On Efficient Message Passing on the Intel SCC

Randolf Rotta
Brandenburgische Technische Universität Cottbus
Konrad Wachsmann Allee 1
03046 Cottbus, Germany
Email: rrotta@informatik.tu-cottbus.de

***Abstract*—The Single-Chip Cloud Computer (SCC) is an experimental processor created by Intel Labs. Instead of the usual shared memory programming, its design favors message passing over a special shared on-chip memory. However, the design of efficient message passing is still an ongoing research work, because the system differs quite much from traditional hardware. This paper presents design options for message passing protocols on the SCC and discusses some implications.**

I. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. Although the SCC features shared on-chip memory, it is not intended for traditional shared memory programming. Instead, the platform is designed for research around many-core message passing concepts. In consequence, the on-chip memory, also called *message passing buffer*, is quite small and deliberately does not provide automatic cache coherence.

While the SCC provides fast access to the on-chip memory, middleware libraries have to provide actual message passing *protocols*, i.e. algorithms organizing the concurrent access to the message memory. The SCC's non-coherent memory and its high number of cores is quite different from previously known processors, and thus, porting existing message passing code from cc-NUMA systems to the SCC does not result in optimal performance (see for example [2]). Therefore, the design of efficient message passing poses a renewed design challenge: Which old or new strategies are feasible on the SCC? What is achievable within the limits of the current hardware? How much could the performance benefit from future adapted hardware support?

This paper discusses some aspects of the design space of message passing protocols on the SCC. The focus will be on non-blocking asynchronous in-order transfer of small messages (<200 bytes) between arbitrary cores, that is without explicitly established point-to-point connections. These requirements are based on active message middle-ware layers like TACO [3], but are useful for other software as well, for example MPI on top of active messages [4].

The next section discusses relevant parts of the SCC hardware, and Section III introduces some useful performance indicators. The main part is Section IV with an overview of the design dimensions and options for message passing protocols. The paper concludes with a discussion of related work and possible directions for future work.

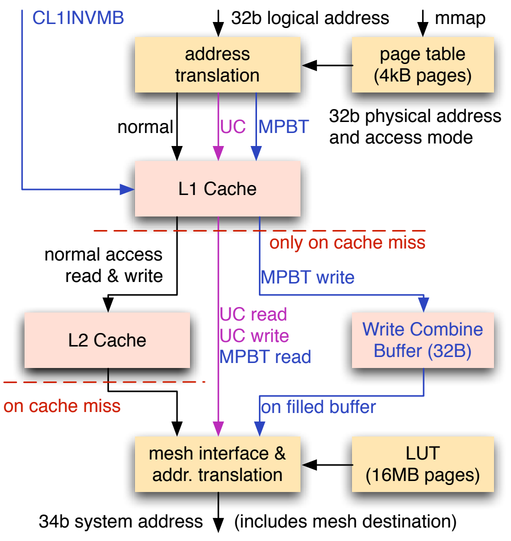

Fig. 1. Conceptual address translation and access modes on the SCC.

II. The Intel SCC

This section start with an overview of the communication capabilities of the SCC. After some notes about the performance, the section concludes with a discussion of differences to other systems.

A. Memory Access over the Mesh Network

The SCC combines 48 standard processor cores, derived from the P54C Pentium, on a single chip. All communication is carried out over a packet-switched 2D mesh network of 6×4 routers. Communication between cores is performed indirectly by writing to and reading from shared on-chip SRAM and external DRAM memory using the standard machine instructions, i.e. `MOV` with byte, word (2 byte), or double word (4 byte) granularity [5]. The destination and access mode of a request is determined in two steps as shown in Figure 1: First, the usual page table maps from the logical to the core's physical address space and sets the access mode on 4kB page granularity. Between core and router, the physical addresses are mapped to system addresses through a lookup table (LUT) with 16MB granularity. These addresses contain the mesh coordinates of the destination router and the local destination (e.g. SRAM, device registers, or attached external devices).

