In-Memory Communication Mechanisms for Many-Cores – Experiences with the Intel SCC

Randolf Rotta, Thomas Prescher, Jana Traue, Jörg Nolte
{rrotta,tpresche,jtraue,jon}@informatik.tu-cottbus.de

***Abstract*—Many-core processors combine fast on-chip communication with access to large amounts of shared memory. This makes it possible to exploit the benefits of distributed as well as shared memory programming models within single parallel algorithms. While large amounts of data can be shared in the memory and caches, coordinating the activities of hundreds of cores relies on cross core communication mechanisms with ultra low latency for very small messages. In this paper we discuss two communication protocols for the Intel SCC and compare them to the MPI implementation of the SCC. Our micro-benchmark results underline that special purpose protocols for small messages make much finer levels of parallelism possible than general purpose message passing systems.**

***Index Terms*—many-core, message passing, shared memory**

I. Introduction

The number of cores in current many-core architectures is increasing while the performance of most cores decreases in favour to smaller cores [1]. Processors with more than 1000 cores are expected by some researchers [2], but the actual number depends on the efficiency of real applications. Such many-core systems are actually hybrids that combine aspects of distributed and shared memory systems [3]. An on-chip network interconnects the cores and can be used to pass messages between them. On cache coherent systems (e.g. processors based on Intel's MIC [4], and Tilera processors [5]), these messages are transmitted implicitly from the sender's to the receiver's cache by the coherence protocol. Other architectures provide fast on-chip memory for message passing (e.g. Intel SCC [6] and IBM Cyclops64 [7]) or direct communication channels between cores (e.g. Tilera's UDN [5] and Adapteva [8]). However, passing large messages tends to be inefficient on such architectures, because composing and receiving these messages evicts large portions of the caches. In the worst case, the message data is read from main memory, passed through the caches of the communicating cores, and is finally written back to the very same memory module. Moreover, the network connects all cores to the main memory, thus providing a shared memory that allows to share data without the need of copying it between cores.

Consequently, parallel programming environments for many-cores try to avoid large unnecessary data copies. For example, in the zero-copy multicast protocol ZIMP [9] all receivers of the same message read it directly from shared memory. More radical approaches avoid data transfers by moving the work to the data instead. Such *function shipping* (remote execution of function calls) can also improve the cache utilization for shared objects [10]. The authors of [11] use function shipping to offload critical sections and synchronisation mechanisms to dedicated cores.

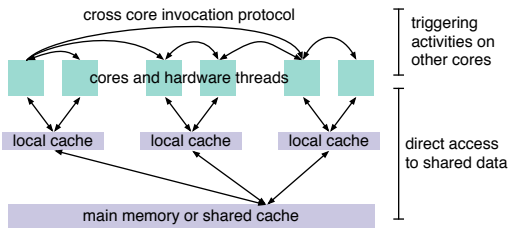

Fig. 1. Combining CCIs and data sharing on many-core processors.

Suppose that each core runs a fixed application process or thread which uses message passing for coordination and shared memory for direct access to application data as depicted in Figure 1. Then, function shipping mechanisms actually become *cross core invocations* (CCIs) that trigger the execution of specific tasks on other cores. Such CCI messages are usually extremely small [12]; thus, special purpose communication protocols can be optimized for small message sizes.

The experimental Intel SCC processor is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. We developed several CCI communication protocols that use SCC's hardware support for message passing. The next section introduces the message passing features of the SCC and presents two of the communication protocols. Their impact on the performance of CCIs was evaluated by porting the TACO framework [13] to the SCC. Section III provides a short overview of the framework and presents benchmark results in comparison to TACO on top of the general purpose message passing provided by MPI. Finally, we discuss related work and provide concluding remarks.

II. Communication Protocols for CCIs

The Intel Single-chip Cloud Computer (SCC) [6] combines 48 P54C Pentium processor cores on a single chip. Details about programming on the SCC can be found in [14]. All communication is performed by memory read/write operations that are carried out over a packet-switched 2D mesh network. Each router is connected to two cores and a 16kB on-chip SRAM device called *Message Passing Buffer* (MPB). Some routers are connected to external devices like main memory and the system interface (SIF), which is a FPGA chip that provides, among others, 2×48 atomic increment counters.

