Evaluation of Hardware Synchronization Support of the SCC Many-Core Processor

Pablo Reble, Stefan Lankes, Florian Zeitz and Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University, Germany
{reble,lankes,zeitz,bemmerl}@lfbs.rwth-aachen.de

Abstract

The integration of many cores per chip will lead to inefficiency of traditional multi-processor techniques. In particular, a hardware cache coherency protocol includes performance and hardware overhead, so that for a growing number of cores the coherence wall problem will become more serious.

The Single-chip Cloud Computer (SCC) is a recent research processor of a Cluster-on-Chip architecture, that waives a hardware-based coherency and possesses a network on chip technology. An attractive alternative to enable shared memory programming models on future many-core systems is the introduction of a software-oriented coherency.

Any software based approach, such as shared virtual memory (SVM), will need fast synchronization methods. The assumption is that hardware support is essential to achieve this performance. In this paper we will study and evaluate this hypothesis.

1 Introduction

The Single-chip Cloud Computer (SCC) experimental processor is a *concept vehicle* created by Intel Labs as a platform for many-core software research. The processor consists of 48 P54C cores arranged in a 6×4 on-die mesh of tiles containing two cores each and resembles a *Cluster-on-Chip* architecture by providing distributed, but shared memory.

A major goal of the research many-core architecture is to find an answer to the coherence wall problem. The SCC platform waives hardware cache coherency and provides a low-latency infrastructure called *Message Passing Buffer* (MPB). Therefore, each tile has a special shared on-die memory of 16 kByte[1]. These memory regions are accessible by all cores without any coherency and mainly used for message-passing between the SCC cores in an explicit way.

The lack of a hardware cache coherency arises problems for shared memory synchronization constructs. Because the use of atomic operations, to update a shared memory location, is not possible on such an architecture. Nevertheless, busy wait implementations, which are commonly used for low latency synchronization constructs can be used with limitations on the SCC. By switching off or explicitly flush the local core cache, the on-die MPB memory as well as the off-die shared memory can be used for flag based synchronization. Additionally, to provide atomic operations, the SCC possesses a small set of special memory mapped hardware registers, namely Test and Set Register (TSR) and Atomic Increment Register (AIR).

This paper is structured as follows. In Section 2 we motivate the evaluation of this hardware synchronization support for shared memory applications. Section 3 presents an overview of the MetalSVM project a well as classic work to increase scalability of busy-wait synchronization methods. In Section 4 we describe the SCC hardware with a focus on synchronization support. Next, we analyze in Section 5 the fairness and scalability of busy-wait synchronization methods on the SCC platform and present alternatives. In Section 6 we present optimized barrier implementations based on these results.

2 Motivation

The default configuration of the SCC platform suggests the use of the message passing programming model. The global shared memory is partly partitioned among the cores to run a separate Linux instance on each core. With RCCE a light weight message passing oriented programming library exists to facilitate the use of the message passing programming model [10].

However, many applications benefit strongly from using a shared memory programming model. Established

[1] default distribution 8 kByte per core

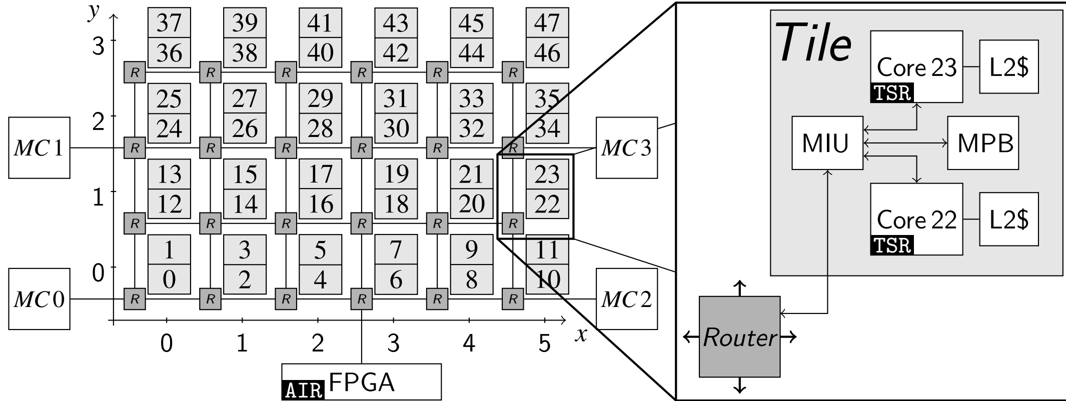


Figure 1: SCC layout with a focus on tile architecture (based on [11])

shared memory programming interfaces like OpenMP focus on thread and loop parallelism. Explicit synchronization primitives such as locks and critical sections as well as implicit synchronization means in the form of reduction clauses are defined.

