Reliable and Efficient Execution of Multiple Streaming Applications on Intel's SCC Processor

Lars Schor, Devendra Rai, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele

Computer Engineering and Networks Laboratory,
ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

Abstract. Intel's Single-chip Cloud Computer (SCC) is a prototype architecture for on-chip many-core systems. By incorporating 48 cores into a single die, it provides unique opportunities to gain insights into many-core software development. Earlier results have shown that programming efficient and reliable software for many-core processors is difficult due to a lack of appropriate programming tools. In this paper, we present a programming framework to execute multiple applications specified as Kahn process networks on the SCC. These applications might be started or stopped at runtime based on requests of the user. The proposed application programming interface (API) abstracts low-level implementation details from the application designer enabling high-level performance analysis and automated mapping optimization. To efficiently execute workload specified by the proposed API, a lightweight runtime-system and an automated program synthesis backend are presented. Extensive experiments are carried out to characterize the performance of the proposed framework.

Keywords: Many-Core Programming, Single-chip Cloud Computer, SCC, Runtime-System, Mapping, Distributed Application Layer, DAL.

1 Introduction

The demand for high computing power of novel real-time multimedia applications coupled with a single core's inability to support such high demands forces hardware designer to choose architectures with a high degree of parallelism. Intel's Single-chip Cloud Computer (SCC) [5] is a prototype of such architectures. By incorporating 24 tiles, each composed of two cores and local memory, into a single die, the SCC provides software developers early insights in developing software for highly parallel and distributed architectures.

The performance of such systems will critically depend on the efficient execution of applications on multiple cores. However, programming parallel applications for distributed systems, and Intel's SCC processer in particular, is difficult due to several reasons. The programmer has to handle the data transfer between cores using low-level message passing libraries and the distribution of the workload across the cores. For instance, unequally distributed workload results

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 790–800, 2014.

in low throughput or high latencies. Thus, traditional (low-level) methods to design and program parallel applications are not anymore appropriate to many-core systems that are architecturally more complex. Consequently, the goal is to provide an application programming interface (API) and design-flow that abstracts low-level implementation details from the application designer enabling high-level performance analysis and automated mapping optimization.

Programming in a high-level language has typically various advantages including being less error-prone and having various design optimization steps automated. A promising paradigm to program distributed systems is to use process networks as model of computation (MoC). Process networks have the advantage that they explicitly express the parallelism of an application and separate computation from communication. Such separation of concerns enables the parallelization of computation and communication allowing process networks to fully exploit the available resources. At the same time, using a process network as MoC allows the designer to rapidly evaluate the performance of different design candidates and hardware/software partitioning options. The distributed application layer (DAL) [11], for instance, uses the Kahn process network (KPN) [6] MoC to specify applications. Due to the well-defined semantics of the KPN model, data races, non-determinism, or the need for strict synchronization are avoided. In addition, the DAL MoC uses a finite state machine (FSM) to represent interactions between applications. Each state represents an execution scenario, i.e., a certain set of applications running in parallel. At runtime, the user can switch between the execution scenarios by starting or stopping applications.

Analyzing the performance, providing quality-of-service guarantees, and optimizing the process-to-core mapping of workload specified by the DAL MoC has been described in [7, 11]. This paper reports our work to execute workload specified by the DAL MoC onto the SCC. This includes the definition of a high-level API for programming parallel applications specified as KPNs that abstracts low-level architectural details from the application designer enabling high-level performance analysis, process clustering, or mapping optimization. The runtime-system comes with the capability to start and stop processes, and to create and destroy FIFO channels at runtime leading to low memory footprint. Great emphasis is placed on minimizing the communication overhead by using a highly optimized inter-process communication protocol. In particular, the RCKMPI library [2] is used as middleware layer for communication between cores. Finally, extensive experiments are carried out on Intel's SCC processor to characterize the performance of the proposed framework and runtime-system.

The Eclipse SDK has been extended with an editor for the DAL MoC, a high-level mapping optimization framework, and the program synthesis backend proposed in this paper. The plugin is available for download under `http://www.tik.ee.ethz.ch/~euretile/dalipse`.

The remainder of the paper is structured as follows: The SCC is revised in Section 2. In Section 3, the DAL MoC is presented. In Sections 4 and 5, the runtime-system and program synthesis backend are described. Section 6 presents experimental results and related work is reviewed in Section 7.