The processor has no implicit cache coherence mechanism. Instead, two memory access modes that allow manual control

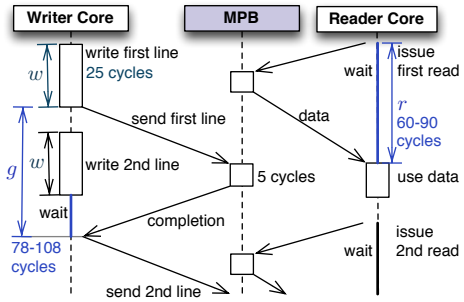

Fig. 2. MPBT writes and reads to the on-chip Message Passing Buffers.

TABLE I
READ AND WRITE ACCESS TIMES TO THE MPBS (IN CYCLES).

	UC	bypass	MPBT	bypass
w	5	5	25	25
g	51–84	17	78–108	39
r	54–87	19	60–90	21

over the caches can be configured through flags in the page table. Read operations with the *Message Passing Buffer Type* (MPBT) are cached only in the L1 cache and a new instruction called `CL1INVMB` invalidates all MPBT lines. Write operations are collected in a write combine buffer that tries to fill up an entire line before sending it in a single message. Read and write operations with the *uncached* (UC) mode go directly to the network, except when there is a L1 cache hit. It is possible to mix MPBT and UC access to the same memory by using `CL1INVMB` before switching from MPBT to UC access.

The next subsection summarizes our experience with the SCC in a cost model for basic memory operations to support reasoning about design decisions. Then, two communication protocols, NVP and SRBP, are presented. Both place all of the messages and coordination data into the MPBs.

A. Cost Model for Memory Access on the SCC

The P54C cores are strictly in-order and can handle a single outstanding memory request only—a core simply stalls until a previous access is acknowledged. This ensures write ordering when accessing different destinations. The behaviour can be summarized by a model with three parameters: the write overhead w, write gap g and read overhead r (c.f. Fig. 2). MPBT writes are collected in a write combine buffer. The data is sent to the memory when either all 32 bytes of a line are written, or when the next MPBT write goes to another line. The *write overhead* w is the time necessary to fill the buffer and trigger the transfer. Because it is a local operation, the write overhead is independent from the distance to the destination. When the data is issued to the network, the write combine buffer can be filled again. However, the next line can only be issued to the network after receiving the completion message of the previous write. The *write gap* g describes this time. Read operations without a L1 cache hit send a request to the memory and the core stalls until the data arrives. The *read overhead* r denotes this waiting time. The UC mode follows the same model, but without using the write combine buffer.

We did micro-benchmarks for read and write operations to MPBs and Table I reports the results for the system

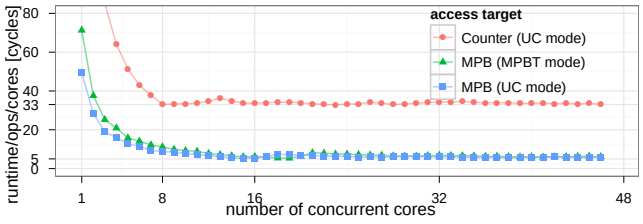

Fig. 3. Saturation of the on-chip MPB and off-chip atomic counters.

configuration with 800 MHz core frequency and 1600 MHz mesh frequency. The value ranges represent the smallest and largest network distance. Intel's manual [15] states 45 core cycles plus $8n$ mesh cycles for a MPB access. Since the mesh is twice as fast as the cores, this yields exactly 4 cycles per router and these numbers fit to our measurement for the UC write gap. Reading a line with MPBT mode is as fast as reading a single word in UC mode. Thus, MPBT reading should be preferred whenever there is a chance of cache-reuse. The costs of local writing plus remote reading are equal to the costs of remote writing plus local reading. Hence, it does not matter whether messages are placed in the sender's or receiver's MPB. The send gap and read overhead increase just slightly with the distance. Thus, most of the time is spent inside the cores and at the mesh interface. This might be better on more modern cores and the *bypass* columns give an impression of what is possible. The experimental *bypass* accesses the local MPB directly without going through the mesh interface. This saves around 35 cycles in UC mode and 39 cycles in MPBT mode. Unfortunately, the bypass does not work reliably due to a conflict when both cores access their shared local MPB concurrently.

