On mapping distributed S-Net to the 48-core Intel SCC processor

Verstraaten, M.; Grelck, C.; van Tol, M.W.; Bakker, R.; Jesshope, C.R.

Publication date
2011

Published in
3rd Many-core Applications Research Community (MARC) Symposium

On Mapping Distributed S-NET to the 48-core Intel SCC Processor

Merijn Verstraaten, Clemens Grelck, Michiel W. van Tol, Roy Bakker, Chris R. Jesshope
Informatics Institute, University of Amsterdam
Science Park 904, 1098 XH Amsterdam, The Netherlands

***Abstract*—Distributed S-NET is a declarative coordination language and component technology primarily aimed at modern multi-core/many-core chip architectures. It builds on the concept of stream processing to structure dynamically evolving networks of communicating asynchronous components. These components themselves are implemented using a conventional language suitable for the application domain. Our goal is to map Distributed S-NET to the Intel SCC processor in order to provide users with a simplified programming environment, yet still allowing them to make use of the advanced features of the SCC architecture.**

Following a brief introduction to the design principles of S-NET, we sketch out the general ideas of our implementation approach. These mainly concern the use of SCC's message passing buffers for lightweight communication of S-NET records and control data between cores as well as remapping of large data structures through lookup table manipulation. The latter avoids costly memory copy operations that would result from more traditional message passing approaches. Last, but not least, we present prototypical performance measurements for our communication primitives.

I. 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. It provides 48 P54C cores, an on-chip message passing network, non cache-coherent off-chip shared memory and dynamic frequency and voltage scaling on different subsets of cores [1]. Creating programs for systems which support large amounts of parallelism is already difficult; even in the absence of specific exploitation of the power saving features. Trying to also utilise these power management features at the same time complicates the programmer's job considerably.

Currently RCCE [2] (including the community contributed iRCCE [3]) and RCKMPI [4] are available for programming the SCC. RCCE is an SCC-specific low-level message passing library and RCKMPI is an adaptation of MPI largely based on RCCE. To the best knowledge of the authors, no higher level programming environments are (yet) available on the SCC. While message passing is a very efficient way programming it also requires a lot of attention to detail from the programmer. Our goal is to make this task simpler.

S-NET[1] [5], [6], [7] is a coordination language whose aim is to simplify the programming of parallel systems. As a pure coordination language S-NET provides no features to express any sort of computation; it relies on an auxiliary language for the implementation of sequential, state-less components. S-NET in turn organises the interaction of these components — called boxes — in a streaming network. A type system on streams gives essential static guarantees on the behaviour of S-NET streaming networks.

Our two step approach, internally sequential components on the one hand and orderly component interaction, somewhat reflects the hardware design principle of the SCC and other contemporary multicore chip architectures: cores that were originally designed as central processing units combined on a single die.

This paper reports on our on-going efforts to map S-NET to the Single Chip Cloud Computer as an alternative programming environment. Starting from our MPI-based distributed implementation of S-NET [7] the obvious choice is to simply use RCKMPI as underlying middleware layer. However, the performance of MPI is less than what can be achieved by an implementation which uses the SCC's hardware features directly. Additionally MPI restricts us to a static number of nodes, which is something which is undesirable in light of future development directions. Instead we focussed our efforts on implementing asynchronous message passing using the message passing buffers (MPBs) as communication channel and using interprocessor interrupts as out-of-band notifications. To accomplish this we had to come up with a way for our userspace code to receive the interrupts which are being trapped by the kernel.

In this paper we will analyse the approaches we considered and their impact on our goal of having a programming model which is more convenient to use than writing programs directly using RCCE or MPI, while still capable of using the features provided by the SCC. We will begin with an overview of both S-NET (Section II) and Distributed S-NET (Section III) before we discuss the effects of the SCC's design on our design (Section IV). We will discuss our initial measurements in Section V and conclude in Section VI.

II. S-NET

S-NET turns functions written in a standard programming language (e.g. C) into asynchronously executed, stateless stream-processing components termed *boxes*. Each box is connected to the rest of the network by two typed streams, one for input and one for output. Messages on these typed streams are organised as non-recursive records, i.e. sets of label-value pairs. The labels are subdivided into *fields* and *tags*. Fields are

[1]The development of S-Net has been funded by the European Union through the FP-6 project Aether and the FP-7 project Advance.

