on the completion of the prototype and the experimental evaluation.

References

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, and et al., "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS," *2010 IEEE International SolidState Circuits Conference ISSCC*, vol. 9, pp. 58–59, 2010.

[2] J.-A. Sobania, P. Tröger, and A. Polze, "Linux Operating System Support for the SCC Platform - An Analysis," in *3rd Many-core Applications Research Community (MARC) Symposium*. KIT Scientific Publishing, Karlsruhe , 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] Intel Corporation, *MultiProcessor Specification*, May 1997.

[5] Hewlett-Packard Corporation, *Advanced Configuration and Power Interface Specification*, Apr. 2010.

[6] Intel Labs, *SCC External Architecture Specification (EAS)*, Apr. 2010.

[7] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, 1999.

[8] G. F. Pfister, *In search of clusters*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[9] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *SIGARCH Comput. Archit. News*, vol. 34, pp. 2–13, Oct. 2006.

[10] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor," in *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 10–10.

[11] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system," in *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[12] M. Chapman, "vNUMA: Virtual Shared-Memory Multiprocessors," Ph.D. dissertation, Computer Science and Engineering, The University of New South Wales, 2008.

[13] X. Wang, M. Zhu, L. Xiao, Z. Liu, X. Zhang, and X. Li, "NEX: Virtual Machine Monitor Level Single System Support in Xen," in *International Workshop on Education Technology and Computer Science*, vol. 3. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 1047–1051.

[14] ScaleMP, "Versatile SMP (vSMP) Architecture," http://www.scalemp.com/architecture.

[15] S. Lankes, "MetalSVM: A Bare-Metal Hypervisor for Non-Coherent Memory-Coupled Cores," http://www.lfbs.rwth-aachen.de/content/metalsvm, 2011.

Fiasco.OC on the SCC

:

Markus Partheymüller, Julian Stecklina, Björn Döbel
{mpartheym,jsteckli,doebel}@tudos.org
Operating Systems Group, TU Dresden

***Abstract*—Our initial goal was to port the Fiasco.OC microkernel to the Single-Chip Cloud Computer in order to use it as an experimentation platform for our research. In this paper we describe the few hardships we encountered during this porting work and our solutions for those problems. With our kernel running on top of the SCC, we evaluated message passing performance between cores and came across a cache-related issue that can seriously decrease message-passing performance.**

I. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research. To facilitate our research in this area, we decided to port the Fiasco.OC microkernel developed in our group to the SCC. Establishing a working environment to run Fiasco.OC on the SCC will then pioneer experiments concerning all kinds of many-core related research topics, including energy saving, inter-core message passing and trusted, robust server applications.

With the SCC consisting of x86 CPUs and Fiasco.OC already available for this CPU architecture, most of the porting work was straightforward. However, on our way we encountered a couple of issues related to the non-standard SCC hardware features. In this paper we present these issues and describe our solutions.

We then describe a first experiment we conducted to evaluate message passing performance between instances of Fiasco.OC running on different SCC cores. This experiment led us to discover a caching-related hardware issue, which for unaware developers may pose a serious performance problem.

II. Fiasco.OC and L4Re

Our work takes place in the context of the Fiasco.OC microkernel [1]. Fiasco.OC is the only piece of software running in privileged processor mode and following the philosophy of L4 microkernels provides only mechanisms for constructing an operating system but does not implement any policies, such as resource management.

The main mechanism provided by Fiasco.OC are capabilities [2], which can be viewed as kernel-protected references to objects. The objects themselves are either implemented in the kernel (e.g., tasks, threads, interrupts) or by user-level applications (e.g., memory managers, device drivers, protocol stacks). An object's functionality can be used by clients possessing access to an object's capability through Fiasco.OC's invoke() system call. The close relation between objects and capabilities in Fiasco.OC also motivated choice of the .OC suffix.