The processing overhead of a request determines how many cores can communicate with the same MPB in parallel because the intermediate network interleaves the requests from many cores. Figure 3 shows the results of a saturation experiment in which an increasing number of cores read concurrently from the MPB at core 0. The MPBs are fully saturated with around 15 cores and need 5–6 cycles to process each request.

The system interface hosts a set of atomic increment counters. A counter is incremented by reading from a configuration-specific memory address. In our experiments, the atomic counters were relatively slow (240–270 cycles per increment) and easy to saturate with 8 cores, because each increment needs around 33 cycles to be processed (Fig. 3). Thus, while concurrent access to the MPBs incurs only small additional costs, the atomic counters quickly become a performance bottleneck. However, in a non-experimental processor, such counters can be implemented directly on the chip alongside the MPB. Then, the access costs and internal processing overhead would be similar to reading from the MPBs.

B. Notification Vector Protocol (NVP)

The NVP protocol uses the UC and MPBT modes but not the atomic counters. Because concurrent senders cannot synchronize, they have to write to separate memory locations.

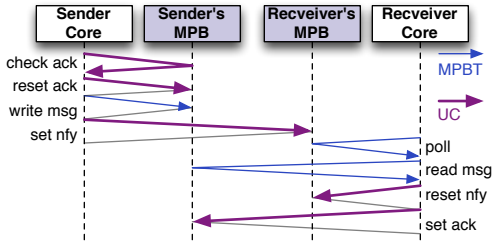

Fig. 4. One-way message transfer in NVP.

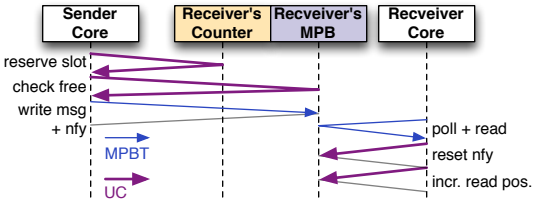

Fig. 5. One-way message transfer in SRBP.

A fixed message slot is used for each pair of cores and each direction, but advanced implementations could manage them more dynamically. The message size is limited to at most 96 bytes (3 cache lines), because this size was sufficient in our applications. Figure 4 summarizes a message transfer.

The receiver has to check its 48 inbound slots, which is accelerated with a separate vector of 48 notification flags that is stored at each receiver's MPB. Because a byte is the smallest unit that can be written atomically (using UC mode or a partially filled write combine buffer), the vector is 48 bytes long which means that the flags can be fetched with two MPBT reads. The search for new messages is further accelerated by checking double words (4 bytes) first, which reduces the number of conditional branches from 48 to 12 when no message arrived. For simplicity, always the complete vector is scanned including the own notification flag and all of the found notifications are processed immediately. The receiver reads and processes the message (MPBT read) and resets the notification flag (UC write). Because a sender should never overwrite its own previous notification before it was processed, he has to wait for the receiver. This is solved by a second vector of 48 acknowledgement flags that is stored on the sender's side. The sender writes the message data into the receiver's slot (MPBT write) and, then, sets its flag in the receiver's notification vector (UC write). Unfortunately, an additional local MPBT write is necessary after writing the data because an interrupt could disturb the MPBT write, resulting in incompletely sent data and a partially filled write combine buffer. The dummy write forces the write combine buffer to send the remaining data.

Some modifications of the protocol would be possible. In principle, the sender could check the notification flag directly instead of using an acknowledgement vector. In this case, waiting for the receiver results in repeated remote instead of local reads which increases the load on the network. In experiments, simple message roundtrips became around 100 cycles faster, but complex collective operations were around 1000 cycles slower. The memory consumption of the protocol could be reduced by using a dynamic assignment of slots instead of the fixed slots per pair of cores. This makes the number of slots independent from the core count, but slot IDs have to be transferred in the notificiation flags.

C. Synchronized Ringbuffer Protocol (SRBP)

In this protocol, each core has a ring buffer of message slots and a publicly readable *read position* that is stored at the core's MPB. The number of slots is independent of the number of cores and is limited just by the size of the MPBs. With 8kB per core and 96B per slot, 85 slots per core are possible. For each message, the sender increments the *write position* of the receiver's ring buffer. The write positions are implemented by SCC's atomic counters in order to dispatch concurrent senders to unique message slots. Before writing the message to this slot (MPBT write) the sender has to wait for the slot to become free. This is detected by checking the logical distance between the acquired write position and the receiver's current read position (UC read). The notification is implicitly contained in the message data by writing the non-zero message header last, that is writing the data backwards. Thus, the receiver just has to check for the non-zero header at its current read position using a MPBT read and then already has the first line of the message. Dummy writes to protect against a partially filled write combine buffer are not necessary. After processing the received message, the receiver resets the embedded notification and increments its read position (two UC writes).