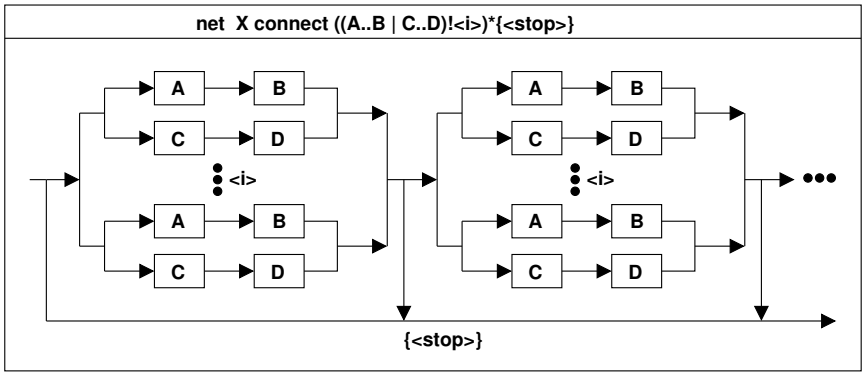

Figure 1. Example of an inductive streaming network constructed using network combinators. A, B, C and D denote boxes or networks defined elsewhere. Then A..B|C..D denotes the subnetwork where any record either goes through A and then B or through C and then D. The actual routing depends on the types of A, B, C and D and is left out in this figure. The whole subnetwork is then replicated in parallel based on the index i carried along by all records in the system. At last, this network is replicated serially with the presence of stop tag acting as a dynamic drop-out condition for a records after having passed each instance of the replicated network.

associated with values from the box language domain, they are entirely opaque to S-NET. Tags are associated with integer numbers that are accessible on both the coordination and on the box level.

A box triggers when it receives a record on its input stream, the box then applies its box function to that record. In the course of function execution the box may output records to its output stream. Once the function has finished the S-NET box is ready to receive and process the next record on the input stream.

On the S-NET level a box is characterised by a *box signature*; a mapping from an *input type* to a disjunction of types, named the *output type*. For example,

```
box foo ((a,<b>) -> (c) | (c,d,<e>));
```

declares a box that expects records with a field labeled `a` and a tag labeled `b`. The box responds with an unspecified number of records that either have just field `c` or fields `c` and `d` as well as tag `e`. The associated box function `foo` is supposed to be of arity two: the first argument is of type `void*` to qualify for any opaque data; the second argument is of type `int`.

The box signature naturally induces a *type signature*. For a proper specification of the box interface it is essential to have a concrete ordering of fields and tags. However, on the S-NET level we ignore the ordering when reasoning about boxes: treating them as sets of labels instead of tuples of labels. Hence the type signature of box `foo` is {a,<b>} -> {c} | {c,d,<e>}.

This type signature states `foo` accepts *any* input record that has *at least* field `a` and tag `b`, but may well contain further fields and tags. The formal foundation of this behaviour is *structural subtyping* on records. Any record type t_1 is a subtype of t_2 iff $t_2 \subseteq t_1$. This subtyping relationship extends to multivariant types. A multivariant type x is a subtype of y if every variant $v \in x$ is a subtype of some variant $w \in y$.

Subtyping on input types of boxes raises the question what happens to the excess fields and tags. These are not just ignored in the input record of a network entity, but are attached to any outgoing record produced by it in response to that record. Subtyping and flow inheritance are indispensable when it comes to getting boxes that were designed separately to work together in a streaming network.

It is a distinguishing feature of S-NET that it neither introduces streams as explicit objects nor defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-NET provides four network combinators: static serial and parallel composition of two networks and dynamic serial and parallel replication of a single network. These combinators preserve the SISO property; any network, regardless of its complexity, again is a SISO entity.

Let `A` and `B` denote two S-NET networks or boxes. Serial combination (`A..B`) constructs a new network where the output stream of `A` becomes the input stream of `B`, and the input stream of `A` and the output stream of `B` become the input and output streams of the combined network, respectively. As a consequence, `A` and `B` operate in pipeline mode.

Parallel combination (`A|B`) constructs a network where incoming records are either sent to `A` or to `B` and the resulting record streams are merged to form the overall output stream of the combined network; the type system controls the flow of records. Each network is associated with a type signature inferred by the compiler. Any incoming record is directed towards the subnetwork whose input type better matches the type of the record. If both branches match equally well, one is selected non-deterministically.

