Revisiting Shared Virtual Memory Systems for Non-Coherent Memory-Coupled Cores

Stefan Lankes
Chair for Operating Systems
RWTH Aachen University
Kopernikusstr. 16
52056 Aachen, Germany
lankes@lfbs.rwth-aachen.de

Pablo Reble
Chair for Operating Systems
RWTH Aachen University
Kopernikusstr. 16
52056 Aachen, Germany
reble@lfbs.rwth-aachen.de

Carsten Clauss
Chair for Operating Systems
RWTH Aachen University
Kopernikusstr. 16
52056 Aachen, Germany
clauss@lfbs.rwth-aachen.de

Oliver Sinnen
University of Auckland
Private Bag 92019
Auckland 1142, New Zealand
o.sinnen@auckland.ac.nz

ABSTRACT

The growing number of cores per chip implies an increasing chip complexity, especially with respect to hardware-implemented cache coherence protocols. An 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) is a recent research processor of such architectures.

This paper presents an approach to deal with the missing cache coherence protocol by using a software managed cache coherence system, which is based on the well-established concept of a shared virtual memory (SVM) management system. Through SCC's unique features like a new memory type, which is directly integrated on the processor die, new and capable options exist to realize an SVM system. The convincing performance results presented in this paper show that nearly forgotten concepts will become attractive again for future many-core systems.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; C.1.4 [**Processor Architectures**]: Parallel Architectures

Keywords

Shared Virtual Memory, Single-chip Cloud Computer, Non-Coherent Memory-Coupled Cores, Many-Core Architectures

1. INTRODUCTION

PMAM 2012 February 26, 2012 New Orleans LA, USA

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. Furthermore, cache coherence protocols increase the also hardware and performance overhead. 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 (memory-coupled) Cluster-on-Chip architecture.

The Single-chip Cloud Computer (SCC) experimental processor [8] 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 into private memory sections. 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. Yet, this does not exploit the capabilities and potential of such a system. In contrast to classical message-passing architectures, the memory space and banks are shared, there are hardware mechanisms for fast inter-processor synchronisation and the network is fast and tightly coupled.

Many applications strongly benefit from using the shared memory programming model. The shared virtual memory (SVM) approaches have been intensively investigated before, but never had sustained success due to performance issues. However, on the newly emerged architecture of cluster-on-chip, with its described advantages, SVMs might experience a renaissance.

The project *MetalSVM*–the realization of an SCC-related shared virtual memory management system–aims to investigate the potential of SVMs for non-coherent memory-coupled multicore systems. It is implemented in terms of a bare-metal hypervisor and 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 through the utilization of special SCC-related features such as its on-die Message-Passing Buffers (MPB). In order to offer maximum 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 discusses the design of *MetalSVM* and its SVM system. It focuses on the SVM system and its consistency and affinity solutions. We discuss the design of our mailbox system for the fast interprocessor synchronisation and communication. Other components and technologies of *MetalSVM* have been addressed in more detail in preliminary works [21, 14]. The first experimental evaluations of *MetalSVM* demonstrate very good performance, and hold promise for SVMs on non-coherent memory-coupled multicore systems.

The rest of the paper is structured as follows. In Section 2 we refer to our previous work on the SCC and summarize related work regarding SVM systems. We present a detailed description of the Single-chip Cloud Computer in Section 3. The following Section 4 explains the design of *MetalSVM* and our small self-developed operating system kernel that builds the base of *MetalSVM*. The realization of an inter-kernel communication layer and the prototype of our SVM system are presented in Section 5 and Section 6. Section 7 explains the benchmarks used for evaluating our inter-kernel communication layer and our SVM system and presents the respective performance results. The final Section 8 summarizes this paper and gives an outlook to our next research goals.

2. RELATED WORK

Since the *IVY* project [15], a lot of work has been done on SVM systems. TreadMarks [12] is an important SVM system, which also builds the basis of Intel's Cluster OpenMP. However, they are based on traditional message-passing oriented networks or they used a RDMA (Remote Data Memory Access) engines, which allowing access to remote memory locations via the network without any involvement of the receiver. However, the setup costs to program the RDMA engine are high and increase the overhead of using a SVM system.

The *Scalable Coherent Interface* (SCI) [7] belongs to the memory-mapped networks and offers a transparent read-/write access to remote memory. The SCI standard defines also a cache coherency protocol, but the PCI-SCI adapter cards do not support this feature. Several research projects used SCI-based PC clusters, which possessed a similar characteristic like the SCC. Both systems consist of several processing units which are able to communicate transparently over shared memory regions without the support of cache-coherency.

Several projects realized an SVM system on top of an SCI cluster. *NOA* [18] used SCI as fast message-passing interconnect and did not exploit capabilities of transparent remote read/write memory access. Paas et al. have developed in [20, 23] an SVM system for Windows and Unix clusters, called *SVMlib*, which stores write notices and related changes in the global memory to realize a *Lazy Release Consistency* [11] model.

Both approaches are implemented at user level, which decreases the usability. *SCI-VM* [24] enables the caches and realizes the cache flushing by software. However, *SCI-VM* used only a static memory mapping in which any memory location within the global address space is accessible at all time. In contrast to that, SciFS [2] combines SCI memory mappings and techniques like migration and replication.