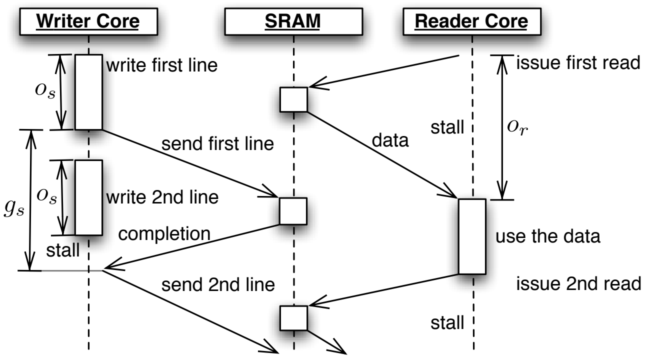

Fig. 2. Writing and reading over the mesh network.

Although all cores can access the same DRAM and SRAM memory, the system has no implicit cache coherence mechanism. Instead, a new access mode called *Message Passing Buffer Type* (MPBT) is provided. Data read from such memory is only cached in the L1 cache and the new instruction `CL1INVMB` invalidates all such MPBT lines from the cache. Write operations to MPBT memory are collected in a write-combine buffer, which tries to fill up an entire cache line before sending the data to the destination.

The usual *un-cached* memory access mode (UC) can be used as well. Data read from such memory is not cached and write operations are directly issued to the network. Concurrent writes to the same memory line do not conflict. Note that it is possible to mix MPBT and UC access to the same physical memory by mapping it twice into the logical address space. In this setting, it is just necessary to invalidate MPBT lines from the cache before accessing the UC-mapped memory.

In addition, one atomic test-and-set bit per core is available in the device registers. Reading from it activates the bit and returns its previous state. Writing resets the bit to zero. The firmware of the system interface provides a set of atomic counters. Reading from a counter increments it and returns its previous value. Unfortunately, the access is expected to be much slower than to the on-chip SRAM.

B. Performance Model for Memory Access

The LogP-model of Culler et.al. [6] summarizes the behavior of communication systems in a few parameters, namely the latency L, send overhead o_s, receive overhead o_r, gap g between subsequent messages, and the number of processors P (=cores). While it was not designed to model memory accesses, SCC's behavior can be represented quite well using the overhead and gap as depicted in Figure 2.

The P54C cores are strictly in-order and can handle only a single outstanding memory request. Write operations over the mesh are completed by a small response message from the destination. This results in write ordering when accessing different destinations, but also simplifies costs predictions. It takes time o_s to fill the write-combine buffer or issue an un-cached write operation. These overlap with the previous request, but the core stalls until the previous request is completed. The time between issuing the write and its completion is modelled by the gap g. When reading, the core stalls until the data arrived. This time is represented by the receive overhead o_r.

TABLE I
MEASURED CYCLE COUNTS FOR WRITING AND READING OVER THE MESH. RANGES REFER TO THE SMALLEST AND LARGEST DISTANCE.

	1 byte (UC)	4 byte (UC)	32 byte (MPBT)
o_s	5	5	28
g_s	48–81	48–81	75–105
o_r	53–86	53–86	58–88

All three parameters depend on the memory type (MPBT or UC) and the mesh distance. Micro-benchmarks like [7] can be used to estimate the actual parameters. Table I provides cycle counts for the chip configuration with 800 MHz core clock rate and 1600 MHz mesh clock rate.

C. Differences to other systems

An ultra low latency network: With networks like Infiniband the hardware latency (especially between CPU and network controller) dominates most other costs. In combination with high processor speeds, differences between message passing implementations are quite small. In addition, the network already implements many details, for example, hardware-managed message queues. In contrast, on the SCC, the communication latency is quite low, but several communication steps are necessary for a single message transfer. Thus, the performance differences between protocols are much larger.

A different memory model: Most message queue designs for shared memory are based on cache coherent memory and compare-and-swap operations (see for example [8], [9]). The SCC provides no hardware cache coherence. Thus, the location of data in the mesh is known exactly and all data transfers are triggered explicitly. This also eliminates false-sharing problems. Unfortunately, no remote compare-and-swap is available on the SCC, but can be emulated with the lock registers. In contrast to traditional cache-coherent systems, it is much more efficient to use un-cached writes to update individual values directly in the memory instead of performing cache line round-trips. Combining un-cached and MPBT access to the memory provides new design opportunities.