The parallel and serial combinators have infinite counterparts: serial and parallel replicators for a single subnetwork. The serial replicator `A*type` constructs an infinite chain of replicas of `A` connected by serial combinators. The chain is inspected before every replica to extract records that match the type specified as the second operand.

The parallel replicator `A!<tag>` also replicates network `A` infinitely, but the replicas are connected in parallel. All incoming records must carry the tag; its value determines the replica to which a record is sent.

Fig. 1 shows an example for the use of network combinators to construct streaming networks.

Whereas boxes can easily split one incoming record into multiple records sent to the output stream one after the other, the opposite operation, i.e. merging two records on the input stream to form a single record on the output stream, is not possible with the means of S-NET introduced thus far. Merging independent records is the essence of synchronisation in the streaming context of S-NET. To isolate this coordination-level feature as far as possible from the box or application level, S-NET uses a special built-in box for this purpose called *synchrocell*. For example, the *synchrocell* {[|{a,b},{c,d}|] awaits records containing fields a/b and c/d and merges them into a record containing all four fields. For an exact definition of synchrocells and an extended discussion of the whole issue of synchronisation in the context of S-NET see [8].

We refer interested readers to [6], [5] for a detailed account of the design of S-NET and to [9], [10] for application examples.

III. DISTRIBUTED S-NET

S-NET as described so far is an abstract notation for streaming networks of asynchronous components. There is no notion of computing resources in S-NET, nor does S-NET make any specific assumptions about the execution environment.

Distributed S-NET [7] is a conservative extension of S-NET that introduces the concept of abstract compute nodes as an organisational layer on top of the logic network of boxes defined by standard S-NET.

We introduce two *placement* combinators. *Static placement* places the execution of a box or a network onto a logical node. *Indexed dynamic placement* places the execution of a box or a network onto a logical node on a per-record basis, based on a specific tag of that record. More precisely, each record is routed through a replica of the box or network instantiated on the relevant logical node.

We deliberately restrict ourselves to plain integer values for identifying nodes to retain the advantages of an abstract model as far as possible. The concrete mapping of numbers to machines is implementation-dependent. Our prototype implementation of Distributed S-NET is based on MPI where numbers correspond to MPI task identifiers.

Implementation-wise, Distributed S-NET takes care of initially setting up and dynamically maintaining disconnected S-NET streaming graphs on multiple nodes. Input and Output managers at node-boundaries render the distributed memory transparent by automatically serialising, transmitting and deserialising S-NET records when moving from one node to another.

While tags are easily sent around the network, field data can potentially be large. In this case serialisation, transmission and deserialisation at every node boundary between the creating node and the consuming node of some field would be costly. We instead transmit a qualified reference which allows the actual data associated with some field to be fetched, on-demand, from its current location to where it is needed.

Interested readers are referred to [7] for a more thorough introduction of the design and implementation of Distributed S-NET. Several case studies in [7] as well as [11] illustrate that implementing real-world distributed applications in Distributed S-NET is easier than using a low-level message passing middleware directly.

IV. S-NET ON THE SCC

Our initial work focussed on preparing the S-NET codebase to support multiple implementations of its distribution layer, implementing and experimenting with communication primitives on the SCC and later, when it became available, getting the current MPI implementation running on the SCC using RCKMPI. As mentioned in Section II, the MPI implementation of Distributed S-NET copies (potentially large) application data structures by (de-)serialising them over MPI messages. This is an expensive operation which can and should be avoided on a shared memory machine like the SCC.

Instead of copying it should be much faster to remap the memory from one core to another using the programmable lookup tables (LUTs). These control the translation of memory requests from "virtual" addresses to actual physical memory addresses. This means that changing an entry in the LUTs lets us move data in and out of a core's visible memory space without having to actually copy or move the data.

One problem with this approach is that the LUTs are behind the L2 cache. This mean cache hits and misses are checked *before* the final physical address look up is done. Hence, the L2 cache may contain stale data, e.g. the cache contains data at virtual address x (which translates to physical address y), meanwhile the LUTs have been changed so virtual address x translates to physical address z. When the core attempts to read from virtual address x it will still read the cached data at y instead of z.

This isn't a problem if it is possible to — efficiently — invalidate or flush the relevant L2 entries (or the entire L2). However, since the P54C cores of the SCC never had an L2 cache they don't have such an instruction. There is an L2 reset pin available on the machine, but using it crashes the core. This leaves two possible solutions for remapping memory: marking the used address range as uncacheable or flushing the L2 after every remap. Since there is no flush instruction the only way to accomplish a flush is to manually read in a full cache worth of data, thereby evicting the entire contents of the cache.