For a simple and transparent access of the shared memory, SVM systems can also be integrated into virtual machines, so that common operating systems and development environments can easily be applied without changes. An example for such a hypervisor-based SVM system is *vNUMA* [3] that has been implemented on the Intel Itanium processor architecture. In [6] one founder of *vNUMA* proposed to extend this concept for Many-Core Chips. For x86-based compute clusters, the so-called *vSMP* architecture developed by ScaleMP[1] allows for cluster-wide cache-coherent memory sharing. This architecture implements a virtualization layer underneath the OS that handles distributed memory accesses via InfiniBand-based communication. These approaches are similar in some respects to our hypervisor approach because both implement the SVM system in an additional virtualization layer between the hardware and the operating system.

The main difference between these approaches is that *vSMP* and *vNUMA* explicitly use message-passing between the cluster nodes to transfer the content of the page frames, whereas our SVM system can cope with direct access to these page frames. 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. This feature will help to overcome a drawback of other hypervisor-based approaches regarding fine granular operations. A recent evaluation [22] of ScaleMP's *vSMP* with synthetic kernel benchmarks as well as with real-world applications has shown that *vSMP* architecture can stand the test if its distinct NUMA characteristic is taken into account. Also, we developed in [1] optimized applications for *vSMP* and reached excellent performance results. By using 104 cores on cluster of 13 nodes, our applications reach speedups up to 80. Nevertheless, the evaluation has also shown that fine granular operations like synchronization is the big drawback of such architectures. Our aim is to avoid this shortcoming by using the distinguished capabilities of transparent remote read/write memory on the SCC.

In [14], we present a first prototype of a SVM system for the SCC. However, this design underachieves the potential of the SCC's on-die memory and based more on polling on the off-die memory. Therefore, this approach runs against the so-called memory wall and doesn't scale very well for certain applications. The basic concepts of the MetalSVM's inter-kernel and communication layer are sketched in [21]. In contrast to the current work, the communication layer was not event triggered and consequently not asynchronous, which is needed for our new design and improvements of the SVM system.

3. THE INTEL SINGLE-CHIP CLOUD COMPUTER (SCC)

The Single-chip Cloud Computer (SCC) experimental processor [8] is a *concept vehicle* created by Intel Labs as a

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

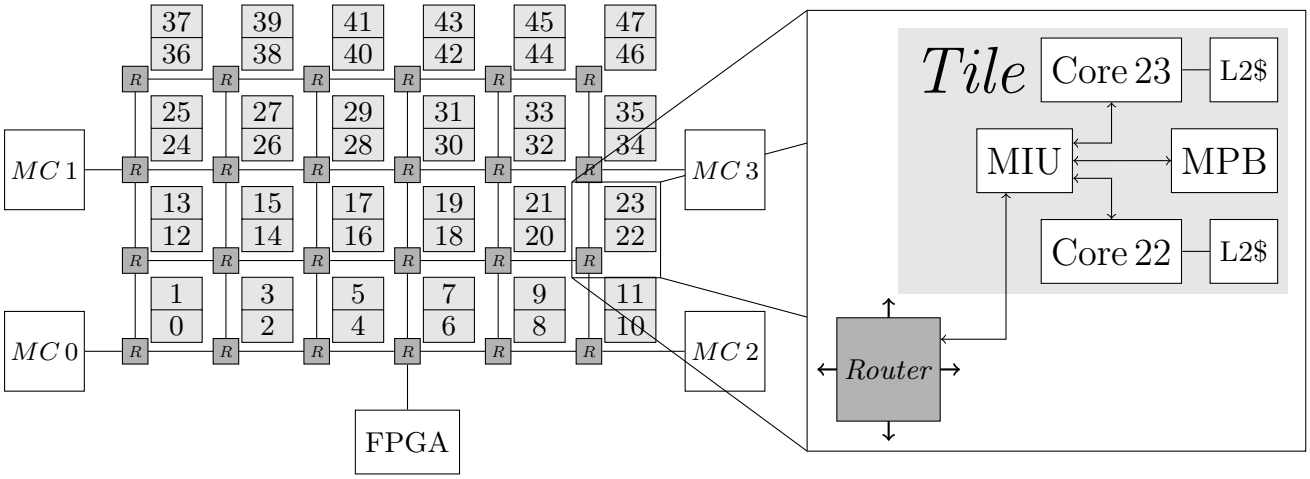

Figure 1: Top Level Block Diagram of the SCC Architecture (based on [17])

platform for many-core software research, which consists of 48 P54C cores. The P54C belongs to the Pentium (I) family and owns, in contrast to its predecessor P5, an on-chip *Advanced Programmable Interrupt Controller* (APIC). Although a more modern core like the *Atom* processor may appear more applicable for performance reasons, the focus of the SCC architecture is to analyze programming models for the future many-core era. Therefore, a large number of cores is in this case more important than the speed of each individual core.

The 48 cores are arranged in a 6×4 on-die mesh of tiles with two cores per tile, which is shown in Figure 1. The SCC chip possesses four on-die memory controllers for addressing the external main memory. The supported DRAM type is *DDR3-800*. The frequencies of the cores and the routers of the mesh are configurable. The routers support frequencies of 800 MHz and 1.6 GHz, while the cores use a frequency between 100 and 800 MHz. The power consumption of the full chip depends on the configuration (frequency and voltage of the mesh and cores) and is between 25 and 125 W.