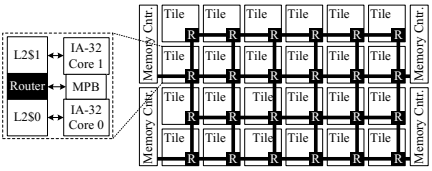


Fig. 1. Schematic outline of Intel's SCC processor

2 Single-Chip Cloud Computer

The SCC processor [5], schematically outlined in Fig. 1, is a 48-core experimental processor from Intel. It consists of 24 tiles that are organized into a 4×6 grid and linked by a 2D mesh on-chip network. A tile contains a pair of P54C cores, a router, and a 16 KB block of SRAM. Each core has an independent L1 and L2 cache. The on-tile SRAM block is also called "message passing buffer" (MPB) as it enables the exchange of information between cores.

The most commonly used software platform for Intel's SCC is to run a Linux kernel on each core. For communication purpose, various message passing libraries have been developed including RCCE [14], iRCCE [1], and RCKMPI [2]. Instead of implementing our own software platform, we run our runtime-system on top of the Linux kernel. This enables us to use the basic multi-threading mechanisms provided by the Linux kernel.

3 DAL Model of Computation

The dynamic behavior of the workload is captured by a set of execution scenarios that form a finite state machine (FSM). Each state represents a set of concurrently running applications and each state transition corresponds to an application start or stop request. An example of a FSM is outlined in Fig. 2.

Applications are specified as KPNs [6]. More precisely, an application consists of autonomous processes that can only communicate through unbounded point-to-point FIFO channels, see Fig. 3 for an example. However, as channels with unbounded capacity cannot be realized in real implementations, we use a KPN semantics-preserving implementation with finite buffers that are accessed using blocking read and write functions [4]. Blocking means that a process stalls if it attempts to read data from an empty channel or write to a full channel.

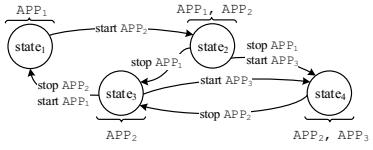


Fig. 2. Example of a FSM specifying the dynamic behavior of the workload

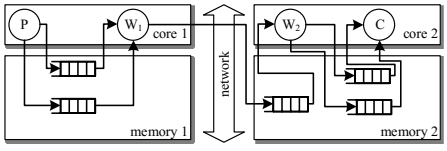


Fig. 3. Application specified by the KPN MoC together with its mapping

Listing 1. Specification of a KPN with two processes

```
<processnetwork>
  <process name="prod">
    <port type="output" name="out"/>
    <src type="c" location="prod.c"/>
  </process>
  <process name="cons">
    <port type="input" name="in"/>
    <src type="c" location="cons.c"/>
  </process>
  <channel cap="8" name="channel">
    <send process="prod" port="out"/>
    <rec process="cons" port="in"/>
  </channel>
</processnetwork>
```

Listing 2. Implementation of a KPN process using the proposed API

```
procedure INIT(ProcessData *p)
  initialize();
end procedure

procedure FIRE(ProcessData *p)
  fifo->READ(buf, size);
  manipulate();
  fifo->WRITE(buf, size);
end procedure

procedure FINISH(ProcessData *p)
  cleanup();
end procedure
```

The proposed API for KPNs is outlined in Listings 1 and 2. The topology, i.e., the connections between processes by FIFOs, is specified in an XML format. The functionality of the individual processes is specified in C/C++ and is composed of three procedures. The INIT procedure is executed once at startup of the application. Afterwards, the execution of a process is split into individual executions of the FIRE procedure, which is repeatedly invoked by the system scheduler. Finally, the FINISH procedure is called before an application is stopped. Each process can read from its input and write to its output channels by calling the high-level READ and WRITE procedures.

4 Runtime-System

Executing a workload specified according to the DAL MoC on Intel's SCC processor requires a runtime-system and a program synthesis backend. The task of the runtime-system is thereby to provide an implementation of the API, i.e., services for inter-process communication, a mechanism to iteratively execute processes by calling their FIRE procedure, and services to manage processes and channels at runtime.

Inter-Process Communication. To be efficient, the runtime-system has to differ between communication over shared and distributed memory, see Fig. 4. While ring buffers in private memory are used for intra-core communication, an advanced architecture-dependent FIFO implementation is required for efficient inter-core communication.