Finally, the scalability of the overal message passing becomes an issue: The increasing number of cores results in an even faster growing number of communication partners and managing these connections can quickly become a bottleneck in respect to computational overhead and memory usage.

III. COMMUNICATION PATTERNS

The LogP model was designed to describe essential performance characteristics of communication systems. Based on the mesh parameters (Table I), predictions of the mesh-induced protocol overhead O_s, O_r, gap G and latency L are easily calculated.[1] However, the raw LogP parameters are difficult to

[1]Here, uppercase letters denote parameters of a protocol while lowercase letters are used for the mesh. The receive overhead O_r is the time to process a transmitted message.

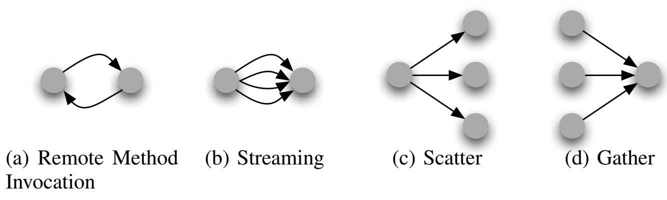

Fig. 3. Communication patterns.

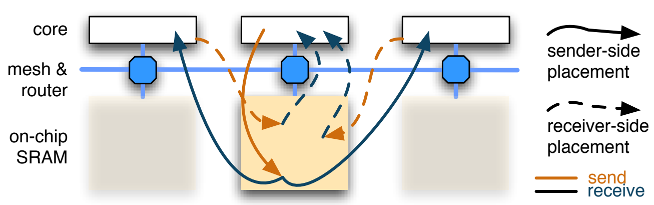

Fig. 4. Message placement on receiver vs. sender side.

interpret. This section introduces four communication patterns that are used to investigate specific aspects and implications of a protocol's performance (Figure 3).

The first pattern, called *Remote Method Invocation* (RMI), describes a function call with result value, which is executed on a remote core. It is similar to message roundtrips and its completion time is roughly $T_{\text{RMI}} = 2(O_s + L + O_r)$.

A second scenario are distributed processing pipelines, where the cores send and receive continuously *streams* of data or asynchronous remote method calls. For small messages like method calls, the steady-state *message throughput* is of interest, which should be about $\text{TP} = 1/(O_s + G)$ messages per cycle. For large data transfers that are split into smaller messages, the *bandwidth* ("goodput") is more interesting. It should be about $\text{BW}_n = n/(O_{s,n} + G_n)$ bytes per cycle when messages of n bytes payload are used.

Some applications use collective operations in which a task is initiated by a core and then performed in parallel on a group of cores. This involves propagation of the task (multicasting) and possibly waiting for its completion (collecting and merging results). Unfortunately, the completion time depends on the multicast topology and the optimal topology depends on the protocol parameters. Instead, the collective operations are dissected into their basic local communication patterns: The *Scatter* pattern delivers a message to k direct receivers. Its completion time at the sender is roughly $k(O_s + G)$ and the arrival time at the last receiver is $k(O_s + G) - G + L + O_r$. The inverse direction is the *Gather* pattern. Its completion time is kO_r (processing k transmitted messages).

Obviously, all of these performance indicators depend on the message size and the mesh distance. More importantly, they also depend on details of the protocols and thus the above formulas provide just a rough impression.

IV. A Design Space for Message Passing

This section presents a design space for message passing protocols and discusses several available options in each design dimension. The discussion begins with the software level at which message passing could be implemented (Section IV-A). The next sections discuss *message placement* (Section IV-B) and *memory allocation* (Section IV-C).

Some kind of flow control is necessary to protect against overwriting of unprocessed messages, which could happen by two cores writing concurrently or by one core writing too early to the same place. This protection is achieved by notification and acknowledgement mechanisms: *Notification mechanisms* signal the arrival of new messages and have to ensure that no message is missed (Section IV-D). *Acknowledgement mechanisms* signal which messages have been processed and thus enable the save reuse of message memory (Section IV-E). When all message slots are in use or no notifications are currently possible, a sender cannot send further messages without conflicts. This is resolved by *wait mechanisms* (Section IV-F).