Each core has 16 kByte L1 and 256 kByte L2 cache. Additionally, each core has 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 because they are intended to improve the performance of message-passing based programming models. A table showing the approximate latencies for L2, MPB and DDR3 read accesses can be found in [9].

The SCC's architecture does not provide any cache coherency between the cores, but 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.

To avoid any cache coherency problems, the SCC divides the off-die memory in private regions for each core and one shared region for all cores. The owners of the private regions have exclusive access to their memory areas. Because of the exclusive usage of these regions, the cache is enabled here per default.

Normally, these regions are used to start a Linux kernel on each core. Therefore, the SCC is able to start 48 Linux instances. These instances are able to use the shared region to share data between the cores. Per default, the cache for this shared region is disabled. The software developers have the possibility to configure the region sizes and to change the cache behaviors. However, they have to consider that the SCC realizes no cache coherency between the cores. The logical view of the hardware is summarized in Figure 2.

Intel Labs extended the P54C instruction set architecture (ISA) by a new instruction `CL1INVMB`. This instruction invalidates the L1 cache entries of pages which are marked in the page tables as `MPBT`, a new memory type. Furthermore, accesses to this new memory type bypass the L2 cache. By default message-passing buffer entries are tagged with this new type. 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 the write combine buffer, which is also enabled via `MPBT` flag in the page table. Its primary intention is to accelerate the data transfer for message-passing between the cores [8]. Therefore, the write combine buffer extends the write through strategy from byte granularity to cache-line granularity. If the buffer, which holds exactly one cache-line, is full or a write operation touches another cache-line (miss), the buffered data is transferred to the next stage in memory hierarchy. For our SVM system, the combine of write through data is extremely useful to increase the bandwidth.

We limit our current experiments with an SVM system prototype to support only the L1 cache (and not the L2 cache) for shared regions of memory. To control the write strategy for cached data, each page table entry contains a bit that the memory management of *MetalSVM* sets for shared pages in order to enable a *Write-Through* strategy for these pages.

Obviously, a drawback of this solution is a significantly smaller amount of cache in use for shared regions. But when waiving the use of Level 2 cache for shared memory regions, a major advantage arising is the possibility to tag SVM related data. Thus, a selective invalidate of cached data via

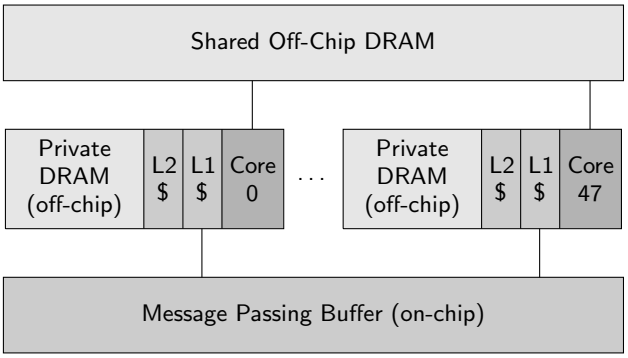

Figure 2: Logical View of the Hardware (based on [17])

`CL1INVMB` becomes possible. The flush of cached data is enabled by just flushing the write combine buffer due to the fact that the current SVM system uses the *Write-Through* strategy. The P54C architecture features an external Level 2 cache without the possibility to flush it with hardware support. A flush routine could be realized by software but it turned out to be costly.

4. STRUCTURE 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 3 illustrates the design approach of *MetalSVM*.

A major advantage of our approach, as introduced in [21, 14], is the absent binding of *MetalSVM* to a certain version of Linux, because *integrating* would for example mean *patching* the kernel. The lightweight hypervisor is based upon the idea of a small virtualization layer based on a monolithic-kernel developed from scratch by the authors. A well-established interface to run Linux as para-virtualized guest which is part of the standard Linux kernel is used to realize our hypervisor. Consequently, no modifications to the Linux kernel are needed.

The aim of common processor virtualization is to provide multiple virtual machines for separated OS instances.

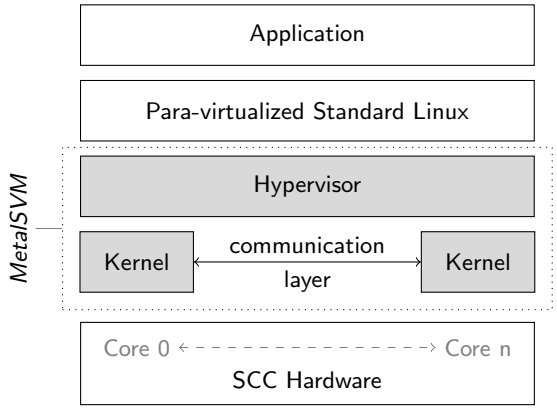

Figure 3: Concept and Design of MetalSVM [14]

We want to use processor virtualization that provides *one* logical but parallel and cache coherent virtual machine for a single OS instance, that is to say 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. Currently, the a prototype of the hypervisor exists and we are able to boot Linux on one core.

