Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs

Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, Lothar Thiele
Computer Engineering and Networks Laboratory
ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

***Abstract*—As single-processor systems are ceasing to scale effectively, multi-processor systems are becoming more and more popular. While there are many challenges of designing multi-processor systems in hardware, writing efficient parallel applications that utilize the computing capability of multiple processors may reveal to be even more challenging. In this paper, we introduce a framework that allows to efficiently execute applications expressed as Kahn process networks on multi-processor systems using protothreads and windowed FIFOs. We show that application developers can use this framework to achieve considerable speed-ups on the Cell Broadband Engine without needing to write architecture-specific code.**

I. Introduction

Currently, we are witnessing the evolution from single-processor to multi-processor computing. Due to technology restrictions, single-processor systems have ceased to scale effectively, such that the growing demand for computing power can only be satisfied by multi-processor systems. Consequently, the performance of future systems will critically depend on the efficient execution of applications on multiple processors.

A promising paradigm for achieving efficient execution is to use the Kahn process network (KPN) [1] model of computation for programming multi-processor systems. The KPN model allows to explicitly express the parallelism in the application, while at the same time separating computation from communication. This allows to parallelize the computation and the communication in an application and thus exploit the available parallel resources in a multi-processor system. In addition, due to the well-defined semantics of the KPN model, many pitfalls of parallel execution, such as data races, nondeterminism, or the need for strict synchronization can be avoided. Specifically, by using a KPN-based design flow, such as the one proposed in this paper, application developers only need to develop a high-level specification of the application while the architecture-specific implementation is automatically synthesized, leading to implementations that are correct-by-construction.

Basically, a KPN describes an application as a network of autonomous processes that communicate through FIFO channels, as shown in Fig. 1 where a KPN application is implemented on a dual-core platform. Especially streaming applications, such as audio and video codecs, (array) signal processing applications, or networking applications can be naturally expressed as KPN applications because their algorithmic structure matches well with the KPN model.

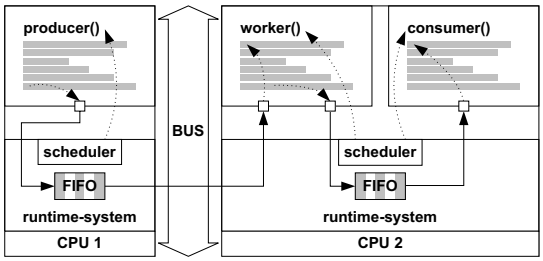

Fig. 1. High-level model of a KPN application consisting of three processes executing on a multi-processor system.

The biggest drawback of the KPN model is that the communication between processes and their orchestration can considerably limit the speed-up achievable by parallel execution. The challenge is thus to keep this overhead low when implementing a KPN application. The second challenge is related to the parallel specification of an application as a KPN where different levels of granularity can be employed. Therefore, an ideal situation would be to hide the communication and orchestration overhead when mapping processes on the same processor and to execute a fine-grained process network as if processes would be merged into a coarse-grained implementation with less overhead.

On a first view, a runtime-system for executing multiple processes on a single processor seems to come at considerable cost in terms of development effort, runtime overhead, and code size. This disregards, however, that providing quasi-parallelism to execute processes in a KPN is much easier than providing quasi-parallelism in a fully preemptive event- or time-triggered system. In particular, we propose to use so-called protothreads [2] to implement the quasi-parallelism required for executing a KPN application. Protothreads are usually used for programming small embedded systems, such as wireless sensor nodes. Interestingly, the capabilities of single processors in a multi-processor system are similarly constrained concerning memory and performance, making protothreads a good match for implementing the required quasi-parallelism. Furthermore, we propose to use windowed

978-1-4244-5170-8/09/$26.00 © 2009 IEEE

FIFOs instead of standard FIFOs for efficient communication. Contrary to standard FIFOs where data need to be explicitly copied into the FIFO buffer, windowed FIFOs allow processes to directly access the FIFO buffer, thereby avoiding memory copy operations.

Consequently, we propose an architecture-independent application programming interface (API) for programming parallel applications as KPNs using protothreads and windowed FIFOs. We implemented the corresponding runtime-system and code synthesis backend for the distributed operation layer (DOL) [3] framework that allows KPN applications to efficiently execute on both single- and multi-processor systems. Extensive experiments are carried out to support our claims and show the effectiveness of the proposed approach. In particular, we target a distributed memory architecture, namely the Sony/Toshiba/IBM Cell Broadband Engine [4], where we achieve speed-ups close to seven when executing KPN applications using the PowerPC and six synergistic processing elements. The framework is available for download under http://www.tik.ee.ethz.ch/∼shapes/.