III. Performance Impact on the SCC

We ported TACO [13] to the SCC by replacing the MPI-based communication protocol. TACO-based programs run as a set of processes where each process has its own local address space and a unique process identifier. Global object pointers combine the processes into a partitioned global address space (PGAS) that consists of all combined local address spaces. The pointers consist of the object's home identifier and the memory address within that process. On top of these pointers, TACO provides synchronous and asynchronous *remote method invocation* (RMI) mechanisms. Collective operations extend the point-to-point RMIs to method calls on distributed groups of objects. On many-core systems, the TACO processes are assigned to cores or hardware threads. Thus, object pointers address specific cores and RMIs provide high-level mechanisms for cross core invocations.

The next subsections compare TACO on top of NVP, SRBP, and the general purpose RCKMPI [16] with respect to the CCI roundtrip times, the completion time of collective operations, and the scalability in relation to the system size. All measurements were performed with the 800x1600MHz clock configuration. The last subsection discusses preliminary results on the energy consumption.

A. Cross Core Invocations

Each TACO-process can create and access instances of arbitrary C++ classes on other cores. Such objects are referenced by *global object pointers* and these provide methods

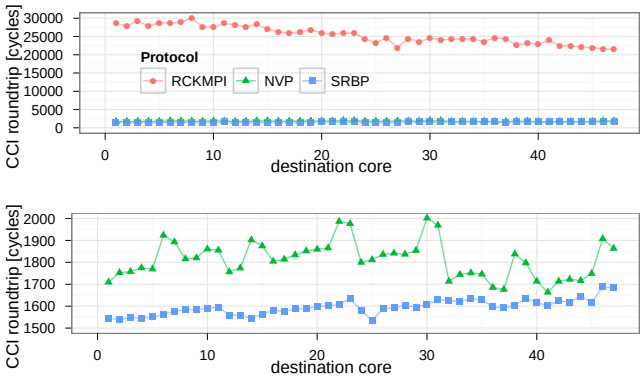

Fig. 6. Synchronous CCI roundtrip times from core 0 to other cores.

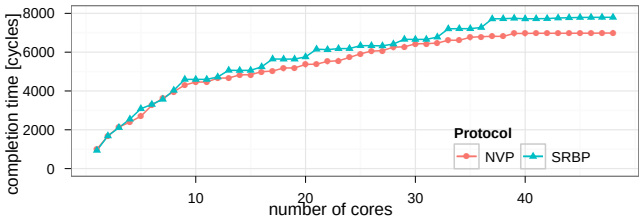

Fig. 7. Step completion time for different group sizes.

to send method calls to the destination object. Synchronous and non-blocking calls are implemented with the help of future variables [17]. These wait implicitly for the arrival of the result when their value is read. The method calls are wrapped automatically in function objects (so-called *functors*) that contain a pointer to the destination method as well as the call arguments and can be applied on any object of matching type. TACO's communication backend sends these as active messages to the destination core. The active messages are very small (24 bytes without any arguments), because they consist only of a pointer to the active message's handler function, a pointer to the target object, the functor, and an optional global pointer to the future variable for the result. TACO's idle loop checks for new messages and these are executed directly from the MPB. The standard active message handlers copy the message to an internal receive queue and idle helper threads pull the functors from there to execute them. Thus, threads that wait, for example, in a future variable do not block the entire process. The helper threads are implemented as cooperative user-level threads. This approach has a much lower overhead than standard POSIX threads because it avoids (preemptive) context switching and system calls.

The roundtrip benchmark measured the average completion time of synchronous cross core invocations from core 0 to another core. The results are shown in Figure 6. A roundtrip takes 1600–2000 cycles with NVP (2–2.5μs) and 1500–1700 cycles with SRBP (1.9–2.1μs). The distance on the underlying network contributes just 10% to the communication costs. Thus, job placement strategies do not need to worry about distances and message latencies.