In general, the FIFO channel might be implemented in private memory of the sender or receiver, or in shared memory. We implemented the FIFO channels in the private memory of the receiver and used the RCKMPI library [2] for inter-core communication. The RCKMPI library automatically takes the memory organization of the SCC into account and uses the MPB, if appropriate. As no DMA controller is available on the SCC for inter-core communication, we launch a LISTENER thread on each core. The LISTENER thread is responsible for

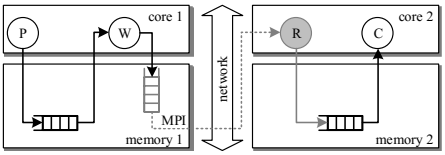


Fig. 4. Example of intra- and inter-core communication. Virtual FIFO and LISTENER thread are illustrated in grey.

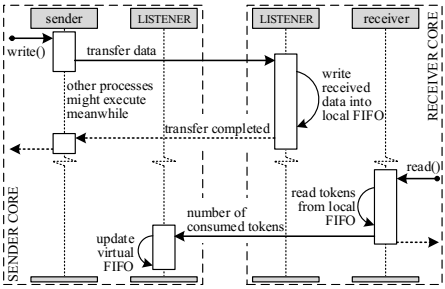


Fig. 5. Inter-process communication protocol between processes located on different cores

handling all incoming traffic and writing the data to the correct FIFO channel. To keep the LISTENER thread lightweight, it uses the memory of the local FIFO channel as receive buffer for the data transfer, thereby avoiding expensive allocation and copy operations.

To avoid deadlocks, the LISTENER thread must not be blocked at any time, i.e., we have to ensure that a data transfer is only initialized if the receiver has enough available space to store the data. This is ensured by a virtual FIFO at the sender. The virtual FIFO has the same metadata (amount of free space) as the actual FIFO, but if a process attempts to write, the data is either directly transferred or the calling process is blocked as long as the receiver has not enough space to store the data. The disadvantage of this approach is that the virtual FIFO has to know when the receiver process has consumed data, which is ensured by a signal. The inter-process communication protocol is sketched in Fig. 5.

Multi-processing. As our runtime-system runs on top of a Linux kernel, we use the multi-processing features provided by the operating system (OS) to run multiple processes in a quasi-parallel fashion on a single core. In particular, processes are mapped onto POSIX threads and scheduled by the OS' scheduler. When a process is blocked due to empty input or full output channels, the scheduler automatically selects a different process to execute.

Process and Channel Management. The FSM enables the programmer to start and stop applications at runtime. We will show in the next section that this dynamism is handled by a runtime-manager in the form of an additional process network. The runtime-system has therefore to provide services to install, uninstall, start, and stop processes, and to create and destroy FIFO channels.

The memory footprint of the system is reduced by storing the individual processes as dynamic libraries that are loaded when the application is started. Thus, installing a process involves loading and dynamically linking the corresponding library, and then executing its INIT procedure. Similarly, uninstalling a process involves executing the FINISH procedure and unloading the dynamic library. The procedure to create a channel depends on the mapping of the sender and receiver. If both processes are mapped onto the same core, a local FIFO channel

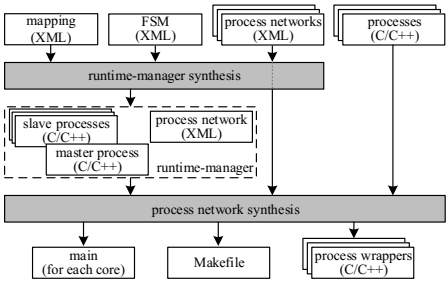
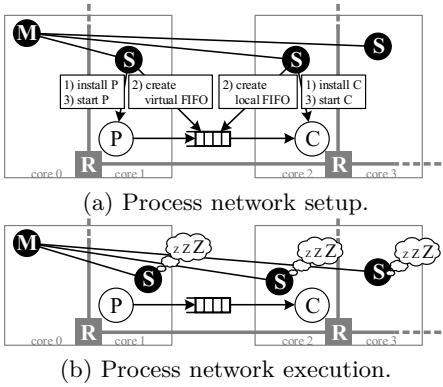


Fig. 6. Program synthesis flow generating the source code of the runtime-manager and the process wrappers



(a) Process network setup.

(b) Process network execution.