In this paper, we present the integration of an SVM system in our self-developed kernel, which will build the base of the hypervisor and is needed to boot Linux on several core.

5. THE INTER-KERNEL COMMUNICATION LAYER

The realization of the hypervisor requires a fast inter-core communication layer, which will be used to manage resources between the kernels. Intel provides a customized programming library for the SCC, called RCCE [16, 17], that offers baremetal support and could be an attractive library for *MetalSVM*. RCCE allows for using the message-passing as well as the shared-memory programming model. However, RCCE supports only uncacheable memory regions as shared-memory. The message-passing part, that is in turn based on simpler one-sided communication mechanisms (`RCCE_put`/`RCCE_get`), offers two-sided point-to-point communication functions (`RCCE_send` / `RCCE_recv`) as well as a set of collective communication operations (`RCCE_barrier`, `RCCE_bcast`, ...). In doing so, and this is important, all of these functions utilize the SCC's MPBs for the message transfers.

However, the semantics of RCCE's send and receive functions (as well as the semantics of the collective operations) is *blocking* and *synchronous*. On the receiver side, the term blocking implies that the respective receive call does not return until the complete message has been transferred to the receive buffer. An important requirement to *MetalSVM*'s 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. Our non-blocking extensions to RCCE, called iRCCE [4, 5], does not solve this problem because it requires that sender and receiver are working coevally in a non-blocking but synchronizing manner on the communication progress. Hence, iRCCE still realizes a synchronous communication model, which is not suitable for *MetalSVM*.

Therefore, an *asynchronous mailbox system* has become part of *MetalSVM's* communication layer, partly outlined in [14], that extends iRCCE [4] to enable an event driven and fast asynchronous communication path between the SCC cores. A logical view of this mailbox system is shown in Figure 4. 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 $48 * 32\,\text{Bytes} = 1.5\,\text{kByte}$ 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 per core. If the the buffer of a specific core is full (for example buffer 1 in Figure 4) then the core is (busy) waiting until the receiver has consumed the mail. The access to a mailbox is restricted for the receiver, which is only allowed to read data and toggle a send flag that the mailbox contains. A sender with the

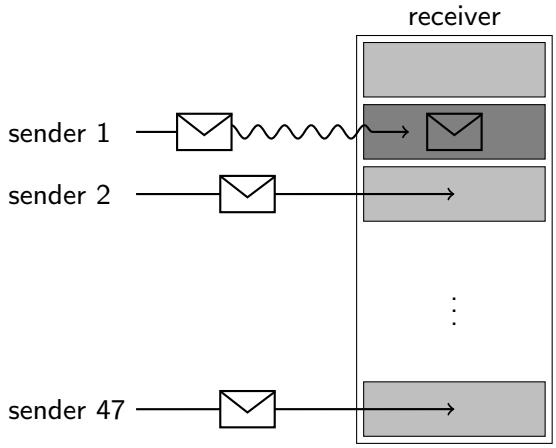

Figure 4: Logical view of the Mailbox System

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.

At every interrupt (e.g. timer interrupt, software interrupt triggered by a system call), the kernel checks all receiving buffers for incoming messages. This approach functions very well. However, the costs[2] for checking mails increases with the number of cores. The second disadvantage of this approach is the dependency on the local interrupts. The time slice between two interrupts could be very large, which decrease the performance of the mailbox system.

Since *sccKit 1.4.0* the system FPGA holds a Global Interrupt Controller (GIC) [10], which offers among others the possibility to trigger an *Inter-Processor Interrupt* (IPI). In addition to the former method to generate an IPI the possibility arises to indirectly generate an IPI using the GIC. By using the GIC, this IPI can be used to obtain the information by which core it has been raised.

By sending a mail from one core to another core, *MetalSVM* is able to send an IPI afterwards. If such an interrupt arrives, the interrupt handler checks only the receive buffers of that core which triggered the remote interrupt. This avoids unneeded checks and reduces the time between incoming and processing of mails.

6. AN SVM SYSTEM FOR NON-COHERENT MEMORY-COUPLED CORES

There exist several memory consistency models for SVM systems. The differences between these memory models is the point of time when the modifications to the memory of one core will be recognized by the other cores. Currently, we support two memory models in our SVM system: *Lazy Release Consistency*, which is original presented by Kehler et al. in [11], and a stronger memory model, which we call *Strong Memory Consistency Model*.

[2]Currently, the mailbox system requires 100 processor cycles to check one receive buffer.