The SRBP protocol is just slightly faster than NVP although it is much simpler. The overhead of checking for new messages is lower (NVP: 188 cycles, SRBP: 130 cycles), but the overhead of sending a message is twice as large (NVP: 264 cycles, SRBP: 472 cycles). The latter is caused by the relatively slow off-chip atomic counters. However, moving the counters to the MPBs and fixing the bypass mode will not change the situation, because then both protocols will be around 200 cycles faster (based on Tab. I). Hence, more elaborate hardware-based notification mechanisms will be necessary in order to further decrease the communication overheads.

A roundtrip with TACO on top of RCKMPI takes around 20 000–30 000 cycles (25–37μs) which is up to 15 times longer than with NVP and SRBP. The roundtrip experiment is similar to MPI send/recv roundtrips and our measurements results match the times reported in [16]. One reason is the polling overhead of the SCCMPB protocol (see also Sec. III-C). However, this is not easy to circumvent in protocols that have to be able to transfer arbitrary large messages.

CCIs can be used to parallelize a computation by offloading tasks to other cores. This is efficient only when the workload of such tasks is at least as large as the send and receive overhead. With the presented protocols, this overhead is about as large as the roundtrip time. In conclusion, special purpose protocols like NVP and SRBP on the SCC make it possible to efficiently offload tasks with up to 15 times smaller workloads compared to general purpose message passing.

B. Collective Operations

Often, related tasks have to be distributed over a group of several cores. TACO simplifies this by providing *collective operations* on distributed groups of objects. The construction of object groups is automated by predefined topology classes that create a multicast tree and *group pointers* reference the root of such trees. The group pointers provide methods to initiate collective method calls and the call functors are propagated through the tree by using point-to-point CCIs. Return values or completion notifications are sent back through the tree and are merged at the intermediate nodes.

The collective *step* operation applies the method call to all group members in parallel and a future variable is used to wait for the completion of all calls. The following benchmark measured the average completion time of such steps. A simple balanced 4-ary multicast tree with the root on core 0 was used and the group size ranged from 1 to 48 cores. The results with TACO on NVP and SRBP are shown in Figure 7. As can be expected from a multicast tree, the completion time grows logarithmically with the size. Moreover, the overhead per core is constant by design: The cores can sleep or work while the operation is propagated. A step operation on the complete 48-member group takes around 6800 cycles (8.5μs) with NVP. Although the SRBP protocol has faster CCI roundtrips than NVP, the collective operations were slightly slower. This is caused by the concurrent load on the off-chip atomic counters. The benchmark was not applied to TACO on RCKMPI because already a single roundtrip takes longer.

Earlier experiments showed that the overhead caused by the middleware is not negligible. One of the largest processing overheads was caused by TACO's cooperative threads. Figure 8

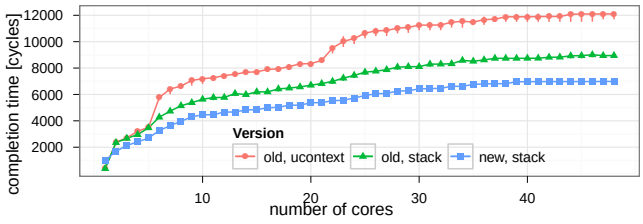

Fig. 8. Improvements (with NVP) by reduced thread switching overheads.

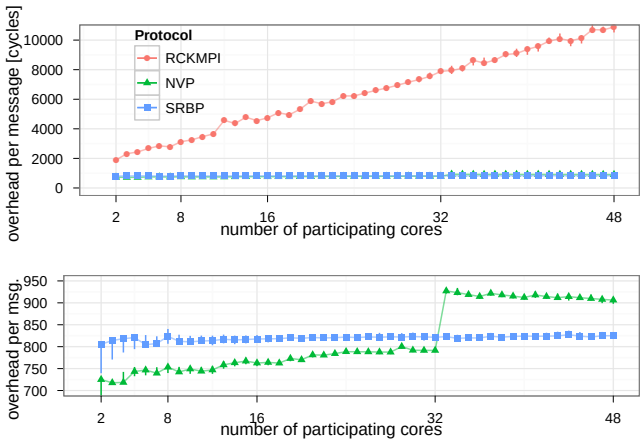

Fig. 9. Communication overhead per message with different core counts.

shows the effects of two improvements. First, the thread switching overhead was reduced by just swapping the stacks of the threads instead of using the generic `ucontext` approach. This is achieved by changing the stack pointer (ESP register) using a small assembler routine. Finally, TACO's scheduling was rewritten to avoid unnecessary thread switching and the step operation was modified to reduce the number of thread activations. The new version collects partial results in a variable and only after all results arrived, a thread is activated to send the result to the parent in the multicast tree. In combination, the performance was doubled.

