A Fast Inter-Kernel Communication and Synchronization Layer for MetalSVM

Pablo Reble, Stefan Lankes, Carsten Clauss, Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
{*reble,lankes,clauss,bemmerl*}*@lfbs.rwth-aachen.de*

***Abstract*—In this paper, we present the basic concepts for fast inter-kernel communication and synchronization layer motivated by the realization of a SCC-related shared virtual memory management system, called MetalSVM. This scalable memory management system is implemented in terms of a bare-metal hypervisor, located within a virtualization layer between the SCC's hardware and actual operating system. In this context, we explore the impact of the mesh interconnect to low level synchronization. We introduce new scaling synchronization routines based on SCC's hardware synchronization support targeting improvements of the usability of the shared memory programming model and present first performance results.**

***Keywords*—Inter-kernel Communication, Low-level Synchronization, Shared Virtual Memory**

I. Introduction

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 cores arranged in a 6×4 on-die mesh of tiles with two cores per tile.

Each core possesses 8 kByte of a fast on-die memory that is also accessible to all other cores in a shared-memory manner. These special memory regions are the so-called *Message-Passing Buffers* (MPBs) of the SCC. The SCC's architecture does not provide any cache coherency between the cores, but rather offers a low-latency infrastructure in terms of these MPBs for explicit message-passing between the cores. Thus, the processor resembles a *Cluster-on-Chip* architecture with distributed but shared memory.

The focus of this paper is a layer to realize fast inter-kernel communication and synchronization for *MetalSVM*. *MetalSVM* will be implemented in terms of a bare-metal hypervisor, located within a virtualization layer between the SCC's hardware and the actual operating system. This new hypervisor will undertake the crucial task of coherency management by utilizing special SCC-related features such as on-die Message-Passing Buffers (MPBs). In that way, common Linux kernels will be able to run almost transparently across the entire SCC system.

A requirement to inter-kernel synchronization is to control the access to shared resources such as physical devices or units of information. Limited availability of resources can be controlled by defining the access as a *Critical Section*. A critical section can thus be mapped to low-level synchronization methods such as a simple spin-lock.

Synchronization is a challenge in shared-memory programming. Especially for the SCC, whereas a high contention is possible, the currently available synchronization primitives (lock and barrier) could become a scalability bottleneck.

II. Motivation behind MetalSVM

A common way to use a distributed memory architecture is the utilization of the message passing programming model. However, many applications show a strong benefit when using the shared memory programming model. *Shared Virtual Memory* is an old concept to enable the shared memory programming model on such architectures. However, the success story of SVM systems could be greater. Many implementations are realized as additional libraries or as extensions to a programming language. In this case, only parts of the program data will be shared and a strict disjunction between the private and the shared memory is required. Intel's Cluster OpenMP is a typical example of such SVM systems, which in turn is based on TreadMarks [2], and first experiences are summarized in [3] . However, the disjunction between private and shared memory has side effects on traditional programming languages like *C/C++*. For instance, if a data structure is located in the shared virtual memory, the programmer has to guarantee that all pointers within this data structure refer also to the shared memory. Therefore, it is extremely complex to port an existing software stack to such an SVM system.

Furthermore, an SVM system virtualizes only the memory. Therefore, on distributed systems, the access to other resources like the file system requires additional arrangements. The integration of an SVM system into a distributed operating system, in order to offer the view of a unique SMP machine, increases the usability. Such systems are often specified as *Single System Image* (SSI) and *Kerrighed*[1] is an typical example for an SSI system. However, an extreme complexity and difficulties in maintainability are attributes of SSI's.

In the *MetalSVM* project, the SVM systems will be integrated into a hypervisor so that a common Linux is able

[1] http://www.kerrighed.org

to run as a virtual machine on the top of *MetalSVM*. Such a system is more simple to realize than a huge distributed operating system because it realizes only a unique view to the hardware and not to a whole operating system (including the file system and the process management). Several other projects have been started in this area. An example for a hypervisor-based SVM system is vNUMA [4] that has been implemented on the Intel Itanium processor architecture. For x86-based compute clusters, the so-called vSMP architecture developed by ScaleMP[2] allows for cluster-wide cache-coherent memory sharing.