The MetalSVM project has been started to enable thread based shared memory programming on many-core architectures. The concept is the integration of a shared virtual memory (SVM) management system into a bare-metal hypervisor. As a result, a shared memory application is executed on a standard operating system, which runs as a paravirtualized guest on top of MetalSVM. Therefore, *lguest* is integrated to a minimalist monolithic kernel, self-developed by the authors. Compared to the Single System Image (SSI) approach, an SVM system only virtualizes the memory, which reduces the complexity of the virtualization environment.

Our project approach shares similarities to the vSMP architecture developed by ScaleMP. Evaluation of this architecture has shown that expensive synchronization is a big drawback [14]. Here, we see an advantage of the design of our architecture. The access to special synchronization resources can be easily managed by our hypervisor and transparently provided to selected user applications.

3 Related Work

In this paper, we benchmark common shared-memory synchronization algorithm implemented by the subsequently added synchronization support of the SCC platform. This work is based on the presentation of an inter-kernel communication and synchronization layer for MetalSVM in [13], at the 3rd MARC [1] symposium. This already includes an analysis of the effect of a mesh interconnect to the latency of on-die synchronization support. Furthermore, we introduce the use of iRCCE [4] for a non-blocking and low latency inter-kernel communication.

A first prototype of our SVM system has been presented at the 4th MARC Symposium [7]. Further Optimizations of our prototype and first experiments with relaxed consistency models are presented in [8].

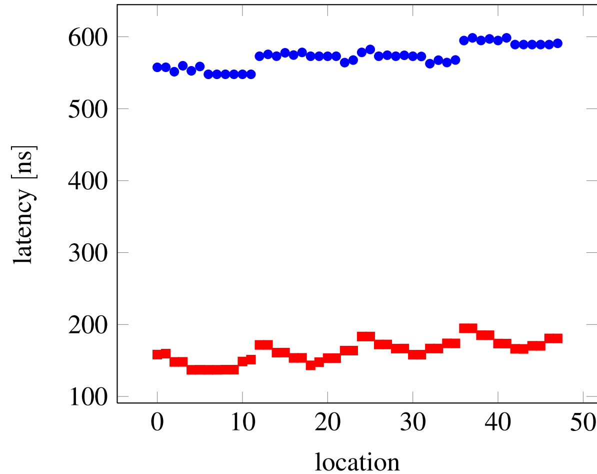
Anderson has motivated years ago in his article [2] the need of hardware support of busy-wait mutual exclusion on shared memory multiprocessors. Consequently he presented spin-lock alternatives to increase scalability, for instance by the use of a back-off policy. Previously, Lamport discovered that pure software mutual exclusion is quite expensive [6].

Graunke and Thakkar [5] proposed queuing based locking algorithms for cache coherent systems instead of simple test and set locks. Their experiments have shown that on the strategy of spinning on a different cache-line outperforms a centralized approach.

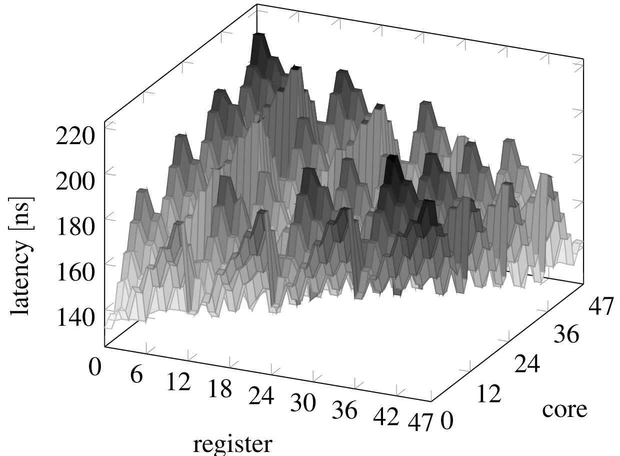
4 SCC Hardware

The SCC platform supports basic synchronization primitives implemented in hardware. These are *Test and Set Registers* (TSR) and *Atomic Increment Registers* (AIR).