A. Levels of Abstraction

At which software layer should the message passing be implemented? Independent *one-to-one* queues are easier to adapt to specific communication needs as they can be mixed in the same application. *Many-to-one* protocols exploit that they will receive messages from multiple sources and *one-to-many* protocols optimize sending to different destinations. Combining both into *many-to-many* protocols provides most chances for exploiting synergies. While such approaches complicate incorporating application specific knowledge, their performance might make this unnecessary.

For example, with independent one-to-one queues, the memory requirements grow quadratically with the number of cores (at least $P^2 - P$ individual queues for P cores) and polling slows down linearly ($P - 1$ queues to check). Checking for messages on the SCC while using only MPBT memory takes at least 3200 cycles (47 cores times 58+9 cycles for fetching a line and testing the content). Combining the notification mechanism of all queues into a many-to-one protocol, allows to reduce this overhead to 220 cycles and possibly even lower as will be discussed in Section IV-D.

In summary, efficient protocols will most probably require global approaches. These cannot be implemented at the level of individual message queues inside applications, but should be provided as a shared service by the system.

B. Placement: Pulling vs. Pushing Messages

With *receiver side placement* (aka push mode) the sender writes the message payload into the remote receiver's memory. In contrast, with *sender side placement* (aka pull mode) the message is put in the sender's local memory.

For the method call and streaming patterns there is no direct difference: On the SCC, a remote write with a local read takes as long as a local write with a remote read. The effort just shifts between sender and receiver. However, for the scatter pattern, the sender side placement is better (Figure 4): Local writing reduces the send overhead and gap (sequential local writing), while the more expensive remote transfer to the receivers is parallelized. In contrast for the gather pattern, the receiver side placement is better: Sequential reading is faster on local memory, while the senders would do their remote writing in parallel. What a doubtful choice: When a scatter is followed by a gather, both effects cancel out each other. Therefore, a protocol can exploit the parallelism of appropriate message placement only by breaking the symmetry.

C. Allocation: Managing the Message Memory

This design dimension is concerned with the allocation of message memory. Here, we consider a fixed amount of fixed size *message slots* per core, which reside in the on-chip SRAM. The main focus of this subsection lies in quick memory allocation. A separate acknowledgement mechanism will be necessary to reuse the slots.

1) Static Allocation: Each core uses a specific slot depending just on the destination. In the simplest case just a *single slot* is used for all destinations. In consequence, a new message can be sent only after the acknowledgement of the previous message and thus the send gap G is at least $L+O_r$. This effect is decreased by using a *separate slot per destination*. Then, subsequent messages to the same destination still have the longer gap $G_{\text{same}} = L+O_r$, but sending to varying destinations has a smaller gap $G_{\text{other}} \ll G_{\text{same}}$.

The single slot approach requires just P slots (one for each core), and P^2-P slots are required when using separate slots. Separate slots improve the performance of the gather pattern, while streaming remains inefficient because of the larger gap.

2) Static Allocation by Direction: Each core owns a sender side and a receiver side set of separate slots per destination. The sender side slots are used for scatter communication and the receiver side slots are used to send gather messages. Other messages can use any of the two possible slots. This increases the scatter and gather performance and reduces the send gap problem slightly, but requires even more slots (in total $2P^2$).

3) Receiver-based Allocation: The sender acquires a slot from the receiver, for example, by exploiting the notification mechanism. This works best with receiver side placement.

4) Sender-based Round-Robin: Each sender has a set of slots in its local memory (sender side placement) and the slots are used in a round-robin fashion: Before using a slot, the sender checks for the acknowledgement of the slot's previous message. If not yet free, the next slot is tried. Under normal conditions it should take some time until a slot is revisited and thus the first slot tried is free with a high probability.

With this approach, the sender has to wait just when all slots were checked without success. Thus, especially the streaming throughput and bandwidth is improved in comparison to static allocation. A further advantage is the gained control over the size of the message memory, because it no longer depends on the number of cores. Instead, the number of slots per core can be chosen freely and more slots decrease waiting times.