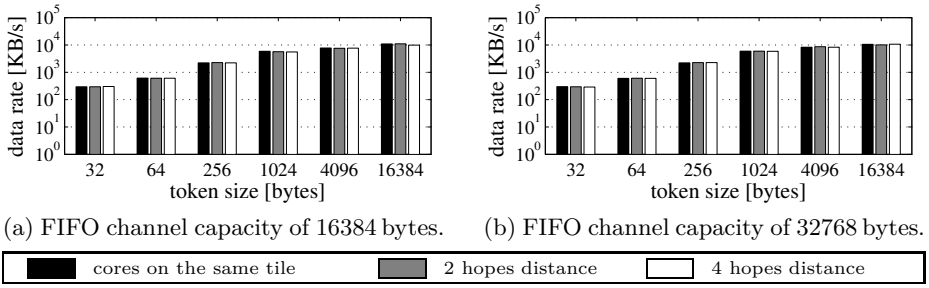
Fig. 7. Structure of the proposed runtime-manager. "M" and "S" represent master and slave processes, respective.

is instantiated. Otherwise, a virtual FIFO channel is instantiated at the sender and a local FIFO channel at the receiver. Afterwards, virtual and local FIFO channels are registered at the corresponding LISTENER thread. A FIFO channel is destroyed by deregistering it at the LISTENER thread and freeing the memory buffer. Finally, a process is started by registering the thread at the scheduler, and stopped by aborting the FIRE procedure and deregistering it from the scheduler.

5 Program Synthesis Backend

A program synthesis backend is the second component required to execute a workload specified by the DAL MoC. It basically consists of creating the runtime-manager, embedding each process into a POSIX thread, and creating a MAIN function for each core, see Fig. 6.

Runtime-Manager Synthesis. The task of the runtime-manager synthesis is to automatically construct a runtime-manager that satisfies the given system specification. The result of this step is a process network with one master process and one slave process per core that uses the additional services provided by the runtime-system to manage the processes and channels, see Fig. 7. The runtime-manager monitors the system, starts and stops individual process networks depending on the requests of the user. The master process thereby manages the dynamic execution of the system and the slave processes are responsible for the management of the processes and FIFO channels. Thus, the master process distributes the actual computation to the slave processes. Once the slave processes performed their work, they go to sleep until the master process sends them a new job making the proposed runtime-manager very lightweight in the sense that it does not affect the execution of the process network.



(a) FIFO channel capacity of 16384 bytes. (b) FIFO channel capacity of 32768 bytes.

Fig. 8. Data transfer rate between two cores for three different hop distances

Process Network Synthesis. Finally, the process network synthesis step embeds each process into a POSIX thread and creates a MAIN function for each core. Embedding each process into a POSIX thread is achieved by a process wrapper that repeatedly calls the FIRE procedure of the process. The process definition is then stored together with the process wrapper as a dynamic library in the file system so that it can be loaded on request of a slave process. The MAIN function has two tasks. First, it starts the LISTENER thread. Then, it initializes the processes and channels of the runtime-manager and turns the control of the system over to the master.

6 Experimental Results

In this section, we provide experimental results characterizing the framework.

Experimental Setup. An Intel SCC processor running at 533 MHz for the cores and 800 MHz for the routers and DDR3 RAM has been used for the experiments. A Linux image with kernel 2.6.32 has been loaded on each core. RCKMPI has been configured to use the default channel, i.e., the SCCMPB channel. The ICC-8.1 compiler with optimization level -O2 is used for all experiments.

Data Transfer Rate. A synthetic application consisting of two processes and one FIFO channel has been designed to measure the data transfer rate between two cores. The application executes 100000 iterations and in one iteration, the source process writes one token to the FIFO channel and the sink process reads the token from the FIFO channel. No other processes except the runtime-manager are running on the SCC. Figure 8 shows the data transfer rate between two cores whereby the token size is varied between 32 bytes and 16384 bytes. The experiment has been repeated for two capacities of the FIFO channel and three hop distances between the cores. The observed peak data rate is 11 Mbytes/s. While hop distance and capacity have small influence on the data transfer rate, the rate significantly increases with the size of a single token.