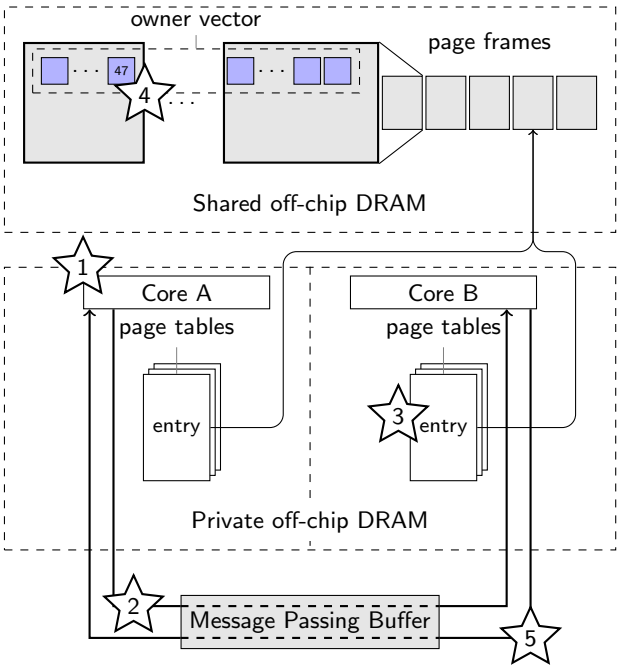

Figure 5: Concept and design of the SVM subsystem

6.1 Strong Memory Model

The motivation for the realization of the *Strong Memory Consistency Model* is that many legacy codes implicitly assume a stronger memory model than *Lazy Release Consistency* can provide. Our SVM system provides the function `svm_alloc` to allocate an amount of bytes in a cached shared-memory region. At each point in time only one owner of a page exists, which is allowed to read from or write to it. This ownership is registered as an element of a dedicated *owner vector*, which is located in the off-die memory (cf. Figure 5). As one can see, 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[3] and invalidates via `CL1INVMB` the cache entries 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. Finally, a mail is sent back to the requesting core to signalize that the core is able to continue the calculation. In contrast to our first approach [14], the requesting core does not need to poll on the ownership vector for changes of the ownership, which unloads the memory bus and avoids the memory wall.

Obviously, the performance of the mailbox system has a direct impact on the performance of the SVM system. Figure 5 exemplarily illustrates the situation example where an SVM related page fault occurs at Core A involving Core B. Following steps have to be performed, which are tagged as star symbols in Figure 5.

1. A page fault occurs at Core A

[3]In this scenario this is just the flushing of the write combining buffer.

2. Core A looks the ownership up in the owner vector and sends a mail to the owner (Core B)

3. Core B flushes its cache and changes the page table entry

4. Core B changes the ownership

5. By sending a mail back to Core A, Core B signalizes the changing of the ownership.

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

6.2 Lazy Release Consistency

The *Lazy Release Consistency Model* assumes that every access to shared data is protected by a lock. To get correct results, it is sufficient that the changes within a critical section will be seen by all cores just after releasing the lock. This memory consistency model is much weaker than traditional cache coherence protocols like *MESI*, which invalidates directly shared data on all cores if one core modifies this data.

Our SVM system is based on the *Write-Through* cache strategy, which can be enabled per page frame by a special flag in the page table. In our context, we need to flush the write combining buffer only, to be sure that the modifications are written down to the memory. For the realization of the *Lazy Release Consistency Model* we extend our synchronization primitives. By entering the critical section, the cache entries are invalidated via `CL1INVMB` for all page frames, which are managed by the SVM system. Likewise, by leaving the critical section, the SVM system flushes the write combining buffer. In contrast to the original definition of *Lazy Release Consistency Model*, our system writes down also all modifications to the memory which are not concerned by the critical section. Yet the advantage is that our approach generates nearly no overhead.

6.3 Affinity-on-First-Touch

Our first steps to realize an SVM system [14] used specialized memory allocation functions, which gave the developers the possibility to define explicitly the mapping between the data and the four memory controllers. Now, we use the well-known *Affinity-On-First-Touch* mechanism to map the data. Like on traditional SMP systems, our memory allocation function reserves only a region in the virtual address space. Thus, no physical page frames are initially mapped behind the virtual address space. Only when the first access to a page occurs, a page fault is triggered and the related page fault handler will afterwards map a physical page frame at the specific virtual address. On NUMA systems, this mechanism is used to bind data to specific memory controllers. In this context, the page fault handler allocates the page frame nearest to the current node. Hence, software developers are able to use this feature to develop NUMA-aware applications. In order to get optimal performance results, they have to guarantee that the initialization routine uses the same access pattern like the later computation algorithms.

In our SVM system, the page tables are located in the private memory and, consequently, each core possesses its own version of the page tables. Therefore, the first access triggers a page fault on all cores but only the first to access core has to allocate the page frame. To realize *Affinity-On-First-Touch*, we use the SCC's on-die memory partly as *scratch pad*. Each shared page has a 16 bit representation in the scratch pad. With this representation, the SVM system is able to build the physical address from the virtual address. On each core, the page fault handler looks into this scratch pad whether already another core has allocated a page frame and thus pushed the representation into the scratch pad. If not, the current core allocates a page frame near to the core, maps it into the address space and finally pushes the representation to the scratch pad. To avoid races, the accesses to the scratch pad are protected by a lock, which is realized by the SCC-specific *Test-And-Set-Registers*.

The small amount of on-die memory limits the shared-memory of our SVM system to 256 MByte. To increase the memory size, we can relocate the scratch pad into the off-die memory. However, this increases the number of memory accesses, which in turn decreases the performance of our system.

Software developers are able to use the same optimization strategies, which are used for NUMA-aware applications. In the same way, the SVM system allocates page frames near to the core, which has initialized the data.