Aside from the big field data S-NET also sends a large amount of smaller meta-data messages consisting of tags. We want to be able to quickly deal with these messages, while wasting as little time as possible on this "non-work" computation.

Secondly, we need to be able to deal with multiple incoming message queues. This is important because S-NET utilises fixed size buffers, which propagate back pressure through the network, to prevent a box which produces huge number of output records from overwhelming the network.

If communication to one of the incoming queues can also block communication to other queues, then the network suddenly becomes susceptible to deadlocks. For example, imagine we use a synchronous communication method (like RCCE) or an asynchronous communication method which uses the MPB as a single queue for all incoming messages. When a node blocks on a send — because it is synchronous or the

receiving MPB queue is full — the network deadlocks when further progress of the network depends on the same node (further down the network) doing a receive.

Since a node can have an arbitrary number of incoming message queues, the MPB with its limited size is not suitable for holding all incoming queues for a node. A node should move incoming messages from the MPB to queues in its own memory as soon as possible to keep receive space free. There are two obvious mechanism for a core to know when to move messages from the MPB: the core can either occasionally poll its MPB to see if there is anything new or we can use an out-of-band signal, in the form of an interprocessor interrupt.

The downside of polling is that when done infrequently it can take too long to make room in the MPB for new messages. If done too frequently time and energy are wasted. The interrupt based approach means having to context switch to the kernel to handle the interrupt and then context switch back to the program. The viability of this later option depends on the cost of a context switching.

Once the more basic work of getting S-NET running on the SCC is finished we intend to investigate the possibilities of the power management features. Since not all tasks take the same time and S-NET propagates back pressure through the network; there will always be one or more bottlenecks in a network, potentially leaving the parts of the network which are behind this bottleneck idle. The runtime system should be capable of lowering the clock speed on these idle nodes to save power, while increasing it on the bottleneck nodes to increase throughput of the network.

This functionality could then be extended to take into account that S-NET networks are dynamic, they grow and shrink as the computation moves through phases. Dynamically splitting and merging networks into smaller or bigger components would allow the runtime to allocate more cores to bottleneck tasks or deallocate cores from idle components, potentially even shutting down cores entirely or starting them up as computation demands.

V. MEASUREMENTS

As discussed in the previous section, there are only two ways to deal with the possibility of stale data in the L2 cache when remapping memory using the LUTs. Manually flushing the entire cache or marking the memory as uncacheable. Our measurements show that it takes approximately 1 million cycles to flush the cache manually; plus the cost of evicting still useful data, which has to be fetched from memory again.

The difference between L2 cache hits and L2 cache misses/uncached memory is a factor 6 writing speed reduction. Uncached memory and L2 cache misses write at a speed of up to 20 MB/s, cache hits write at up to 125 MB/s. For reading the speed difference is a factor 14. Uncached memory and L2 cache misses read at up to 20 MB/s whereas cache hits read at up to 285 MB/s.

The uncached speeds translate to a read and write speed of approximately 102 cycles per 4 bytes, or 25.5 cycles/byte. This means it costs 51 cycles in total to transfer one byte using uncached memory, from here we can see that the performance penalty of disabling caches start to outweigh the 2 million cycles required to flush two L2 caches once we want to transfer more than 38 KB.

Hops	Remote	Kernel	Roundtrip
0	2.8	4.6	7.4
1	2.9	4.5	7.5
2	3.0	4.6	7.6
3	3.0	4.7	7.7
4	3.1	4.7	7.8
5	3.0	4.6	7.6
6	3.0	4.7	7.7
7	3.0	4.6	7.6
8	3.1	4.6	7.7

Table I
INTERPROCESSOR INTERRUPT LATENCIES (IN 1000 CYCLES)

The Barrelfish developers already did some measurements [12] with regard to the speed of notifications using the MPBs and the latencies of interprocessor interrupts (IPI). But since we plan to run in user space under Linux, unlike Barrelfish, we are interested in how much overhead we would incur doing this. We did all our measurements on a system running the cores at 533 MHz and mesh network and DDR controllers running at 800 MHz.

To free up one of the two available interrupt pins we modified the rckmb driver to expose a kernel parameter (using sysfs) which lets us switch between interrupt-driven and polling mode. Next we wrote a simple Linux kernel module which would register a handler for the freed interrupt. This handler simply sends a POSIX signal to a user process; this way we can use interrupts to effectively send POSIX signals between processes running on different cores. Which signal is send and to which user process it is sent is configured via kernel parameters exposed using sysfs.