The main difference between these approaches and *MetalSVM* is that they explicitly use traditional networks (Ethernet, InfiniBand) between the cluster nodes in order to realize their coherency protocols, whereas *MetalSVM* should support the SCC's distinguishing capabilities of transparent read/write access to the global off-die shared memory. However, a recent evaluation [5] with synthetic kernel benchmarks as well as with real-world applications has shown that ScaleMP's vSMP architecture can stand the test if its distinct NUMA characteristic is taken into account. We have already developed optimized applications for vSMP. In our scenario, vSMP produces only an overhead between 6% and 9%. In turn, 97% of this overhead is created by migrating the pages between the nodes.

vSMP benefits that InfiniBand supports DMA transactions, which don't stress the cores. The SCC doesn't support a similar feature. However, in MetalSVM the shared pages or the write notices – this depends on the memory model – are located in the shared off-die memory. Therefore every core has a transparent access to the shared data and doesn't take computation time of the other cores. MetalSVM has only to signalize the changes with small messages, which are sent via our inter-kernel communication layer. This layer will be explained in detail in Section VII.

Figure 1 shows exemplarily the simplest form of an SVM subsystem. In this case, only one core has access to a page

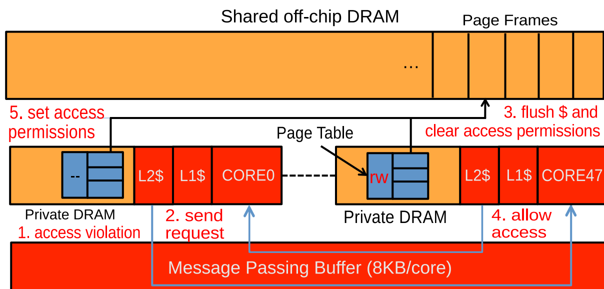

Figure 1. One Strategy to Realize a SVM Subsystem

frame.

If core 0 tries to access to a page frame, that holds core 47, an access violation will be triggered. Afterwards, the kernel of core 0 sends an access request to core 47. Core 47 receives this request, flushes its cashes, deletes the access permissions in its page table and sends the allowance for the page access back to core 0. Following this, core 0 sets the read/write permissions to its page table and continues its computation.

Nevertheless, the evaluation of vSMP has also shown that expensive synchronization is the big drawback of this architecture. We believe that especially this drawback will not occur in the context of our solution for the SCC, because *MetalSVM* provides better options to realize scalable synchronization primitives.

III. Basic Concept of MetalSVM

Figure 2 depicts *MetalSVM's* design approach, that allows in principle a common Linux version without SVM-related patches to run on the SCC in a common multicore manner. The light weight hypervisor approach is based upon the idea of a small virtualization layer, which is based on a self-developed kernel. The well established interface to run Linux as a para-virtualized guest will be used to realize the hypervisor. This interface is part of the standard Linux kernel thus no major modification to the guest kernel is needed. With the IO virtualization framework *virtio*, a common way to integrate IO device into the guest system exits.

Additionally, this framework will be used to integrate the eMAC Ethernet interface, which is part of the *sccKit 1.4.0* and already supported by our self-developed kernel. [6] While common processor virtualization aims for providing multiple virtual machines for separated OS instances, we want to use processor virtualization for providing *one* logical but parallel and cache coherent virtual machine for a single OS instance 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.

The Realization of such a hypervisor needs a fast inter-core communication layer, which will be used to manage resources between the micro-kernel. Therefore, an important requirement for the communication layer is the support of asynchronous receiving and sending of request messages because it is not predictable when a kernel need an exclusive access to a resource, which is owned or managed by an other kernel. As a consequence, the synchronous communication library *RCCE* is not suitable for *MetalSVM*. [7] The inter-core communication layer is based on enhancements of our asynchrounous *RCCE* extensions, which we called *iRCCE* [8], [9].

In the future version, the Linux kernel will forward its synchronization requests via the para-virtualization interface to the hypervisor, that has to use SCC-specific features to

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

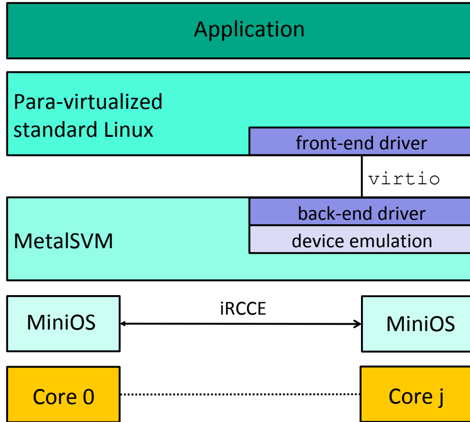

Figure 2. Basic Concept of MetalSVM

reach optimal performance. Therefore, we are in progress to develop an optimized synchronization layer, which will be discussed in the following section.