The remainder of the paper is structured as follows: In the next section, related work is reviewed. In Section III, the considered problems and the approach to solve them are described. In Sections IV and V, the details of our approach are presented. In Section VI, an implementation of the approach targeting the Cell Broadband Engine is described, and Section VII presents experimental results. Finally, Section VIII concludes the paper.

II. Related Work

The Kahn process network model of computation and its ramifications are widely studied models for dataflow-oriented applications [5]. Several frameworks for designing multi-processor systems are based on the KPN model, such as C-HEAP [6], CIC [7], Daedalus [8], [9], Koski [10], MAMPS [11], OpenDF [12], SHIM [13].

In these frameworks, mainly the stages preceding software synthesis have received the most attention, namely KPN generation from programs written in a subset of Matlab or C, design space exploration, and analytic and simulation-based performance analysis. These steps are typically followed by software synthesis for multi-processor systems or architecture synthesis for FPGA platforms. The work in this paper considers KPN software synthesis. In terms of software synthesis, C-HEAP, CIC and Daedalus generate a functional simulation of a KPN based on the Linux pthreads library. For SHIM, a back-end for Intel multi-core processors has been developed that is based on the Linux pthreads library, as well [14]. Similarly, threads of the eCos operating system [15] are used to execute processes in Koski. Unfortunately, as shown in Section VII, kernel-space threads incur a considerable overhead due to context switching that can severely limit the achievable speed-up. An alternative approach for executing a KPN in a quasi-parallel fashion is to use user-space threads. An implementation of this approach is the open-source library YAPI [16] developed at Philips Research.

In contrast to the mentioned approaches, we use so-called protothreads [2], enabling the (preemptive) cooperative execution of processes in a single CPU context using a single stack. Thereby, the context switching overhead is very low and no multi-threading support by an operating system is required to execute multiple processes on a single processor. Furthermore, we propose to use windowed FIFO channels [17] for communication instead of standard FIFO channels, thereby considerably reducing the communication overhead because the costly copying of data can be avoided, as has already been observed [6], [18]. Following this approach, KPN applications can be considerably faster executed compared to using kernel-space or user-space threads and standard FIFOs.

The goals achieved by the proposed approach can also be achieved by restricting the model of computation. The StreamIt framework [19], [20], for instance, is based on the synchronous dataflow [21] model of computation. Compared to KPN, synchronous dataflow is restricted to constant token production and consumption rates for each channel. This allows to statically schedule an application and to fuse adjacent processes to coarsen the granularity of processes and reduce the context switching overhead, for instance. Similar observations apply to the Sequoia framework [22] which is targeted at applications where computations are mainly structured into loops that operate on arrays using highly regular access patterns. We show that our framework based on protothreads and windowed FIFOs is similarly effective, while still being able to leverage the expressiveness of the KPN model.

Finally, another alternative is to program multi-processor systems without restricting to a certain model of computation. Several high-level APIs to facilitate the programming of multi-processor systems have been proposed, such as OpenMP [23], MPI [24], Multiflex [25], and TTL [18]. Like in our approach, these APIs allow to write architecture-independent code that is automatically refined to a given architecture and runtime-system. Due to the lack of an underlying model of computation, however, design activities such as automatic design space exploration or analytic performance analysis are difficult to automate when using these APIs.

For performing experiments, we have developed a runtime-system based on protothreads and windowed FIFOs for the Sony/Toshiba/IBM Cell Broadband Engine (CBE) [4]. Runtime-systems for KPN applications executing on the CBE have been developed for CIC [26], Daedalus [27], SHIM [28], and StreamIt [20], as well as CBE-specific software synthesis tools and libraries [29], [30]. All of these contributions have in common that multiprocessing on the individual synergistic processing elements (SPEs) of the CBE is very limited. Either only a single process is executed on each SPE or, in case of multiple processes, the scheduling is static and processes are non-preemptive. The exchange of the complete SPE context is a further possibility to run multiple processes on a single SPE which is very costly in terms of context switching time and only effective for tasks with a high granularity, however. Contrary, by exploiting a runtime-system based on

protothreads, we are able to execute multiple processes in a quasi-parallel fashion on the SPEs. On the one hand, it is thus possible to overlap communication and computation by switching between processes when another process is waiting for data. On the other hand, no manual merging or static scheduling of processes is required which allows to completely automate software synthesis. Finally, using protothreads also allows to execute multiple applications (a set of disjunct KPNs) in parallel on the CBE.

III. Problem and Approach

In this paper, the efficient execution of KPNs on multi-processor systems is considered where "efficient" refers to speed-up. Efficiency in terms of latency, power, or real-time behavior is not considered in this paper. The goal is to maximize the speed-up, that is, to reach a speed-up that equals the number of processors in a multi-processor system.