After the kernel module was done we had the basic infrastructure needed to test the latencies of IPIs. Each core, except core 0, was running a simple program with an infinite loop doing nothing and an installed signal handler which would raise an interrupt on core 0. Core 0 was running a program which would run through 100,000 iterations of the below steps for every core:

1) Read the cycle counter
2) Raise an interrupt on the other core
3) Read the cycle counter upon entering the interrupt handler in the kernel
4) Read the cycle counter again upon entering the process' signal handler
5) Read the cycle count read inside the kernel in using the sysfs

These tests give us three cycle counts for each iteration, from which we can compute three time intervals: the time it took from sending the interrupt to getting a return interrupt from the remote core, the time from entering the kernel locally to entering the user space signal handler and the full roundtrip time. Table I shows the averages of these measurements under the columns "Remote", "Kernel" and "Roundtrip".

As the table shows, the difference in hops between cores has a negligible impact on the latency. This is as expected when comparing the network's high speed network with the relatively slow core speed. However, it is surprising to see the difference between the remote and kernel columns. The former shows the time between raising an interrupt on the remote core, the remote core trapping to the kernel, running its interrupt handler, signalling the user space process and sending an interrupt back. This took around 3000 cycles on average. This is peculiar because the kernel time, the time between core 0 trapping into the kernel and the user space process receiving the signal, took around 4700 cycles and this time is also included in the remote time measurement.

Some additional tests show that this difference always appears between the initiating and replying core, the core who receives the reply and prints out measurements always takes more time to go from the kernel interrupt handler to the user space signal handler.

A possible explanation for this behaviour is that the remote kernel is idle aside from running the interrupt handler and the process' signal handler, whereas the local kernel has to access the filesystem and execute other system calls to retrieve the cycle counts stored by the kernel's interrupt handler. This is done outside of the reads from the cycle counter, so it should not impact the measurements, but it might still evict parts of the working set from the L1 and L2 caches causing a slow down by needing to go to main memory.

In [12] the Barrelfish developers measure a roundtrip latency of approximately 5000 cycles on bare metal, though this time includes reading and writing to MPB memory twice, making their measurements time higher than the raw IPI roundtrip time they had. This means the overhead of running in user space under Linux is around 2000 cycles more than bare metal. Sending and receiving interrupts is costly, but less so than we expected.

Using the above interrupt implementation we then implemented sending and receiving of S-NET records using interrupts to notify the receiver of a message. For our tests we used records consisting of 8 bytes of metadata and 25 tags (ints) and their 25 corresponding values (ints). This is more than the average S-NET record should have, but still well within range of realistic sizes. We then measured the roundtrip latency of sending these records between two cores to see what sort of latency we could achieve. For comparison we also implemented a busy-waiting version of the interrupt code, an MPI version and a RCCE version. Table II shows the results of these measurements (averaged over 1 million roundtrips).

For S-NET, as mentioned earlier, we are not that interested in low latencies and more in lightweight communication. Our interrupt based messaging implementation outperforms the MPI version by a good 20%, but has double the latency of the RCCE version and almost quadruple that of its busy-waiting equivalent.

The last column in Table II shows the average number of cycles that the interrupt based implementation was idle each roundtrip. This free time can be used for computational work and makes up 75% of the roundtrip time. The RCCE and polling implementations on the other hand waste their time in a busy-wait loop, preventing the core from doing useful computation. This problem can of course be diminished by polling every n micro-/miliseconds instead of busy-waiting, at the cost of increased the latency.

Hops	Polling	RCCE	MPI	Interrupt	Free
0	5.4	9.9	25.7	18.8	14.4
1	5.6	9.9	25.8	21.0	16.2
2	6.6	10.4	26.2	21.8	16.9
3	5.9	10.1	26.2	21.2	16.2
4	6.1	11.0	26.6	21.6	16.6
5	6.3	10.4	26.7	21.6	16.4
6	6.5	10.6	27.3	21.6	16.4
7	6.9	10.7	27.0	22.2	16.8
8	7.4	10.8	26.8	22.4	17.0

Table II
ROUNDTRIP MESSAGING LATENCY (IN 1000 CYCLES)

It would be interesting to compare the efficiency of various polling intervals for the polling and RCCE implementations and see whether a trade-off in latency can improve their efficiency to be on par with that of the interrupt implementation. Unfortunately we did not have time to finish these measurements and include them in this paper. The biggest problem in measuring this information is the granularity of the Linux scheduler. Most ways for a thread to yield its execution (so computational code can run) cause the next poll to be postponed until the next time the scheduler runs it. This causes blocks to last far to long for the polling to be effective. To solve this problem we will need to utilise Linux' real time scheduling support to make suspending the polling threat frequently for intermediate amounts of time feasible.