IV. Synchronization Layer

The SCC has 48 Test-and-Set (T&S) registers to enable a fast locking method. Therefore, each register an atomic bit. Whereas, reading access to target register returns a 0 if its status is unchanged 0, a return value of 1 indicates a previously atomic change from 1 to 0. Furthermore, the status can be changed over a write access.

This behaviour is a limitation compared to atomic functionality provided by shared-memory architectures. Hereby various operations such as *atomic increment* or *compare exchange* at word size using the `LOCK` prefix are provided. In contrast to that flexibility the syntax of T&S registers does not provide a read access without a possible changed value. Another restriction is the limited amount of 48 T&S registers. A synchronization layer for *MetalSVM* will thus require an allocation scheme for the T&S registers. The gory RCCE API provides an interface to the memory mapped T&S registers of the SCC. The function `RCCE_acquire_lock()` implements a simple busy wait loop (spin-lock) with the target register as parameter. The corresponding function `RCCE_release_lock()` releases the granted lock.

Access latencies are dependent of the location of an Unit in Execution (UE) due to the 2D interconnect, as depicted in Figure 3. For this measurement the UE was located at tile 00. A T&S register access is routed through the mesh and therefore affected as well as the MPB accesses. Depending on the distance of an UE to the target, a characteristic latency-rise regarding the 2D mesh and the tiled structure can be examined.

To evaluate the fairness and performance of different spin-lock implementations, it is useful to count the number of concurrent accesses. Whereas such a counter should be accessed with a low latency, due to the lack of atomic counters[3], the preferred location for the value is within the MPB. The counter access can be defined as a critical section and be protected by a spin-lock. Nevertheless, if the location of a UE is fixed we can calculate the expected latencies. Furthermore, a generic `Fetch and` Φ function targeting the MPB has to be filled with `nops` to emulate a symmetric and thus fair accessing scheme. Such a simulated *fair counter* can be used either for debug or for statistical purposes with the single impact to increase the duration of the critical section and thus generate an offset to the latency.

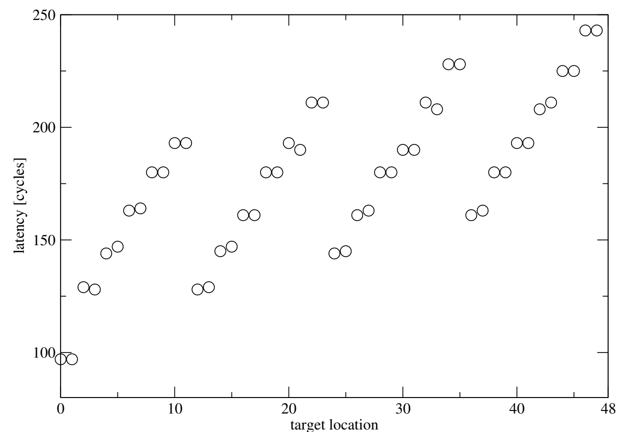

Figure 3. Latencies of uncontended T&S register accesses

V. Spin-Lock Variant: Exponential Backoff

However, by rising contention, a limited scalability results from the access to a single Test-and-Set register, as depicted in Figure 4. Each core contends for a spin-lock a million times for this benchmark. [10] The SCC cores runs with 533 MHz and the mesh interconnect with 800 MHz in our setup.

Expecting such a high contention, it could be useful to expand the simple spin-lock with the goal to minimize the time to acquire a lock. In the following, spin-lock alternatives will be explored with a focus on the implementation of a scaling synchronization layer for the SCC architecture regarding the hardware restrictions.

Programs even based on the gory RCCE API have no possibilty to realize a customized waiting strategy if a lock is occupied. Therefore, a method is needed that returns, just by indicating that a lock has been occupied. We call this method `trylock()`, that is needed to realize a spin-lock with a backoff.