As the SCC cores are based on the 32 bit pentium architecture, an additional mechanism is needed to cover the entire system address space of 46 bit. Each core holds a separate lookup table (LUT) with 256 entries to translate core addresses to system addresses. A system address consists of the encoded mesh coordinates, the router port, and the address of a memory segment including offset. LUT entries are configurable and point to specific types of memory regions (off-chip memory, on-chip memory, configuration and synchronization register) with a maximum size of 16 MB. Therefore, read and write requests of a core are performed transparent by the mesh interface unit (MIU cf. scale-up from Figure 1) across its FSB.



(a) Atomic Increment Register • and Test and Set Register ▪



(b) Access latency to Test and Set Register from Core

Figure 2: Read access latencies of hardware synchronization registers

The synchronization registers, which are described in the following, are memory mapped accessible the described way. A pair of *Test and Set Registers* is physically located on-die at each tile ($2 \times 24 = 48$) to realize an atomic test and set instruction on a binary value. Consequently, a TSR can have two possible states, '0' or '1'. The initial state, which can be restored by a writing access, is '0'. A read access can atomically change the state from '0' to '1', where the read data holds the previous state. For an extension of functionality, the Rocky Lake system FPGA, which is directly connected to the on-die mesh interconnect, provides a set of off-die *Atomic Increment Counter* ($2 \times 48 = 96$ AIC). Here, an AIC is structured as a pair of Atomic Increment Register (AIR), namely *initialization* and *increment* register. A write access to the *initialization* register loads a 32 bit value to the AIC. Whereas, a read access simply returns the current value of the AIC. A read access to the *increment* register triggers an atomic post-increment operation of the AIC value, related to the read data, whereas a write access just decrements the current value atomically.

In addition to the synchronization registers the SCC provides a broad set of configuration registers. For instance, to change frequencies and voltages of mesh, cores and off-chip memory. Core frequencies have a range from 100 MHz to 1 GHz. These are fully configurable only depended on the selected voltage domain, which spans across four cores. For all benchmark results presented in this paper the SCC platform has been statically configured with a frequency of 533 MHz for the cores, and a frequency of 800 MHz for the mesh and the memory.

Figure 2a shows average access latencies, over a million accesses, to a fixed TSR and AIR from all cores. The marks ▪ represent the average access latency to the TSR of core 6, located at x-y mesh coordinate (3,0) (cf. Figure 1). To ensure comparability, we selected the TSR closest to the AIRs for this measurement. Latencies of TSR accesses are lower by a factor of 3 compared to AIR latencies due to the on-chip location. The marks • represent the access latencies to the atomic increment registers, which are located off-chip at the system FPGA.

Figure 2b gives a full overview of read access latencies dependent on location of TSR and location of core. This implies a variation of target synchronization register, as well as, the core that triggered the register access. The result is a deterministic average latency in relation to the mesh distance.

5 Spin-Lock

In this section, we present the effect of a fast on-die mesh interconnect on common busy-wait synchronization techniques and hardware synchronization registers. As these resources are scarce, one has to careful choose their application. The simplest implementation of a spin-lockis a loop, which requests a TSR while the read data is '1'. Currently, acquire and release lock are the only two functions of the RCCE gory API, that exist to explicitly access a TSR.

Figure 2b shows a huge impact of physical mesh distance to access latency of a synchronization register. Thus, we analyze fairness and scalability exemplarily for different spin lock implementations, which use hardware synchronization support. Results of the following experiments can be applied to general busy wait synchronization constructs. For instance, on a barrier construct in Section 5.

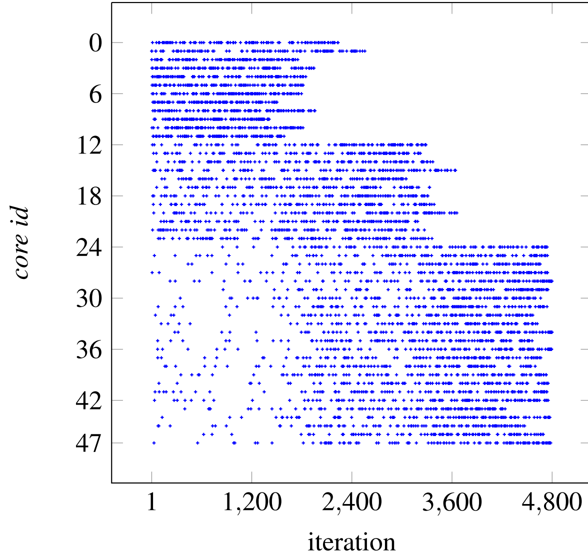


Figure 3: Iteration scattering of simple spin-lock on TSR (location 0) – per core 100 locks