Clearly, this goal could also be achieved by manually optimizing multi-processor applications for a specific architecture which is a time-consuming effort, however. The advantage of the KPN model is that the application-independent parts, namely the implementation of FIFO channel communication and the synchronization of processes, can be implemented once and reused in any application. The application programmer is then relieved from dealing with these low-level details and can focus on the application-specific parts. The goal is thus to specify applications using a high-level API to achieve high speed-ups without sacrificing huge amounts of development time.

To achieve these goals, we separate concerns, as follows:

- First, a high-level API is defined which allows applications to be written in an architecture-independent way.
- Second, the high-level API is implemented in an architecture-dependent runtime-system.

The key to achieve an efficient implementation is the runtime-system. Regarding this implementation, the following properties of the KPN model can be leveraged:

Untimed model. The KPN model is an untimed model of computation, which means that a KPN application can be executed in a completely data-driven manner. Each processor in the system can thus execute autonomously as long as data are available in the input channels and buffer space is available in the output channels. The runtime-system running on different processors can thus be implemented independently from each other, as indicated in Fig. 1.

Cooperative processes. The processes of a KPN are designed to cooperate. Therefore, a runtime-system for running multiple processes of a KPN in a quasi-parallel fashion on a single processor does not require all the mechanisms usually found in a fully-fledged multi-tasking operating system, such as full preemption, memory protection, or different scheduling policies. Rather, preemption is only needed when processes are blocked due to empty input channels or full output channels, and it is even desirable to share memory between processes to reduce the communication overhead. We leverage these properties by using protothreads [2] for executing multiple processes on a single processor and by using windowed FIFOs [17] to share memory between processes.

Separation of computation from communication. While the previous aspects are the basis for the efficient execution of (a part of) a KPN on a single processor, the key for multi-processor performance is to overlap communication between processors with computations performed on these processors. In a KPN, this is greatly facilitated due to the explicit separation of computation from communication. In an implementation, direct memory access (DMA) engines can be used for parallelizing computation and communication in most multi-processor systems.

Considering the properties and constraints described above, the application specification using a high-level API and the implementation of the runtime-system are described in the following sections.

IV. Application Specification

Conceptually, a KPN consists of autonomous processes that can only communicate through unbounded point-to-point FIFO channels [1]. In the KPN model, a process is a monotonic and determinate mapping F from one (or more) input streams to one (or more) output streams. Monotonicity and determinism imply that an output stream can be constructed by iteratively applying the mapping F to subsequent parts of the corresponding input stream(s) and concatenating the results: $\mathbf{Y} = F(\mathbf{X}) : [y_1, y_2, \ldots] = [F(x_1), F(x_2), \ldots]$. Obviously, channels with unbounded capacity cannot be realized in a physical implementation, however. Still, a KPN semantics-preserving implementation can use finite buffers that are accessed using blocking read and write functions [31]–[33]. Blocking means that a process stalls if it attempts to read data from an empty channel or if it attempts to write to a full channel.

From a performance point of view, the communication over FIFO channels can easily become a bottleneck in the implementation of a KPN. This is because in a standard FIFO implementation data need to be explicitly copied into the FIFO buffer by the sender and explicitly copied to a local memory of the receiver. This copying of data wastes CPU cycles which can be avoided by using a windowed FIFO. A windowed FIFO permits direct, random access to a window of the channel buffer. While using windowed FIFO channels preserves the KPN semantics, as has been formally proven in [17], it avoids the costly copying of data because of the possibility to directly access the channel buffer. In [18], for instance, the execution time of an MP3 decoder could be reduced by approximately 30% by using windowed FIFOs instead of standard FIFOs.

Based on these considerations, we propose the high-level API illustrated in Listing 1 for specifying KPN applications. After a one-time initialization in the INIT function, the execution of a process is split up into individual executions of the FIRE function which needs to be repeatedly invoked by a scheduler. Communication is enabled by accessing windowed FIFOs using CAPTURE and CONSUME for reading and RESERVE and RELEASE for writing. The functions

CAPTURE and RESERVE for acquiring a read-window and a write-window, respectively, are blocking and return pointers to a contiguous piece of memory in the FIFO buffer and the actual size of the window. This way, in case there is (temporarily) less than the specified amount of bytes available in the windowed FIFO buffer, CAPTURE and RESERVE might reserve a window with a smaller size than the requested one.