6.4 Read-Only Memory Regions

Applications often use many memory regions without a single write access. After the initialization, these regions could be defined as read-only memory regions by using system calls like `mprotect`. One advantage of this technique is that an undesired write access to these regions triggers a page fault. This reduces the time of debugging, because software developers detect wrong accesses by their first occurrence and not by a wrong final result.

In context of our SVM system, an absence of modifications on shared data enables also the safe use of the L2 caches for these regions. Therefore, we define a collective system call which protects a memory region against any write access. The system call clears the `read/write bit` in the page table, which protects a page frame from write accesses. Furthermore, the system call clears the `MPBT bit` in the page tables. In contrast to writeable pages, the L2 cache becomes enabled for these read-only regions.

7. PERFORMANCE EVALUATION

All benchmark results in this section are measured with our test platform, which has been configured with a core frequency of 533 MHz, a memory, and a mesh frequency of 800 MHz. If we compare our *MetalSVM* results with the SCC Linux distribution by Intel, we use the default kernel[4] of `sccKit 1.4.1`. All *MetalSVM* benchmarks were compiled with the `gcc 3.4.5`, whereas on the Linux system the `icc 8.1` was used. As optimization level `-O2 -march=i586` for the `gcc` and `-O3 -mcpu=pentium` was used for the `icc` compiler. Both compilers belong to the Intel SCC software distribution.

7.1 The Inter-Kernel Communication Layer

In this section we present a performance evaluation of our mailbox system, which acts as a backbone for our SVM system. Thereby, the mailbox system has a huge impact on the overall performance. With the common *Ping-Pong* bench-

[4] 2.6.38.3-jbrummer

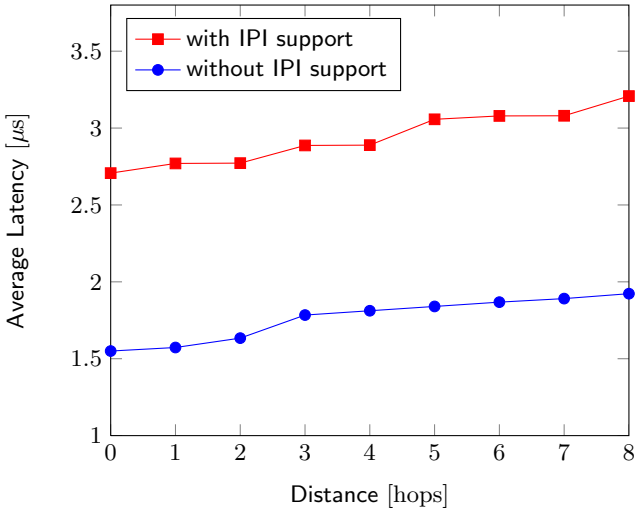

Figure 6: Average latency according to the distance

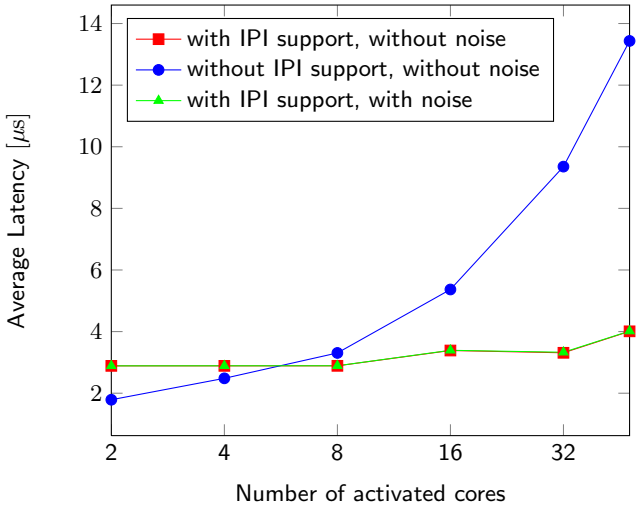

Figure 7: Average latency between core 0 and 30

mark, we determined the impact on the distance between participants within the mesh. As a result, Figure 6 shows the latency according to the distance. The resulting values have to be understood as half round-trip times, hence the elapsed time for sending a mail and handling on the receiver's side.

It can be pointed out that the average latency increases linear according to the distance with a very low gradient. The implementation without IPI support (—●—) has to check all receive buffers on every interrupt as well as in the idle loop. Here, the benchmark activates only two cores. Therefore, only one receive buffer per core has to be checked for incoming messages and the number of checks is identical for both implementations. The result is a significantly lower latency, compared to the event triggered approach. The average latency of the approach with IPI support (—■—) is increased by the disruption of incoming interrupts. However, the gap is very low and shows the excellent interrupt handling of our self-developed kernel.

Figure 7 shows the average latency between the core by increasing the number of activated cores. Curve —●— represents the implementation without IPI support. In this case, only the cores 0 and 30 with a distance of 5 hops are involved in the Ping-Pong benchmark and all other cores are in an idle loop. By a rising number of cores, the average latency increases because more buffers have to be checked for incoming messages.

A similar implementation with IPI support (—■—) has a nearly constant average latency. Thus, the receiver can use the IPI to determine which buffer has to be checked. Curve —▲— represents the implementation with IPI support, whereas the remaining activated cores permanently interact among themselves by sending mails.The average latency is on a similar level for up to 48 cores, compared to the benchmark without background noise (—■—). This result shows the excellent behavior of our mailbox system.