This idea can be extended to variably sized slots. While this increases the memory utilization even further (less bandwidth degradation for small message sizes), the additional management overhead might increase the send overhead too much.

D. Notification: Discovering new Messages

The notification mechanism's task is to signal the arrival of messages and discovering these at the receiver.

1) Separate Notification Flags: Each core d owns an array of notification flags $N_d(i)$. Using SCC's un-cached write operations, a sender s can set his flag $N_d(s)$ at receiver d without interfering with other cores. Let non-zero values indicate arrived messages. The receiver d polls for messages by scanning N_d, which is speed up by fetching whole lines (i.e. at most 32 flags, each one byte) at once by reading from the MPBT-mapped memory. Therefore, the mesh-induced polling overhead is $\lceil P/32 \rceil o_{r,32}^{\text{local}}$, i.e. 116 cycles. This approach is efficient on the SCC, because the senders do not suffer from false-sharing as would be the case on cache-coherent systems.

The search for notifications is accelerated by looking for nonzero double words first, which reduces the tests from 48 to 12 in case no message arrived. Our benchmarks showed about 220 cycles polling overhead with this method. The idea could be extended to notification trees, reducing the necessary tests to $\mathcal{O}(\log P)$. However, this increases the memory usage and computational overhead considerably.

2) Single Flag with Locking: SCC's 48 atomic locks enable the following strategy [10]: The sender acquires the destination's lock and then writes its notification into a single fixed flag. The receiver polls by reading just this single flag. When a notification was found, the receiver resets the flag and releases the lock. This would yield nearly optimal RMI and scatter performance, but the streaming and the gather performance would be quite bad. Without contention, the notification overhead for the sender will be about 106–172 cycles (lock, write flag). Polling needs just about 53 cycles to check the flag. The acknowledgement overhead will include 106 cycles (reset flag, unlock).

3) Notification Ring Buffer: The read position is incremented only by the owner, while the write position is incremented by all senders and thus ideally is an atomic counter. The buffer is full, when the difference between the acquired write position and the current (or last known) read position is larger or equal the buffer size. Then, a sender must not write to its flag, but new senders shall still be able to reserve a flag by incrementing the write position. This can be achieved with full 32 bit counters and computing the actual flag position just for writing. Note that the receiver cannot compare read and write positions to find new messages, because the sender writes the actual notification after incrementing the counter.

Compared to the previous notification mechanisms, this one is scaleable: It supports multiple concurrent senders while the memory requirements and polling overhead are independent of the number of cores. Emulating the counter with locks results in 265–430 cycles notification overhead (lock, fetch write position, write new position, unlock, write flag). Using the hardware atomic counters should be faster.

4) Message Linking: The previous approaches can be combined with this method in order to increase the throughput and reduce the congestion as follows: Instead of acquiring a new notification flag, each message contains an additional notification flag. The sender writes the location of the new message into the flag of its previous message sent to the same destination, thus forming a linked list of messages. The receiver discovers the first message using the primary notification mechanism. Then it processes each message in the list until reaching the last message and continues with the primary notification mechanism. Special care is necessary with the acknowledgements, because the receiver may see the end of the list while the sender appends a new message.

E. Acknowledgement: Freeing Memory

Once a message is processed by the receiver, it is necessary to give the memory back to its owner for use in future messages. This information can be placed in a separate array of *acknowledgement flags* at the receiver or sender side with one byte per message slot. Alternatively, the structure of message headers can be exploited. For example, active messages contain a non-zero pointer to a handler function and resetting it to zero provides an *in-message acknowledgement*.

The owner of a message may *actively check* the flags during polling. In that case it is more efficient to check just the outstanding acknowledgements. However, because the chance of an acknowledgement increases with time, it is even more efficient to just *check on-demand* as late as possible.

A separate array of byte-sized acknowledgement flags is an interesting option in combination with a dynamic slot allocation, because the fast search method for notification flags (Section IV-D1) can be applied to quickly collect acknowledged slots. For this purpose, acknowledgements are represented by non-zero flags. The collector resets these flags to zero and adds the slots to a free-list. This collection can be performed during idle time and when the free-list is empty.