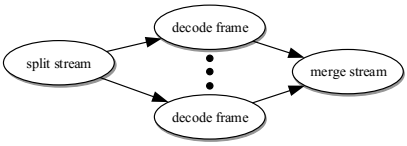


Fig. 9. KPN of the distributed MJPEG decoder

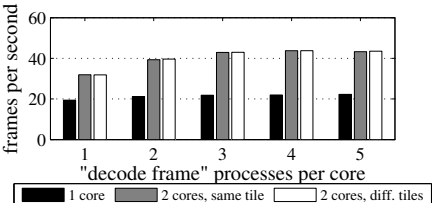


Fig. 10. Decoded frames per second using the MJPEG decoder

Runtime-System Overhead. As the runtime-manager sleeps after performed work, the overhead of the runtime-system can mainly be assigned to the LISTENER thread. To measure this overhead, we use a sequential implementation of a motion JPEG (MJPEG) decoder. Decoding 5000 frames takes thereby about 162.0 s if the MJPEG decoder is executed in parallel to the LISTENER thread and 158.9 s if it is executed as an individual application. Thus, the measured overhead is less than two percent.

Context Switching Overhead. To characterize the effect of multi-processing on a single core, we consider a distributed implementation of the MJPEG decoder, see Fig. 9 for the process network. The network has multiple "decode frame" processes decoding a complete frame in a single iteration. In Fig. 10, the decoded frames per second using the MJPEG algorithm are compared for implementations mapping a different number of "decode frame" processes onto one core. Furthermore, the graph differs between three configurations. First, only one core, then both cores of a tile, and finally two cores of different tiles are used to execute the "decode frame" process. It shows that the frame rate increases significantly if two processes are mapped onto a single core as communication and computation can partially overlap. Furthermore, the frame rate does not decrease even if five processes are mapped onto each core indicating a low multi-processing overhead.

Speed-Up Due to Parallelism. Finally, we evaluate the speed-up due to available number of cores for four different applications. Besides the MJPEG decoder, a MPEG-2 decoder, a ray-tracing, and a quicksort algorithm are studied. The ray-tracing algorithm generates an image of 100×100 pixels and can concurrently analyze multiple rays. We map either one or two such processes onto one core. The quicksort algorithm sorts an array with 5000 elements and can have multiple instances of a "sort" process to concurrently sort multiple sub-arrays. Two configurations with either four or eight "sort" processes are considered. The MPEG-2 decoder concurrently decodes multiple macroblocks. We again map either one or two such processes onto one core

In Fig. 11, the speed-up is compared for implementations running on a different number of cores. The speed-up is calculated with respect to an implementation running on a single core. The maximum speed-up that can be achieved is 20.7 for the MJPEG decoder application. As MJPEG is an intraframe-only

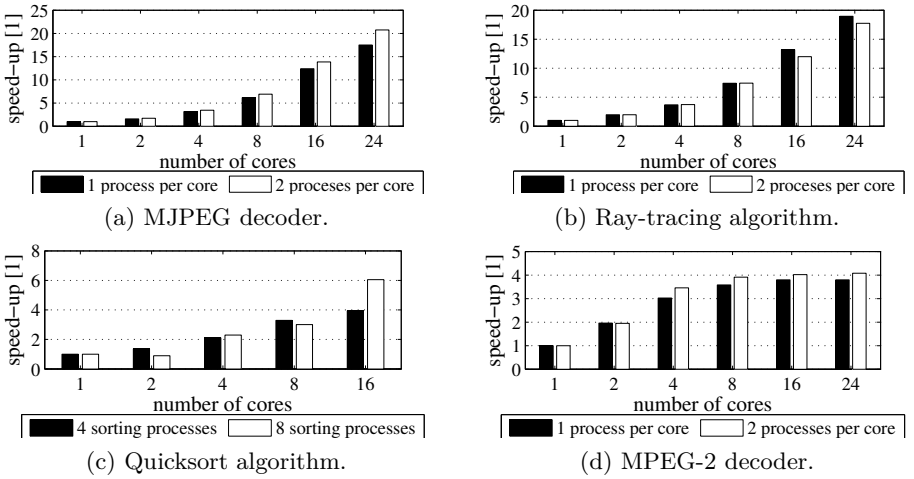


Fig. 11. Speed-ups of four benchmarks for a varying number of cores

compression scheme, the frames can be decoded in parallel on different cores. The ray-tracing algorithm achieves a speed-up of almost 20 on 24 cores. As each ray can individually be analyzed, the ray-tracing algorithm is well-suited for parallelization. The speed-ups achieved with the quicksort algorithm are much smaller than with the previous applications. This might be because additional time is required to partition the input array into sub-arrays and then to collect the intermediate results. A maximum speed-up of 6.0 on 16 cores is achieved when eight "sort" processes are running in parallel. Finally, due to data-dependencies between the frames, the MPEG-2 application only achieves a speed-up of about 4.1. The speed-up increases linearly for a small number of cores before collecting and distributing frames become the bottleneck for higher parallelization.