C. Communication Overhead per Core

This experiment measured the CCI overhead in dependence of the core count. The cores were organized on a logical ring and the time to pass a method call through this ring was measured. Thus, each core sent exactly one message and Figure 9 shows the average overhead per message. RCKMPI's performance would be acceptable on small systems with just a few cores and with two cores it needs just 2000 cycles per message. However, the overhead per message increases by 150 cycles with each additional core, which might be caused by the growing polling overhead. Although similar to NVP, RCKMPI uses separate lines to manage point-to-point ring buffers and, thus, the polling has to check 47 lines. In contrast, SRBP's polling overhead is constant (just checks a single line) and the overhead in NVP increases slowly (one line for 32 cores). Hence, protocols with growing polling overhead will limit the scalability of parallel algorithms considerably and approaches that worked well enough on a few cores are not necessarily efficient on large many-core systems.

D. Energy Consumption

We created a tiny bare-metal environment with limited C++ support for the SCC and ported the SRBP protocol, TACO, and an event driven operating system [18] to it. At this privileged level, idle cores can be halted and the sender of a message can send an interrupt to wake up the receiver. The board management controller of the SCC platform was used for power measurements. We measured around 120W when all cores did busy waiting for new messages and just 60W when all cores halted. The CCI roundtrip benchmark increased the consumption by about 2W in both cases, while the interrupt processing overhead increased the roundtrip time by around 500 cycles. Assuming a roundtrip with halting and interrupts takes 2500 cycles and the two involved cores consume $2\text{W} + 2 * 60\text{W}/48$, this indicates that a CCI roundtrip costs around 14μJ, while it costs 17μJ without halting and interrupts. In conclusion, putting idle cores to sleep can halve their power consumption without increasing the communication costs (energy and time) much.

IV. Related Work

Several other communication libraries for the SCC exist. All of them seem to use solely the MPBT memory type, i.e. always transfer whole cache lines, and are designed for much larger messages. RCCE [19] provides access to all features of the platform, including point-to-point message exchange (without any-source polling). As the first available library, it had to focus on general features instead of special messaging needs. RCCE_comm [20] extends RCCE with collective communication, and iRCCE [3] adds non-blocking communication. The latter is also used as communication channel in SCC-MPICH. The authors of iRCCE are also working on a mailbox extension for the exchange of very small messages [21]. Internally, MPICH uses many small messages (typically 48 bytes) to coordinate collective operations and non-eager data transfers [16]. Hence, the techniques from Section II can be useful in the context of MPI implementations. The Barrelfish operating system [22] has a SCC variant [23]. Its communication protocol is a direct translation from cache coherent systems with minor modifications. However, it already shows how CCIs can be applied in operating systems for such many-core systems. Finally, a port of X10 [24] to the SCC [25] exists that is based on RCCE's protocol. Lightweight CCI protocols would fit very well to the X10 programming model. However, the current runtime environment uses the same communication channel also for large data transfers.

In summary, all of the in-memory communication protocols have a very similar structure. The design space is limited [26] and, therefore, performance gains can be reached only by restricting the intended use and careful design decisions that fit to the target hardware. The two presented protocols achieve the first by restricting the message size to the size of typical CCIs, and the latter by mixing UC and MPBT access to the communication memory and using the atomic counters for resource dispatching.

V. Conclusions

This paper presented two in-memory communication protocols that were tailored to very small messages and the special messaging support of the experimental SCC processor. We integrated these protocols into an PGAS framework and an experimental operating system kernel.

The experiments show, that for such special use cases the communication overhead can be reduced up to 15 times compared to a general purpose middleware. This makes it possible to efficiently offload tasks with much smaller workloads to other cores. Even though we used a thin (mostly inlined) software layer above the basic communication protocol, the protocol costs did not dominate the overall communication costs. Therefore, a careful design of middleware layers becomes more important with many-core processors.