F. Waiting: Handling full Queues

A sender has to wait when all message slots or notification flags are currently in use. Simple busy waiting and any other blocking behavior is not sufficient as it would lead to deadlocks when two cores try to send a message to each other.

Depending on the middleware framework, several options are available. The sender might just do *repeated polling* until the message can be sent. This has the lowest overhead, in case the channel becomes free again very soon. A second option is to *temporarily suspend* the thread (or coroutine) that issued the message and thus other work can proceed. Once the scheduler activates the thread, it will retry sending the message. However, if all threads are waiting, polling is necessary. Alternatively, the thread can be *completely suspended*. The protocol has to wake up the thread when the channel becomes free, which could be detected from incoming acknowledgements. Instead of exploiting threads, *sender side message queues* can store the pending messages. During polling, the channels could be checked and the next message sent. When this queue is full as well, the protocol has to revert to one of the above strategies.

V. Related Work

The design options presented in the previous section can be used to characterize existing protocols and get a quick impression of their relative performance. In this section, the current state regarding existing Scc software is discussed, namely X10, Barrelfish, RCCE, Rckmpb, RCKMPI, and TACO.

X10 is a parallel object-oriented programming language targeted to multi-core systems [11]. The X10 implementation for the Scc [12] uses sender side message placement, separate notification flags at the receiver side and separate acknowledgement flags at the sender side. The flags are probably accessed through the MPBT mode and are stored in separate cache lines. Thus, polling for new messages will cost about 3200 cycles (see Section IV-A), i.e. the RMI roundtrip time is at least 6400 cycles.

The *Barrelfish* operating system has a Scc variant and first benchmarks reported an average message round-trip time of 8746 cycles [2], including the operating system's scheduling overhead. The message data is placed in shared off-chip DRAM and the on-chip SRAM is used just for notification. The notification mechanism uses the inter processor interrupt (IPI) and a MPBT-based ring buffer that is protected against concurrent writing by the receiver's lock register. Therefore the polling overhead will be low (58 cycles to inspect the cache line at the current read position), but the notification overhead is at least 660 cycles (lock, fetch r/w positions, write notification, update write position, unlock, send IPI).

RCCE is a message passing library for the Scc [13]. The low level interface provides put and get operations to the on-chip SRAM and synchronization flags. On top of this, blocking send/receive operations with sender-side message placement are implemented. The synchronization flags are used for notification and acknowledgement, but any-source polling is currently not supported. Applications have to probe for each possible source sequentially and incoming messages can be processed quickly just as long as the sender is known.

The *Rckmpb* driver provides TCP/IP networking between Linux instances running on the Scc cores [13]. It uses sender side placement with round-robin allocation and receiver side separate descriptor flags (each one byte), which are used for notification and acknowledgement. Polling just reads the two cache lines of the descriptor flags (see Section IV-A). Individual flags are updated without locking by exploiting the behavior of the Write Combine buffer (similar to a UC write).

RCKMPI is a MPI implementation for the Scc that adapts the CH3 streaming channel of MPICH [14]. It uses receiver side placement with static allocation in separate slots. Flow control is managed with sequence counters, which are stored in separate cache lines. As with all protocols that use separate cache lines for notification, polling is particularly slow.

We implemented an over-simplified protocol (Figure 5) for the Taco framework [15]. It uses receiver side placement with static allocation, separate notification flags, separate on-demand acknowledgement flags, and waiting by polling. The implementation has a send overhead of $O_s = 250$ cycles, a latency of $L = 600$ cycles, a send gap of $G_{\text{same}} = 880$ cycles, and a roundtrip time of $T_{\text{RMI}} = 1770$ cycles (all values include the middleware overhead). Although this protocol seems to perform very good, it is still not optimal. For example, the