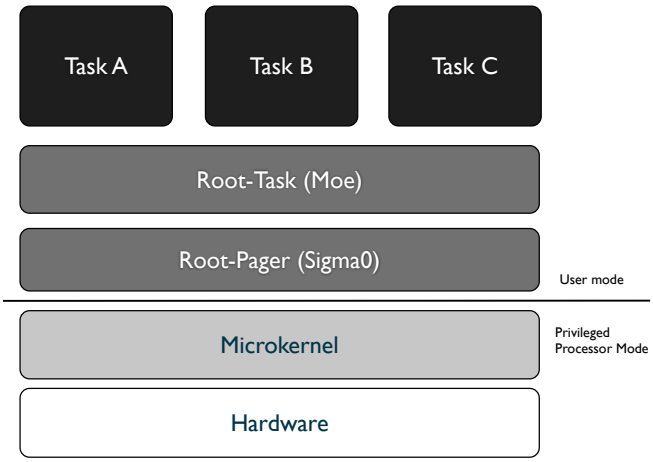


Fig. 1. Basic structure of an L4Re based System

Policies, such as resource management decisions, are implemented on top of Fiasco.OC as user-level applications. The most basic services, such as a C library, application memory managers, and a loader for ELF binaries, are implemented in a set of libraries and servers called the L4 Runtime Environment (L4Re) [3]. Apart from that, our public SVN repository contains a wide range of additional software such as a GUI service, ports of various libraries (Qt, libjpeg, freetype), as well as software packages, such as sqlite and Valgrind [4].

Booting an L4Re setup on Fiasco.OC involves first bootstrapping the microkernel itself. Thereafter, the sigma0 root pager is booted, which initially owns all resources in the system. However, sigma0 only serves as the user-level pager for the next application in line: the root task, called Moe. Moe then provides essential services required by user applications, such as a program loader, an address space manager that is injected into every task, and a memory allocator serving regions of virtual memory. An overview of this structure is given in Figure 1.

Fiasco.OC and L4Re provide all means necessary to implement user-level applications. One of the most demanding applications running on top of our system is L4Linux [5], a para-virtualized version of the Linux kernel running as a user-level application on top of Fiasco.OC. With this approach we are able to run arbitrary Linux binaries on top of our system, while it still allows for all the benefits of virtualization: as the Linux kernel is running as an unprivileged user application, it becomes isolated from the rest of the system and outside

services may impose resource constraints as well as enforce security policies. Furthermore, it is possible to increase resource utilization by running multiple instances of L4Linux on the same core.

III. Fiasco on the SCC

The SCC being an x86 architecture based Pentium processor system suggests to use the existing x86 port of Fiasco. However, the absence of typical hardware devices such as the BIOS, the Programmable Interval Timer (PIT), a keyboard controller and a graphics card did not allow to run the kernel unmodified. In addition, I/O requests issued from the cores are routed to the management PC, because there are no local devices that can handle them. The x86 boot process relies on these devices and would therefore fail immediately.

As the SCC allows running dedicated OS instances on every core, we decided to initially port Fiasco.OC in a way that each core executes one Fiasco.OC instance and explore how the SCC's specific communication mechanisms can be used for messaging between the instances. In later experiments we plan to also explore the concept of a *Single System Image*.

To cope with the SCC's lack of firmware, we implemented an sccKit application called *sccLoad*. This application is responsible to set up a multiboot-compliant boot environment, which provides a memory map, a couple of register values, and a kernel command line. From then on, it is possible to boot the kernel using its native x86 boot code.

While booting Fiasco.OC, a timer calibration is performed, which calculates the ratio of clock cycles (accessible through the TSC register) to time values like nanoseconds. This is usually done using a second timer with configurable frequency. By setting this frequency to a known value and measuring the clock cycles during a fixed time interval, the ratio can be derived. On the SCC a second timer (e.g. the PIT) is missing and therefore there is no way for the core to determine the ratio on its own. To address this issue two solutions are possible: The first one is to calculate the ratio beforehand and hard-code it into the kernel. However, this assumes a fixed core frequency which is not necessarily the case on the SCC because of frequency scaling mechanisms. The second solution is to pass the core frequency as a command line parameter. It allows for this flexibility and was therefore preferred.

Beside the boot environment, *sccLoad* emulates a basic serial console using an I/O handler and can currently run, monitor and even control (via keyboard input) multiple instances of the kernel ELF [6] binary from the management PC. The I/O handler ensures that all I/O related code can be used without modification. When we started our work, the Fiasco.OC kernel was not able to use the Local APIC as configurable source for interrupts other than the timer. However, the SCC provides inter-core interrupts through the Local APIC and therefore, we needed to modify Fiasco.OC. We added an additional initialization section to the kernel binary, which sets up the Local APIC to trigger an interrupt vector directly representing the inter-core interrupt requests.