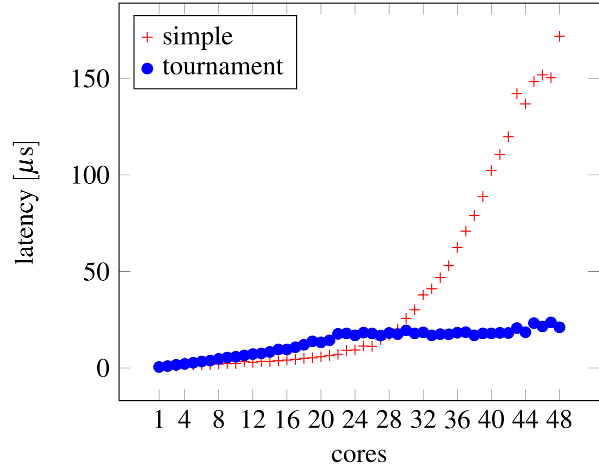


Figure 4: Performance of spin-lock implementations

5.1 Fairness

For the analysis of fairness of a simple busy-wait algorithm, we generate a high contention on a hardware synchronization register. Therefore, all available 48 cores increment a shared counter one hundred times. The increment operation is protected by a spin-lock, which uses the RCCE implementation.

Target hardware synchronization register is physically located at the lower left corner of the mesh (*core id:* 0, cf. Figure 1). Since each core records the obtained counter values, the fairness of a busy-wait synchronization method with a single target can be classified. Therefore, the scattered plot from Figure 3 visualizes the chronological order.

Analysis of the measurement shows the expected result of this benchmark. Cores, that are located at a tile close to the synchronization register, in this scenario id 0 to 11, nearly pass around the lock during the first iterations. Not before this first group of cores finished contending for the lock, all other cores can frequently acquire the lock.

An interesting effect is, that this access behavior mainly depends on their y-coordinate. The explanation for this effect is the x-y routing of the on-die network. A read request to a register generates a network packet, which is first routed to target x and second to target y coordinate. The busy-wait synchronization scheme intentionally generates a high load for the interconnect. Here, the path in y-direction appears to be a bottleneck. This fact can lead to starvation under high contention of cores with a large distance to target synchronization register.

5.2 Scalability

In the next experiment, we compare the simple spin-lock implementation to a tournament lock [5] implementation on the SCC platform. Here, multiple synchronization register are used to build an n-ary locking tree. The diagram in Figure 4 shows measurement results for a tournament lock based on a binary tree with a depth of one (●), compared to the simple spin-lock (+).

Exemplarily three registers are used to build a locking tree and reduce the maximum contention by a factor of 2. Following from the results of the previous benchmark, cores are preferably assigned to locks sharing their y-coordinate. In the presented example, as a first step, each core contends for the assigned group lock and, as a second step, for the global lock to acquire the tournament lock. Accordingly, the locks are released in reverse order to release the tournament lock.

For a comparison, both spin-lock implementations are used to protect a critical section for an increasing number of cores to generate a high contention . In contrast to the evaluation of Section 5.1, here the critical section is empty and a high contention continues over all iterations. Thus, a core which has performed one thousand timed iterations, requests the lock repeatedly to generate noise for the remaining cores. The plotted values from Figure 4 are maximum values of all participants of the average spin-lock latencies to acquire and release a lock.

The results of the presented experiment are a linear increasing spin-lock latencies for both implementations up to a core count of 24. This is an expected behavior, because of the linearly raising core count. A further increase of the core count leads to a constant latency of 20 μs for the tournament lock and the latencies for the simple spin-lock increase exponentially from a core count of 24.

The locking path for the simple spin-lock is minimal and consists of two synchronization register accesses. Consequently, the locking path for the tournament lock, with a depth of two, consists of four synchronization register accesses, which results in a higher latency up to a core count of 28. Average register latencies contention of more than 42 cores must be considered with caution. Due to the fact that cores can starve under high contention, exact values are of little importance.