7.2 The SVM System

First, we use a synthetic low-level benchmark for the evaluation of core performance characteristics of the SVM system. Second, a real world example – shared memory version of a two-dimensional Laplace application – has been applied to the SVM system. This includes a comparison of the performance to a message passing variant, which is based on iRCCE [4].

7.2.1 Synthetic Benchmarks

We used a synthetic benchmark to compare in detail the Lazy Release Consistency and the Strong Memory Model of our SVM system. The further described benchmark has been executed on core 0 and 30 and the results are listed in Table 1.

As a first step, for the allocation of 4 MByte of memory the synthetic benchmark calls a collective function. Thus, the allocated memory is managed by our SVM system. The underlying method reserves only virtual address space, physical memory will be allocated and mapped at first touch.

Therefore, regardless of what memory model is used, the time to reserve the memory region, shown in the first row of Table 1, is constant and very low.

	Strong	**Lazy Release**
allocation of 4 MByte	741.0 μs	741.0 μs
physical allocation of a page frame	112.301 μs	112.296 μs
mapping of a page frame	10.198 μs	2.418 μs
retrieve the access permission of a page frame	8.990 μs	—

Table 1: Average Overhead by using the SVM system

As a second step, core 0 initializes the first four bytes of every page and thereby consequently allocates the physical memory. The average time to allocate a page frame is shown in the second row of Table 1. Here, values are independent from the used memory model because both models are based on the same memory allocation strategy.

Next, core 30 writes to the first four bytes of every page. The pages are already allocated by core 0. Within the page fault handler, the SVM system subsequently looks up and

restores the value passed by core 0 via the scratch pad memory. Now, the calculation can be continued regarding the Lazy Release Model.

In contrast to this memory type, the Strong Memory Model has to retrieve the access permissions from the page owner. Therefore, the average time to map an already allocated page, shown in the third row of Table 1, is clearly lower for the Lazy Release Model compared to the Strong Memory Model.

Finally, core 0 resets the first four bytes of every page. The pages are already allocated and mapped at all cores. However by using the Strong Memory Model, the core 0 has to retrieve the access permissions. The last row of Table 1 reveals that the overhead of our approach which amounts $9\,\mu s$ is extremely low and indicates an excellent behavior.

7.2.2 *The Two-Dimensional Laplace Problem*

Here, a classical synchronous iteration program example is analyzed for the demonstration of our SVM system. For instance, the heat distribution of a square metal sheet with known temperatures at its edges represents the well known two-dimensional Laplace problem. Figure 8 illustrates the further described method.

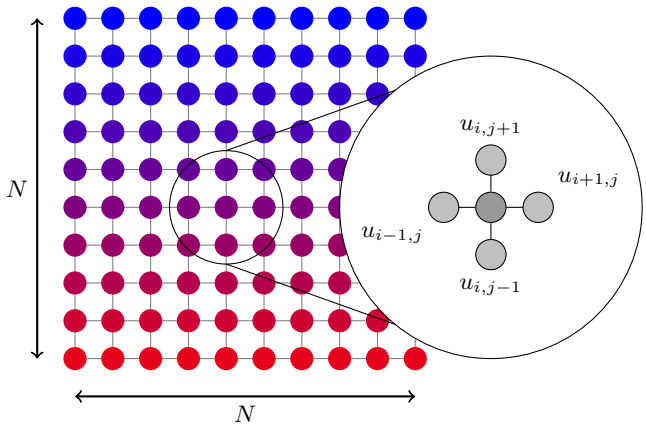

Figure 8: Heat Distribution Problem

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 collective memory allocation function is used 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. 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 the *Strong Memory Model*, or explicitly, regarding the *Lazy Release Consistency*. The current version of *MetalSVM* supports both as described in Section 6.

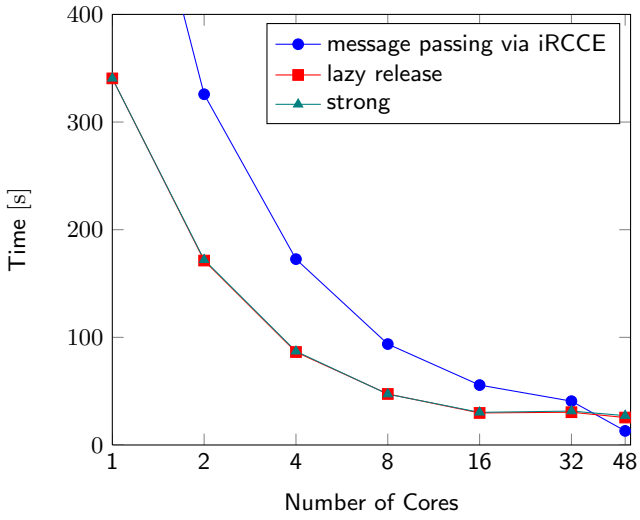

Figure 9: Runtimes of the Laplace Benchmark

Figure 9 shows benchmark results of the previously described application for an increasing number of cores on the SCC platform. Curve —●— depicts terms of a message passing laplace variant based on iRCCE [4] under Linux, which uses a non-blocking behavior to exchange rows after each iteration. The Curves —▲— and —■— represent the performance measurements of the Strong and the Lazy Release Consistency model of our SVM system. Both curves are nearly identical.