Typically, a random component is introduced to minimize the chance that retries fall into the same point in time. We identified the stdlib `rand()` call to be time consuming with a high variance by using the standard SCC configuration.

[3] at least in sccKit v1.3.0

Therefore, we used a linear congruential generator to generate pseudo random numbers for the backoff. This way, a deterministic amount of time is ensured for the generation and this makes the results comparable to other spinlock methods because the generation is in the critical path and therefore degrades the performance.

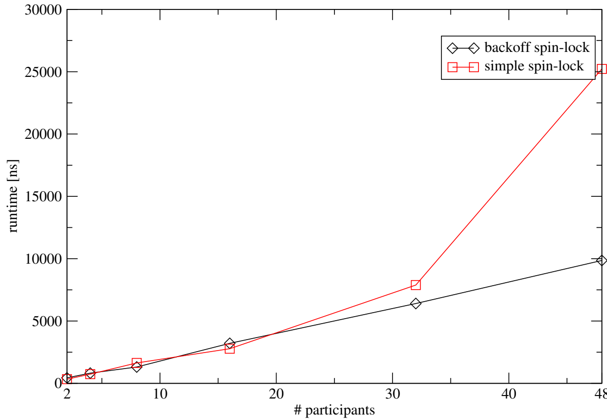

Figure 4. Average Times to acquire a Spin-lock

By using a backoff it is sufficient to use the presented `RCCE_release_lock` method. Thus, in the case of no concurrency a lock competitor needs two T&S register accesses.

VI. Test-and-Set Barrier

Using hardware support for synchronization such as T&S registers, it is evaluated in the following how a linear barrier can be realized. Assuming a communicator size of n, Listing 1 shows an implementation of a barrier using only T&S Register for signaling incoming UE's and wait for release.

```
// linear search for a free lock
  while( !Test_and_Set(step) ) ++step;
  if(step != n-1) {
// wait for release, signal exit
// and wakeup right neighbor
    while(!Test_and_Set(step));
    *(virtual_lockaddress[step]) = 0x0;
    *(virtual_lockaddress[step+1]) = 0x0;
  } else {
// last UE: wakeup first UE and
//          wait for release
    Test_and_Set(step);
    *(virtual_lockaddress[0]) = 0x0;
    while(!Test_and_Set(step));
    *(virtual_lockaddress[step]) = 0x0;
  }
```

Listing 1. Test-and-Set Barrier algorithm in C

A requirement for the algorithm are n available T&S registers with an initial value of 0. However the implementation works without the need for MPB initialization, by not touching it.

Each UE that has entered the barrier tries to grab the first available T&S register, for instance ordered by the numerical count of the located core. After this first step each UE except the last one will spin on the T&S register granted before. Hereby, the contention is minimal and therefore the simple spinlock is sufficient.

Figure 5 shows the runtimes of the presented Barrier to the standard `RCCE_barrier`. A dynamic mapping of linear access ordering and Core-ID by indicating the location of a T&S register is a benefit of the linear cycles. However, a static mapping could provide a better performance by using a tree-based communication pattern [11].

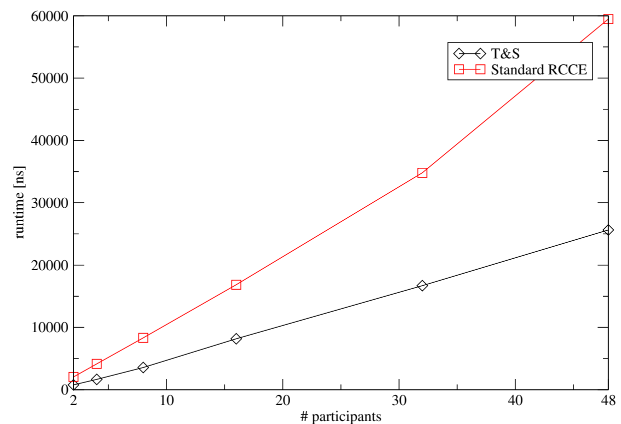

Figure 5. Barrier Runtimes of RCCE Compared to our Approach

VII. Integration of iRCCE into MetalSVM

Due to the lack of non-blocking communication functions within the current RCCE library, we have started to extend RCCE by such asynchronous communication capabilities. In doing so, we aim at avoiding to interfere with the original RCCE functions and therefore we have placed our extensions into an additional library with a separated namespace called iRCCE [9].