However, this experiment proves the assumption that a simple spin-lock implementation has a bad scalability under high contention on the SCC platform. In [13] we introduced the use of an exponential back-off to relax the contention on a synchronization register. Additionally, we presented in this section results of the tournament lock, which are promising to relax the contention. In the next section, we apply further optimizations to the implementations of common barrier synchronization algorithms, based on these results.

6 Barrier

A shared virtual memory system can combine shared memory programming and a many-core architecture, e. g. by approaching the memory consistency problem on a page granularity in software. The programming model OpenMP introduces constructs and clauses to enable the creation of parallel shared-memory programs. Many of these constructs imply a barrier [3].

We present in this section, two barrier implementations, which use the hardware synchronization support of the SCC to atomically increment a 32 bit counter. For a performance analysis, we compare the latency of both barrier implementations to a TSR based barrier from [13] and a MPB based implementation from the RCCE library [10]. Therefore, the different barrier implementations are repeatedly called one million times. Figure 5 depicts the maximum average latency for all threads, whereas each thread runs on a dedicated SCC core.

The RCCE library contains a simple barrier implementation, which is based on a master follower approach. A master thread is responsible to count incoming threads as well as release waiting threads, by using flags, which are located in the MPBs. RCCE follows a *local-put, remote-get* approach for message passing. This means, that a flag based synchronization only touches the MPB at that core, which has initiated an update. As a result of this approach, the linear release cycle requires remote polling of the master core, repeatedly for all follower cores. At least, this approach avoids a centralized structure.

In [13] we presented another flag based barrier approach, which uses a set of atomic test and set registers to indicate and release incoming threads. For the initialization of this barrier, a number of initially unlocked TSR equals the number of threads is allocated. Each thread, which enters the barrier, performs a linear search for an unlocked TSR. Thereby, target TSR becomes locked and this thread, except the last one, enters the release cycle, by polling on the specific TSR.

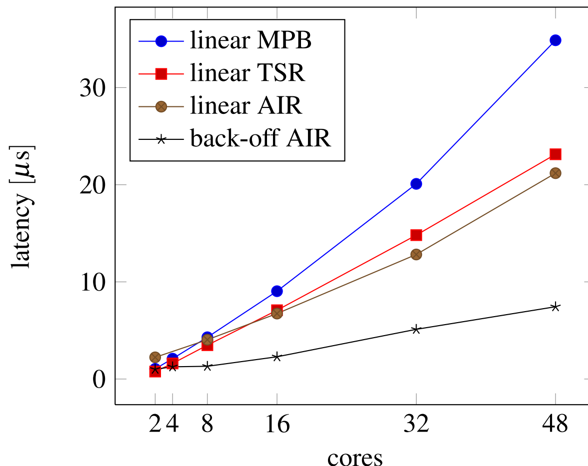


Figure 5: Performance of Barriers

Curve —■— from Figure 5 shows that this implementation of a linear barrier has a lower latency compared to the reference implementation of Curve —●—. However, the allocation of a TSR for each participating thread is expensive, as resources are scarce.

Consequently, we implemented a classic centralized barrier algorithm according to Lubachevsky [9] on the SCC Platform. This simple barrier algorithm is based on counters, which are located in shared memory and can be atomically incremented. Each thread increments a counter to indicate its arrival and polls that counter, to determine when all of the threads have arrived. Therefore, the last thread of a group resets the counter and exchanges the counter reference, to avoid that one thread mistakenly passes a barrier. As a result, of the original Lubachevsky barrier, two shared memory counters have to be allocated, since they are used for the indication as well as the release of incoming threads.

For the straight forward implementation of a Lubachevsky barrier for the SCC, two Atomic Increment Registers (AIR) have to be allocated. Because of its centralized structure, this implementation is not working without further optimization. Our experiments from Section 5 show, that a high contention is problematic on the SCC platform starting from a certain core count (cf. Figure 4). If more than 30 cores are polling on an off-die Atomic Increment Counter (AIR), a starvation of cores with a large distance to the physical location of the AIRs can be detected (cf. Figure 1).

Common techniques to relax this contention problem can be applied to a synchronization register and increase the scalability of a busy wait centralized barrier algorithm [5, 12].