In this example, the benchmark iterates 5000 times. In the case of the Strong Memory Model, each iteration triggers two page faults. Consequently, the overhead is about $5000 \times 2 \times 9\,\mu s = 90\,ms$, whereas $9\,\mu s$ is the estimated time to retrieve access on a page frame (see Table 1). In fact, this overhead is negligible, compared to the total runtime of the synthetic application.

The P54C cores of the SCC are not able to update the cache entries on a write miss. This implies that write accesses to the matrix u^{k+1} is like write accesses to uncachable memory region. Per default the write combining buffers are not used by enabling the `MBPT` flag. Therefore, the iRCCE version could not benefit from this feature and is slower up to a number of 32 cores compared to the SVM versions, which enable the write combining buffers. The reason for the super linear speed up of the message passing variant of the parallel application in the interval 32 to 48 cores is the use of the Level 2 cache. For more than 32 cores the required rows from matrix u^k fit into the L2 cache. The P54C cores update cache entries on read miss only. Unlike the iRCCE version, the SVM systems sacrifices the use of the L2 cache for these regions for the use of the write combining buffer. Therefore, the shared memory variant can not profit from these L2 caching effects.

The chosen application benchmark proves the excellent behavior of our SVM system, which thereby builds an ideal base for our *MetalSVM* project.

8. CONCLUSIONS AND OUTLOOK

This paper has revisited the technique of SVMs for the emerging architecture of cluster-on-chip. The distinguishing feature of such systems, like the here considered SCC, are the many cores coupled through non-coherent shared memory. By using the special features of the investigated SCC processor, the known concept of SVMs can become very attractive. Our SVM system *MetalSVM* is based on a hypervisor-based approach, formed by bare-metal kernels running on the cores. In terms of the SVM, we have presented the initial design and implementation of Lazy Release and the Strong Memory Model, which has been integrated into *MetalSVM*. The basic approach is based on a mailbox system with a low-latency inter-kernel communication layer. The benchmark results of the communication layer and the SVM system prototype are promising. In fact, the overhead of the Strong Release Consistency compared to the Lazy Release Consistency Model is tolerable.

In the future, we will investigate other, weaker memory models, to achieve the best performance for our bare-metal hypervisor. We plan to use our experience [13] from the design of kernel extensions for NUMA systems to reach a more dynamic memory distribution strategy like *Affinity-on-Next-Touch*, which has been firstly proposed by Noordergraaf and van der Pas in [19].

9. ACKNOWLEDGEMENTS

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

10. REFERENCES

[1] N. Berr, D. Schmidl, J. Göbbert, S. Lankes, D. an Mey, T. Bemmerl, and C. Bischof. Trajectory-Search on ScaleMP's vSMP Architecture. In *Proceedings of the International Conference on Parallel Computing (ParCo2011)*, Ghent, Belgium, August/September 2011.

[2] E. Cecchet. Memory Mapped Networks: A New Deal for Distributed Shared Memories? The SciFS Experience. In *Fourth IEEE International Conference on Cluster Computing (CLUSTER'02)*, 2002.

[3] M. Chapman and G. Heiser. vNUMA: A Virtual Shared-Memory Multiprocessor. In *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 349–362, San Diego, CA, USA, Jun 2009.

[4] 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.

[5] 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.

[6] G. Heiser. Many-Core Chips — A Case for Virtual Shared Memory. In *2nd Workshop on Managed Many-Core Systems (MMCS)*, page 4 pages, Washington, DC, USA, March 2009.

[7] IEEE, editor. *Standard for Scalable Coherent Interface (SCI)*. Number 1596 in IEEE Standards. The Institute of Electrical and Electronics Engineers, Inc., 1992.

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

[9] Intel Corporation. *The SCC Programmer's Guide*, July 2010. Revision 0.63.

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

[11] 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*, pages 13–21, 1992.

[12] P. Keleherand, 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*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[13] 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*, pages 576–585, Wroclaw, Poland, 2010.

[14] S. Lankes, P. Reble, C. Clauss, and O. Sinnen. The Path to MetalSVM: Shared Virtual Memory for the SCC. In *Proceedings of the 4th MARC Symposium*, Potsdam, Germany, December 2011.

[15] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions Computer Systems*, 7(4):321–359, 1989.

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

[17] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: The Programmer's View. In *Proceedings of the 2010 ACM/IEEE Conference on Supercomputing (SC10)*, New Orleans, LA, USA, November 2010.

[18] D. Mentre and T. Priol. NOA - A Shared Virtual Memory over a SCI Cluster. In *Proceedings of SCI Europe '98*, pages 43–50, September 1998.

[19] 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.

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

[21] 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.

[22] D. Schmidl, C. Terboven, A. Wolf, D. an Mey, and C. Bischof. How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture. In *Proceedings of 2010 IEEE International Conference on Cluster Computing*, pages 29 –37, September 2010.

[23] 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.

[24] M. Schulz. SISCI-Pthreads SMP-like programming on an SCI-cluster. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, (HPCN EUROPE 1998)*, Amsterdam, Netherlands, April 1998.