An obvious way to realize non-blocking communication functions would be to use an additional thread that processes the communication in background while the main thread returns immediately after the function invocation. Although this approach seems to be quite convenient, it is not applicable in *bare-metal* environments where a program runs without any operating system and thread support. Since we have planed from the start to integrate iRCCE into *MetalSVM*, we had to waive this thread-based approach. Therefore, we have followed another approach where the application (or moreover the *MetalSVM* kernel) must drive on the communication progress by itself. For this purpose, a non-blocking communication function returns a *request handle*, which can be used to trigger its progress by means of additional `push`, `test` or `wait` functions [8].

In the context of *MetalSVM*, these progress functions are called by entering the kernel via interrupt, exception or system call. Additionally, *MetalSVM* checks at this point if the kernel has recently received a message. Therefore, all messages begin with a header, which specifies the message type and the payload size. The definition of a message type is important because this layer will be used to send messages between the cores from the SVM system and also from the TCP/IP stack. Our self-developed kernel, that builds the base of *MetalSVM*, supports LwIP [4] as a lightweight TCP/IP stack. The TCP/IP stack will be used to collect status information from the kernels and to feature the support of administration tools in the future.

The maximum delay between sending and receiving a message header is as large as a time slice, because at least after one slice a (timer) interrupt will trigger, which checks for incoming messages. Additionally, *MetalSVM* allows the sender to trigger a remote interrupt on the side of the receiver in order to reduce the delay. However, this creates an additional overhead because the calculation on the remote core could be unnecessarily interrupted. Further analysis will be done to decide which communication model (with or without triggering an interrupt) will be the best choice for *MetalSVM*.

VIII. Conclusion and Outlook

In this paper, we have presented our first steps to design and implement a low-latency communication and synchronization layer for *MetalSVM*, a shared virtual memory system for the SCC. We have pointed out the demand especially for fast and hardware-based low-level synchronization primitives and we have shown how such primitives can be realized on the SCC be means of its special hardware features. In this context, we have presented an optimized barrier algorithm that utilizes the SCC's Test-and-Set registers and we have shown that this algorithm outperforms the RCCE-related approach of using the SCC's on-die message-passing buffers for this purpose. We have also shown that especially in case of highly contended locks an exponential back-off algorithm can lead to an improved scalability compared to the straight forward approach of the common RCCE library. Moreover, we believe that we are even able to further improve our low-level locking mechanism by applying a *tree-like* access pattern to the Test-and-Set registers. Although such a tree-based locking algorithm would cause an addition overhead in the order of $O(log(n))$ in the non-contented case, we think that this approach will play out its strength especially in highly contended situations, as they will usually occur in the context of *MetalSVM*.

Acknowledgment

The research and development was 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.

[4] http://savannah.nongnu.org/projects/lwip/

References

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

[2] P. K. A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the USENIX Winter 1994 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1994, pp. 10–10.

[3] C. Terboven, D. an Mey, D. Schmidl, and M. Wagner, "First experiences with intel cluster openmp," in *OpenMP in a New Era of Parallelism*, ser. Lecture Notes in Computer Science, R. Eigenmann and B. de Supinski, Eds. Springer Berlin / Heidelberg, 2008, vol. 5004, pp. 48–59.

[4] 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.

[5] D. Schmidl, C. Terboven, D. an Mey, and M. Bücker, "Binding Nested OpenMP Programs on Hierarchical Memory Architectures," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, 6th International Workshop on OpenMP (IWOMP 2010)*, ser. Lecture Notes in Computer Science, Tsukuba, Japan, June 2010.

[6] *The SccKit 1.4.0 Users Guide*, Intel Labs, March 2011, Revision 0.92.

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

[8] C. Clauss, S. Lankes, J. Galowicz, and T. Bemmerl, *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer – User Manual*, Chair for Operating Systems, RWTH Aachen University, December 2010, Users' Guide and API Manual.

[9] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor (accepted for publication)," 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, accepted for publication.

[10] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6 –16, January 1990.

[11] N. S. Arenstorf and H. F. Jordan, "Comparing barrier algorithms* 1," *Parallel Computing*, vol. 12, no. 2, pp. 157–170, 1989.