Our first barrier implementation introduces an exponential backoff by polling the AIR during release cycle for the previously described straight-forward Lubachevsky barrier implementation. This method significantly reduces the contention and leads to promising results. Curve —⋆— from Figure 5 shows the latencies parametrized with an optimal minimum and an optimal maximum backoff for each group size of threads.

Our second AIR barrier implementation, only uses a single AIR to indicate incoming threads and MPB located flags to release waiting threads. Here, the linear method of the reference barrier implementation is used for the release cycle to avoid a high contention on the AIR. A better performance for a group size of more than 8 threads can be achieved, compared to the TSR barrier implementation. A further advantage is that only one hardware synchronization register is allocated for this first AIR barrier implementation. Curve —●— from Figure 5 depicts the resulting latencies for this implementation.

With our barrier implementation according to Lubachevsky for the SCC platform, a significantly speedup can be achieved compared to the linear reference implementation. If two atomic increment registers are reserved, a speedup of up to 6 can be achieved for this synchronization construct. If a single register is only available, the speedup decreases to 1.6, due to the linear release cycle.

7 Conclusion

We present an evaluation of the hardware synchronization support of a recent cluster-on-chip architecture in this paper. This evaluation verifies issues of common busy-wait synchronization techniques on future many-core architectures regarding fairness and scalability.

Our experiments show massive improvements by common optimizations such as an exponential back-off or the use of multiple hardware synchronization primitives. Scalability of a single synchronization point can be increased by structuring the physical resources in a hierarchical way. Obviously, this is a trade-off between an optimal scalability and allocation of synchronization registers as resources are scarce. If the implementation of mechanism is carefully chosen, the use of hardware support for synchronization constructs, such as a barrier, leads to promising results.

8 Acknowledgments

The authors would like to thank Intel Labs Braunschweig for the research grant and in particular Ulrich Hoffmann, Michael Konow and Michael Riepen for their help and guidance.

References

[1] Intel Many-core Applications Research Community. `http://communities.intel.com/community/marc`.

[2] ANDERSON, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (January 1990), 6–16.

[3] CHAPMAN, B., JOST, G., AND VAN DER PAS, R. *Using OpenMP: Portable Shared Memory Parallel Programming*, vol. 10. The MIT Press, 2007.

[4] CLAUSS, C., LANKES, S., GALOWICZ, J., AND BEMMERL, T. *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, December 2010. Users' Guide and API Manual.

[5] GRAUNKE, G., AND THAKKAR, S. Synchronization algorithms for shared-memory multiprocessors. *Computer 23*, 6 (June 1990), 60 – 69.

[6] LAMPORT, L. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS) 5* (January 1987), 1–11.

[7] LANKES, S., REBLE, P., CLAUSS, C., AND SINNEN, O. The Path to MetalSVM: Shared Virtual Memory for the SCC. In *Proceedings of the 4th MARC Symposium* (Potsdam, Germany, December 2011).

[8] LANKES, S., REBLE, P., CLAUSS, C., AND SINNEN, O. Revisiting Shared Virtual Memory Systems for Non-Coherent Memory-Coupled Cores (to appear). In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2012)* (New Orleans, LA, USA, February 2012).

[9] LUBACHEVSKY, B. Synchronization barrier and related tools for shared memory parallel programming. *International journal of parallel programming 19*, 3 (1990), 225–250.

[10] MATTSON, T., AND VAN DER WIJNGAART, R. *RCCE: a Small Library for Many-Core Communication*. Intel Corporation, May 2010. Software 1.0-release.

[11] MATTSON, T., VAN DER WIJNGAART, R., RIEPEN, M., LEHNIG, T., BRETT, P., HAAS, W., KENNEDY, P., HOWARD, J., VANGAL, S., BORKAR, N., RUHL, G., AND DIGHE, S. 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).

[12] MELLOR-CRUMMEY, J., AND SCOTT, M. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS) 9*, 1 (1991), 21–65.

[13] REBLE, P., LANKES, S., CLAUSS, C., AND BEMMERL, T. A Fast Inter-Kernel Communication and Synchronization Layer for MetalSVM. In *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing* (Ettlingen, Germany, July 2011).

[14] SCHMIDL, D., TERBOVEN, C., WOLF, A., AN MEY, D., AND BISCHOF, C. How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture. In *Proceedings of 2010 IEEE International Conference on Cluster Computing* (September 2010), pp. 29 –37.