VI. CONCLUSION

In this paper we have introduced the declarative coordination language S-NET. It is designed to provide a more convenient way of programming for multi-core/many-core chip architectures by viewing programs as independent components with an input and output stream. These components are then used to construct programs using a set of combinators.

We sketched out our ideas for implementing the S-NET runtime system on top of the SCC. Focussing on the two most important short-term goals for S-NET: utilising the LUT capabilities of the SCC to eliminate needless copying of data by being able to quickly remap it and using the message passing buffers and on-chip network to implement a lightweight communication mechanism for S-NET's records.

In the near future we also want to investigate the ability to dynamically change the power and speed at which the cores run, allocating more resources to bottlenecks in the S-NET network and reducing the energy wasted by the parts which are idle.

After this presentation of our ideas we went on to discuss the various tests we implemented, their performance and how their results impacts our ideas for S-NET. We determined that the latencies for sending interrupts are not as small as we would

like, but are low enough what we need. Combined with the message passing buffers they provide us with enough to let us implement lightweight asynchronous message passing.

Unfortunately we were not able to finish implementing tests to compare the efficiency of asynchronous messaging to our polling and RCCE implementations (using various polling intervals), this means we don't have a conclusive "best" solution, but the numbers for our asynchronous implementation make us cautiously optimistic about how well it will stack up against the polling and RCCE implementations.

The biggest problem we encountered during our work was the lack of control over the L2 cache. Without control over the cache's behaviour it is difficult to accurately predict the performance (and in some cases correctness) of code. Making it difficult to accomplish some of the things we want to do, such as remapping memory using the LUTs. We can work around this, but at the cost of convenience and performance.

REFERENCES

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," pp. 108–109, feb. 2010.

[2] 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.

[3] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor (accepted for publication)," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011) – to appear, Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, (Istanbul, Turkey), July 2011. accepted for publication.

[4] Intel, "RCKMPI User Manual," February 2011.

[5] C. Grelck, A. S. (eds):, F. Penczek, C. Grelck, H. Cai, J. Julku, P. Hölzenspies, S. Scholz, and A. Shafarenko, "S-Net Language Report 2.0," Technical Report 499, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom, 2010.

[6] C. Grelck, S. Scholz, and A. Shafarenko, "Asynchronous Stream Processing with S-Net," *International Journal of Parallel Programming*, vol. 38, no. 1, pp. 38–67, 2010.

[7] C. Grelck, J. Julku, and F. Penczek, "Distributed S-Net: High-Level Message Passing without the Hassle," in *1st ACM SIGPLAN Workshop on Advances in Message Passing (AMP'10), Toronto, Canada, 2010* (G. Bronevetsky, C. Ding, S.-B. Scholz, and M. Strout, eds.), ACM Press, New York City, New York, USA, 2010.

[8] C. Grelck, "The essence of synchronisation in asynchronous data flow," in *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA*, IEEE Computer Society Press, 2011.

[9] C. Grelck, S.-B. Scholz, and A. Shafarenko, "Coordinating Data Parallel SAC Programs with S-Net," in *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, USA*, IEEE Computer Society Press, 2007.

[10] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. Shafarenko, R. Barrière, and E. Lenormand, "Parallel signal processing with S-Net," *Procedia Computer Science*, vol. 1, no. 1, pp. 2079 – 2088, 2010. ICCS 2010.

[11] F. Penczek, S. Herhut, S.-B. Scholz, A. Shafarenko, J. Yang, C.-Y. Chen, N. Bagherzadeh, and C. Grelck, "Message Driven Programming with S-Net: Methodology and Performance," *Parallel Processing Workshops, International Conference on*, vol. 0, pp. 405–412, 2010.

[12] S. Peter, T. Roscoe, and A. Baumann, "Barrelfish on the Intel Single-chip Cloud Computer," Tech. Rep. Barrelfish Technical Note 005, ETH Zurich, September 2010. http://www.barrelfish.org.