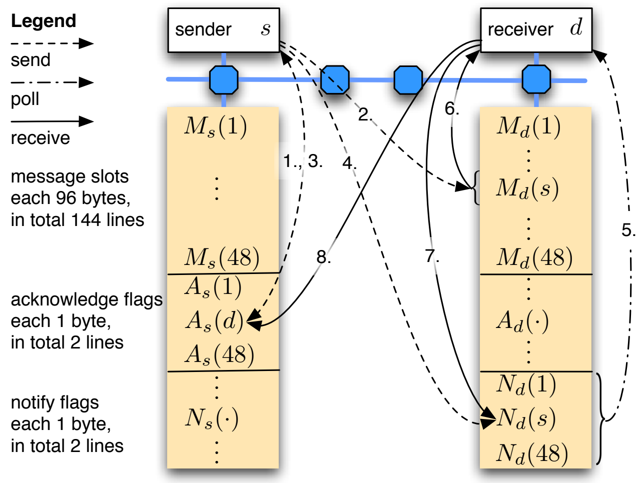

Fig. 5. Exchange of an one-way message over the SRAM.

used allocation and notification mechanisms are not scalable, and the acknowledgement flags could be replaced by checking the notification flags or in-message flags.

VI. Summary and Future Work

Various aspects of message passing on the SCC can be organized in a design space with six dimensions: Abstraction Level, Placement, Allocation, Notification, Acknowledgement, and Waiting. Flow control is a cross cutting concern connecting Allocation, Notification and Acknowledgement. The Level of Abstraction is not an independent dimension, but classifies protocols into a range from independent one-to-one channels to global unified many-to-many protocols.

Future work and collaboration is necessary in the exploration of actual protocols. Cost predictions and microbenchmark results for individual design options could direct the search towards better solutions. Obviously, some choices conflict, require additional steps to work together, or provide chances for optimization. Thus, the real performance can be compared only on real implementations. To date just few and quite similar protocol implementations exists. Protocols should not be compared on their LogP parameters alone. As discussed, a collection of typical usage patterns will be necessary to provide a more detailed insight into performance differences. The performance indicators should be extended to also compare the power consumption.

The design space can be expanded by new hardware features, e.g. compare-and-swap implemented directly in the on-chip SRAM, or hardware-based notification queues. This would allow to predict the improvements over existing protocols even before the actual hardware exists.

Finally, as the message passing is a critical system component and protocols become more complicated, it would be helpful to verify properties like wait-freeness, lock-freeness, or in-order delivery with formal methods.

Acknowledgements

We wish to express our gratitude to our master students Jana Traue, Thomas Prescher and Peter Sauer, who contributed a lot to the discussions and benchmarks. Furthermore we thank Intel for the access to a SCC and the opportunity to contribute to its MARC program. Most specifically we thank Werner Haas and Michael Konow from Intel Research Braunschweig, Germany, for the tremendous insights into the architecture.

References

[1] 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.
[2] S. Peter, T. Roscoe, and A. Baumann, “Barrelfish on the Intel Single-chip Cloud Computer, Barrelfish Technical Note 005,” Tech. Rep., 2010.
[3] 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.
[4] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken, “Low-latency communication on the IBM RISC system/6000 SP,” in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1996.
[5] T. G. Mattson, R. F. 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 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.
[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “LogP: towards a realistic model of parallel computation,” in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 1993.
[7] D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa, “Assessing fast network interfaces,” *IEEE Micro*, vol. 16, pp. 35–43, February 1996.
[8] M. M. Michael and M. L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1996, pp. 267–275.
[9] W. N. Scherer, III, D. Lea, and M. L. Scott, “Scalable synchronous queues,” *Commun. ACM*, vol. 52, pp. 100–111, May 2009.
[10] K. E. Schauser and C. J. Scheiman, “Experience with active messages on the Meiko CS-2,” in *Proceedings of the 9th International Symposium on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 140–149.
[11] 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,” in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 519–538.
[12] K. Chapman, A. Hussein, and A. Hosking, “X10 on the SCC,” Presentation, 2011. [Online]. Available: http://communities.intel.com/docs/DOC-6255
[13] 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, February 2011.
[14] I. A. Comprés Ureña, “Lightweight MPI for the Single Chip Cloud,” Presentation, 2010. [Online]. Available: http://communities.intel.com/docs/DOC-5844
[15] R. Rotta and J. Nolte, “Low latency collective operations on the Intel SCC,” 2011, work in progress.