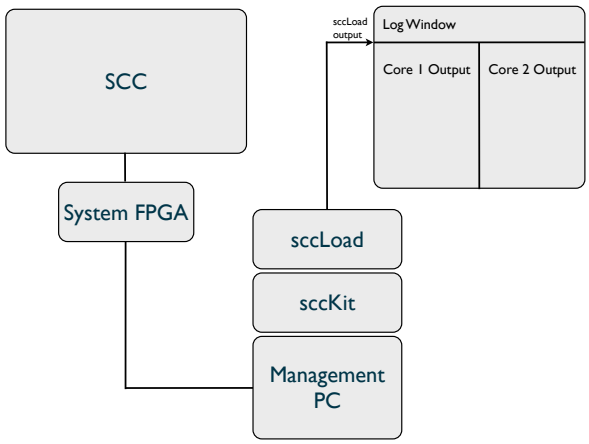


Fig. 2. Environment setup for monitoring two Fiasco.OC instances

Further addtional SCC features include the new MPBT memory type and the message passing buffers. To support these, we disabled support for *Page Size Extension* in the kernel and at the same time enabled the new MPBT feature. In the current state of our port, the addresses of the Message Passing Buffer (MPB) and the control register buffer (CRB) are not provided as regular RAM, but instead hardcoded into the kernel, assuming the standard memory layout described in the EAS. This enables us to view these memory regions as I/O memory. An L4Re server called *io* then provides these locations as flexibly mappable I/O regions.

While working on the *io* server, we discovered an issue that arises when the MPB is declared as uncacheable [7]. When doing so, reading data from MPB memory results in the first eight bytes of the memory content being repeated four times. The remaining 24 bytes of each cache line are inaccessible. To circumvent this, we had to make the *io* server work with the MPB as cacheable memory, which had not been necessary for any previous hardware resources managed by this server. While this combination of settings is not a usual use case for the MPB, it would have been interesting to examine the performance of shared DDR memory tagged as MPBT, because the need for an L2 cache flush routine could be eliminated while using the Write Combine Buffer to circumvent performance issues caused by *non-allocate-on-write*.

Altogether we now have an environment that can run basically any ELF binary produced by the L4Re build infrastructure, for example a Fiasco microkernel with a message passing application or even multiple instances of L4Linux.

IV. Preliminary results

As a first experiment on top of our newly ported OS infrastructure, we tried to evaluate message passing performance between different cores. We started with two cores, both located in tile [0,0], where core 1 sends a 32 MiB large message (separated into packets of 4096 bytes) to core 0 through the MPB. Core 0 copies the content into its own

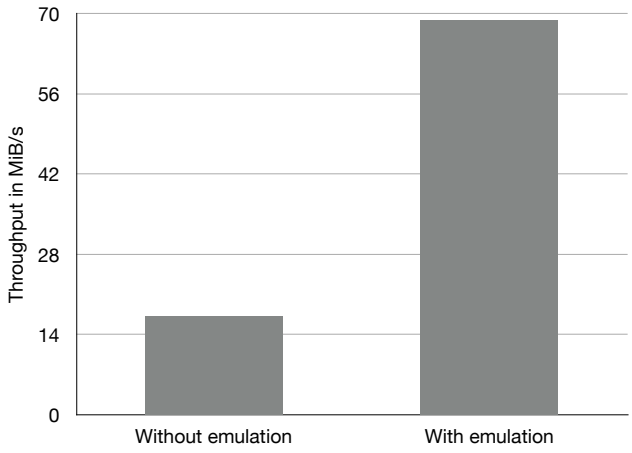


Fig. 3. Performance impact of non-allocate-on-write for `memcpy`.

memory and acknowledges the transfer to core 47, which measures the time that is needed to transfer the whole 32 MiB. The left bar of Figure 3 shows the obtained throughput result for the standard frequency configuration (533/800/800).