7 Related Work

To abstract low-level implementation details from the application designer, various high-level programming environments have been developed for Intel's SCC processor. For instance, a distributed Java virtual machine for Intel's SCC processor is proposed in [10]. In [9], Barrelfish OS is presented that provides the user a single OS instance to manage the complete set of available computing cores. The closest related work is presented in [13] by mapping distributed S-Net streaming networks onto Intel's SCC processor. Similar to our work, functions are written in a standard programming language and then mapped onto stream-processing components (so-called boxes). In contrast, our framework permits stateful processes and interactions between applications. Furthermore, the formal design approach enables high-level performance analysis and mapping optimization at design-time so that quality-of-service constraints can be provided.

The KPN [6] model of computation has been the basis for many frameworks to design multi/many-core systems including Daedalus [8], DOL [12], and SHIM [3]. Most of these frameworks support a wide-variety of target platforms and have recently been extended to dynamic workload. Our approach, in contrast, uses a formal specification of the dynamic behavior in the form of a FSM, which enables an efficient analysis and execution of multiple dynamic KPN applications.

8 Conclusion

In this paper, we have presented a high-level programming framework that allows to execute multiple applications specified as Kahn process networks on Intel's SCC processor. To abstract low-level implementation details from the application designer, a high-level API has been proposed. The API does not only specify the individual applications but also their interactions enabling dynamic behavior in the sense that applications can start and stop at runtime. To efficiently execute workload specified by the proposed API on the SCC, we presented a lightweight runtime-system and an automated program synthesis backend. In particular, efficiency is obtained by a distributed runtime-manager that loads processes and instantiates channels only when the application is actually started. The paper has shown that the proposed program synthesis backend can be extended to integrate high-level performance analysis and process-level mapping optimization enabling quality of service guarantees. Finally, we applied the approach to various streaming applications demonstrating the advantages of programming applications with the proposed framework.

Acknowledgments. This work was supported by EU FP7 project EURETILE. Lars Schor was also partially supported by an Intel PhD Fellowship.

References

1. Clauss, C., et al.: Evaluation and Improvements of Programming Models for the Intel SCC Many-Core Processor. In: Proc. HPCS, pp. 525–532 (2011)
2. Comprés Ureña, I.A., Riepen, M., Konow, M.: RCKMPI – Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 208–217. Springer, Heidelberg (2011)
3. Edwards, S.A., Tardieu, O.: SHIM: A Determinstic Model for Heterogeneous Embedded Systems. IEEE Trans. VLSI Syst. 14(8), 854–867 (2006)
4. Geilen, M., Basten, T.: Requirements on the Execution of Kahn Process Networks. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 319–334. Springer, Heidelberg (2003)
5. Howard, J., et al.: A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In: Proc. ISSCC, pp. 108–109 (2010)
6. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In: Proc. of the IFIP Congress, vol. 74, pp. 471–475 (1974)

7. Kang, S.H., et al.: Multi-Objective Mapping Optimization via Problem Decomposition for Many-Core Systems. In: Proc. ESTIMedia, pp. 28–37 (2012)
8. Nikolov, H., et al.: Systematic and Automated Multiprocessor System Design, Programming, and Implementation. IEEE T. Comput. Aid. D. 27(3), 542–555 (2008)
9. Peter, S., et al.: Early Experience with the Barrelfish OS and the Single-chip Cloud Computer. In: Proc. MARC, pp. 35–39 (2011)
10. Saballus, B., et al.: A Scalable and Robust Runtime Environment for SCC Clusters. In: Proc. MARC, pp. 71–74 (2011)
11. Schor, L., et al.: Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems. In: Proc. CASES, pp. 71–80 (2012)
12. Thiele, L., et al.: Mapping Applications to Tiled Multiprocessor Embedded Systems. In: Proc. ACSD, pp. 29–40 (2007)
13. Verstraaten, M., et al.: On Mapping Distributed S-Net to the 48-core Intel SCC Processor. In: Proc. MARC, pp. 41–46 (2011)
14. van der Wijngaart, R.F., et al.: Light-weight Communications on Intel's Single-chip Cloud Computer Processor. SIGOPS Oper. Syst. Rev. 45(1), 73–83 (2011)