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.

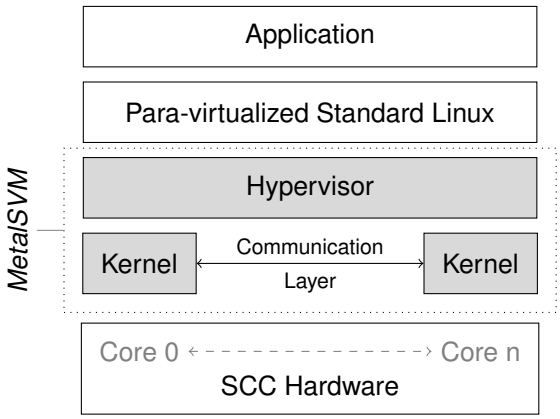


Fig. 1: Concept and Design of MetalSVM

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.

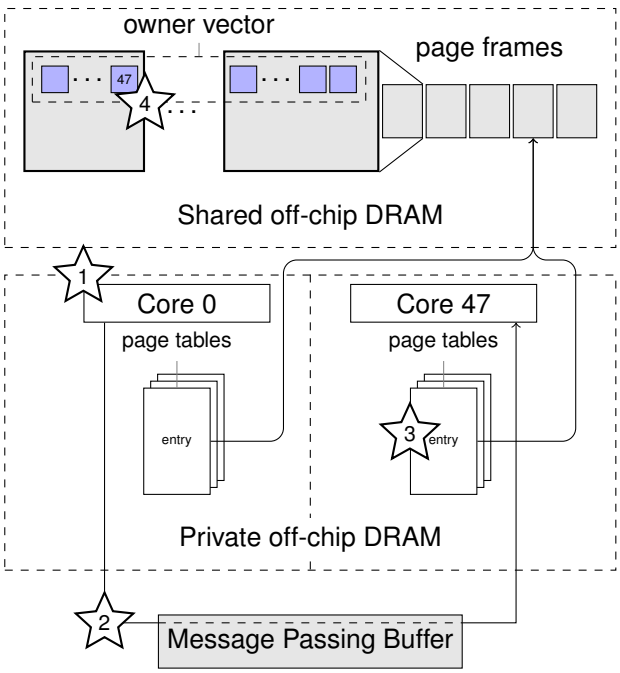


Fig. 2: Concept and design of the SVM subsystem

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

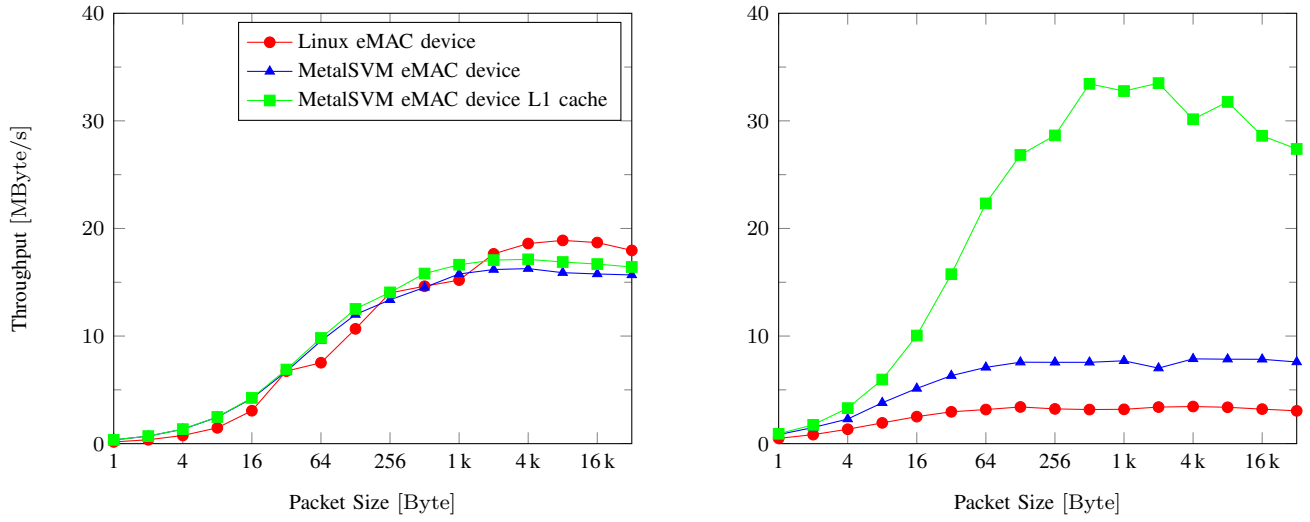
After this procedure Core 0 is the new owner and hereby has full access permissions.