Our experimental data shows a significant impact of a cache-related hardware design choice. The cache property, often referred to as *non-allocate-on-write*, affects the performance of message passing or rather the actual memory accesses. When a program causes a write miss by writing to a memory location that is not present in the cache hierarchy, the caches do not allocate a new cache line for the write request. The request is instead passed on to the memory controller. In the worst case of byte-wise writes, each write issues an individual transaction to main memory.

The significance of this performance decrease can be demonstrated by implementing a software emulation of *allocate-on-write*. For a `memcpy` operation, before writing to a destination address, the location is read. This happens every 32 bytes, thus causing the caches to allocate a cache line for each address the operation will write to. We implemented this emulation and the right bar in Figure 3 shows, that using software emulated allocate-on-write significantly improves the message-passing throughput.

The results of these experiments will be evaluated to determine our next steps concerning message passing on top of Fiasco.OC. We did not intend the application to become a real message passing framework, but rather use it as a prototype to assess the difficulties and opportunities encountered when implementing communication software for the SCC in our microkernel environment.

V. Future Work

Currently we are conducting preliminary experiments exploring the inter-core message passing facilities in terms of performance and implementation complexity. We expect these experiments to give us results helping to decide if and how we should continue our work on message passing. Choices include a combination of a message passing library and an L4Re server multiplexing the MPBs as well as a RCCE port. Related to message passing is the performance issue mentioned above, leading to the question how to compensate for this shortcoming of the cache architecture and where to implement the improvement.

Also in connection to message passing is the inherent security issue when allowing all cores to access every memory location with full rights. In the standard layout, each core can read and modify all configuration registers, MPBs and DDR memory. By modifying its own LUT entries it can basically control the entire system. When the MPB is used for communication, there is no means to prevent cores from tampering with MPB data once they have access to the MPB locations.

Exploring ways to establish security measures could be an interesting topic. For example, one could think of sandboxing cores by not giving them access to configuration registers. This, of course, would render inter-core interrupts, which are implemented using those registers, impossible. A slightly weaker restriction, allowing access to all configuration registers but the cores' own one, would enable interrupts again but also gives the cores the ability to mess with other cores.

Instead of the configuration registers, also the recently added functionality of a global interrupt controller could be used along with the other additional features (global timestamp counter, global atomic increment counter), which are currently not implemented in our software.

In addition to covering performance issues, we also believe the SCC to be a good platform to research energy-related issues, because it provides regulators allowing to modify tiles' frequency and voltage settings. These regulators also allow for controlled *undervolting*, which may lead to energy savings as well failures with yet unknown characteristics. We plan to evaluate the concrete manifestations of such failures and tradeoffs between power saving and fault probability in future experiments.

VI. Conclusion

Although porting Fiasco.OC to the SCC seemed straightforward, we have encountered several problems that made the port more difficult than expected, such as bugs in the sccKit and cache misbehavior. Nevertheless we could establish a working environment for future experiments with microkernels on many-core systems in order to investigate particularities and problems that can occur on such systems, as well as possible approaches to solve them.

Acknowledgments

The authors would like to thank Adam Lackorzyński and Alexander Warg for sharing their Fiasco.OC knowledge. Additional thanks go to Intel and the MARC community, especially Ted Kubaska, Jim Held, and Michiel W. van Tol, who provided insights during discussions in the community forum.

REFERENCES

[1] TU Dresden OS Group, "Fiasco.OC microkernel," http://os.inf.tu-dresden.de/fiasco/, 2010.

[2] A. Lackorzynski and A. Warg, "Taming Subsystems: Capabilities as Universal Resource Access Control in L4," in *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*. Nuremberg, Germany: ACM, 2009, pp. 25–30.

[3] TU Dresden OS Group, "L4 runtime environment," http://os.inf.tu-dresden.de/l4re/, 2010.

[4] A. Pohle, B. Döbel, M. Roitzsch, and H. Härtig, "Capability wrangling made easy: debugging on a microkernel with Valgrind," in *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2010, pp. 3–12.

[5] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *Symposium on Operating Systems Principles*, Saint Malo, France, 1997, pp. 66–77.

[6] "Tool Interface Standard - Executable and Linkable Format," http://www.rcollins.org/intel.doc/Tools.html, 1998.

[7] M. Partheymüller, "Strange behaviour when reading MPB," http://communities.intel.com/message/133047, 2011.