Many-core systems (like the Intel SCC) are hybrid architectures that combine aspects of a distributed system with aspects of a shared memory system. Hybrid programming models that combine message passing and shared memory are needed. The main purpose of message passing shifts from data transfer to the coordination of tasks and coordination of the access to shared data because the shared memory can be used directly for data sharing. In this context, *cross core invocation* of tasks is one of the most performance critical components. This raises the need for ultra low overhead message passing protocols for very small messages (a few cache lines).

With small messages the synchronization overhead between senders and receivers is not negligible. This overhead could be further reduced by moving the notification mechanism into the hardware alongside the on-chip Message Passing Buffers. Such approaches would combine energy efficiency, efficient use of the on-chip message memory, and fast message exchange.

VI. Acknowledgements

We express our gratitude to Intel for the access to the SCC and the opportunity to contribute to its MARC (Many-core Applications Research Community) program. Most specifically we thank Werner Haas and Michael Konow from Intel Research Braunschweig, Germany, who gave us tremendous insights into the SCC architecture.

References

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011.

[2] J. D. Owens, W. J. Dally, R. Ho, D. N. J. Jayasimha, S. W. Keckler, and L.-S. Peh, "Research challenges for on-chip interconnection networks," *IEEE Micro*, vol. 27, pp. 96–108, September 2007.

[3] 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.

[4] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, pp. 18:1–18:15, 2008.

[5] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, 2007.

[6] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. IEEE, 2010, pp. 108–109.

[7] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao, "A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 64–64.

[8] A. Olofsson, "A 1024-core 70 GFLOP/W Floating Point Manycore Microprocessor," Poster on 15th Workshop on High Performance Embedded Computing HPEC2011, 2011.

[9] P. Aublin, S. Mokhtar, G. Muller, and V. Quéma, "ZIMP: Efficient intercore communications on manycore machines," Tech. Rep. [Online]. Available: http://proton.inrialpes.fr/~aublin/zimp/zimp_TR.pdf

[10] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44.

[11] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," *SIGPLAN Not.*, vol. 44, pp. 253–264, 2009.

[12] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight remote procedure call," *ACM Trans. Comput. Syst.*, vol. 8, pp. 37–55, February 1990.

[13] J. Nolte, Y. Ishikawa, and M. Sato, "TACO – Prototyping High-Level Object-Oriented Programming Constructs by Means of Template Based Programming Techniques," *ACM Sigplan, Special Section, Intriguing Technology from OOPSLA*, vol. 36, no. 12, December 2001.

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

[15] "The SCC Programmer's Guide," Tech. Rep., 2012. [Online]. Available: http://communities.intel.com/docs/DOC-5684

[16] I. A. C. Ureña and M. Gerndt, "Improved RCKMPI's SCCMPB Channel: Scaling and Dynamic Processes Support," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam*, vol. 55, 2012.

[17] R. H. Halstead, Jr., "Multilisp: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 501–538, October 1985.

[18] K. Walther, R. Karnapke, A. Sieber, and J. Nolte, "Using preemption in event driven systems with a single stack," in *The Second International Conference on Sensor Technologies and Applications*, Cap Esterel, France, 2008, pp. 384 – 390.

[19] T. Mattson and R. van der Wijngaart, "RCCE: A small library for many-core communication," Tech. Rep., 2010. [Online]. Available: http://communities.intel.com/docs/DOC-5628

[20] E. Chan, "RCCE_comm: A collective communication library for the Intel Single-chip Cloud Computer," Tech. Rep., 2010. [Online]. Available: http://communities.intel.com/docs/DOC-5629

[21] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Recent Advances and Future Prospects in iRCCE and SCC-MPICH – Poster Abstract," in *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011.

[22] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system," in *MMCS'08*, K. Schwan, D. D. Silva, and M. Milenkovic, Eds. ACM Digital Library, 2008.

[23] S. Peter, T. Roscoe, and A. Baumann, "Barrelfish on the intel single-chip cloud computer, barrelfish technical note 005," Tech. Rep., 2010. [Online]. Available: http://www.barrelfish.org/

[24] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, October 2005.

[25] K. Chapman, A. Hussein, and A. Hosking, "X10 on the single-chip cloud computer," in *X10 Workshop at PLDI'11*, 2011.

[26] R. Rotta, "On efficient message passing on the Intel SCC," in *Proceedings of the 3rd Many-core Applications Research Community (MARC) Symposium. KIT Scientific Reports*, vol. 7598. KIT Scientific Publishing, 2011.