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.

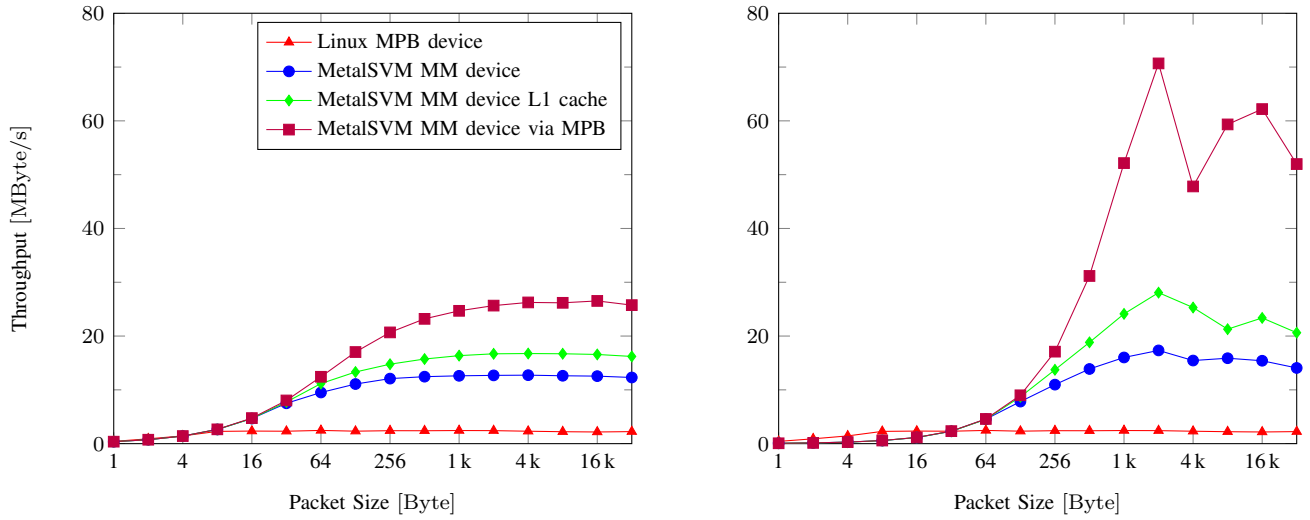
A. Benchmark Results

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.

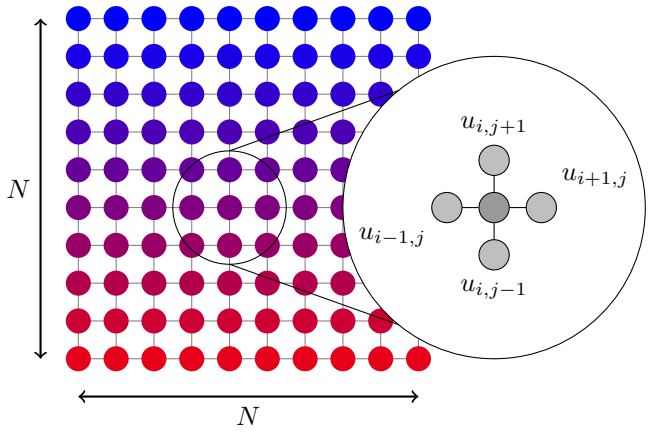


Fig. 5: Heat Distribution Problem

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.

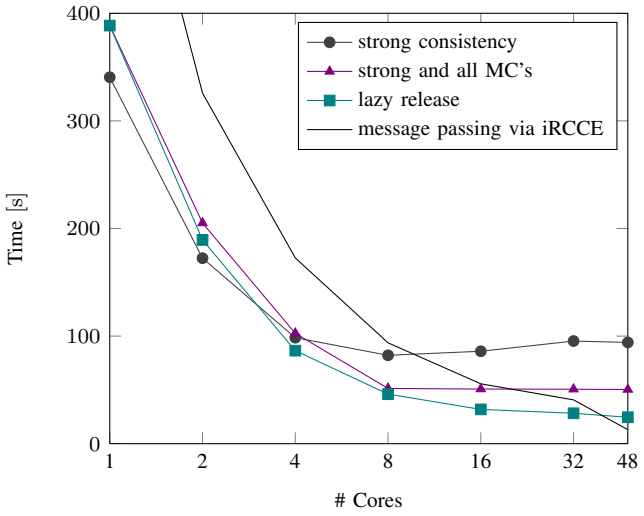


Fig. 6: Laplace Runtimes

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.