Listing 1. Implementation of a KPN process accessing a windowed FIFO channel.

```
01 procedure INIT(ProcessData *p) //process initialization
02   initialize();
03 end procedure
04
05 procedure FIRE(ProcessData *p) //process execution
06   wfifo_in->CAPTURE(rbuf, size);  //acquire read-window
07   wfifo_out->RESERVE(wbuf, size); //acquire write-window
08   manipulate();
09   wfifo_in->CONSUME();  //release read-window
10   wfifo_out->RELEASE(); //release write-window
11 end procedure
```

In addition to the functionality of individual processes which is specified in C/C++ as shown in Listing 1, the connections between processes by windowed FIFOs needs to be described. For this purpose, we propose an XML format to specify the topology of a KPN (see Listing 2).

Listing 2. XML specification of the KPN shown in Fig 1.

```
01 <processnetwork>
02   <process name="producer">
03     <port type="output" name="out"/>
04     <source type="c" location="generator.c"/>
05   </process>
06   <process name="consumer">
07     <port type="input" name="in"/>
08     <source type="c" location="consumer.c"/>
09   </process>
10   <process name="worker">
11     <port type="input" name="in"/>
12     <port type="output" name="out"/>
13     <source type="c" location="worker.c"/>
14   </process>
15
16   <channel type="wfifo" capacity="8" name="channelA">
17     <sender process="producer" port="out"/>
18     <receiver process="worker" port="in"/>
19   </channel>
20   <channel type="wfifo" capacity="8" name="channelB">
21     <sender process="worker" port="out"/>
22     <receiver process="consumer" port="in"/>
23   </channel>
24 </processnetwork>
```

For code synthesis for multi-processor systems, a mapping of the processes to the available processors is required, as well. For this purpose, we employ the XML format exemplified in Listing 3.

Listing 3. XML specification of the mapping shown in Fig 1.

```
01 <mapping>
02   <binding name="binding_producer">
03     <process name="producer"/> <processor name="CPU 1"/>
04   </binding>
05   <binding name="binding_worker">
06     <process name="worker"/>   <processor name="CPU 2"/>
07   </binding>
08   <binding name="binding_consumer">
09     <process name="consumer"/> <processor name="CPU 2"/>
10   </binding>
11 </mapping>
```

V. Runtime-System and Software Synthesis

To execute an application specified according to Section IV, two components are required, namely a runtime-system that provides an implementation of the API and a software synthesis tool that implements the high-level application specification on top of the runtime-system. In this section, both components are described in general, while specific implementation details follow in the next section.

A. Runtime-System

The task of the runtime-system is to implement the high-level API routines for communication and to iteratively invoke the FIRE function of processes. To execute a KPN, a runtime-system has to provide three main services that are described in the following:

- multi-processing on single processor,
- windowed FIFO communication on single memory, and
- windowed FIFO communication between memories.

Multi-processing on single processor. When running multiple processes on a single processor, it is necessary to be able to switch between processes when blocking occurs. The challenge is to keep the runtime-overhead associated to multi-processing low. In the following, the term *thread* is used to denote the mechanism that provides the multi-processing capability on a single processor. In Table I, three basic mechanisms to implement threads are compared, namely kernel-space threads, user-space threads, and stack-less user-space threads (protothreads).

TABLE I
Comparison of different thread implementations.

	kernel-space	**user-space**	**stack-less**
thread switching mechanism	CPU context switch, restoration of thread table	restoration of thread table	stack unwinding
scheduling	time- or event-driven	event-driven	event-driven
preemption	preemption at any point by scheduler	voluntary preemption by threads	voluntary preemption by threads
stack	one per thread	one per thread	shared among all threads
example library	POSIX threads (pthreads)	YAPI[1] [16], SystemC[2]	protothreads[3] [2]

[1] http://y-api.sourceforge.net
[2] http://www.systemc.org
[3] http://www.sics.se/∼adam/pt/

Due to the reasons mentioned in Section III, protothreads are a good choice for implementing the quasi-parallelism required for KPNs: First, a context switch in protothreads basically amounts to returning from one function and calling another one. This takes considerably less time than context switches for kernel-space or user-space threads, as will be shown in Section VII. Second, all threads in a protothreads environment share a single stack. This is a big advantage in memory-constrained systems where dimensioning the stacks for individual processes is often a difficult task. Third,

prothreads are basically implemented using a set of macros. This means that there is also little overhead in terms of code size in contrast to kernel-space or user-space threads which require a separate kernel for execution. Forth, prothreads are architecture-independent, requiring only an ANSI-C compliant compiler. Clearly, porting a kernel-space or user-space thread library to a platform is more involved, usually requiring the use of inline assembler for accessing certain CPU registers to implement the thread context switch.

The main disadvantage of prothreads compared to kernel-space und user-space threads is that the (potentially) blocking windowed FIFO primitives must be contained within one (FIRE) function because blocking statements cannot be used in (nested) functions called from a prothread. Compared to the advantages, this loss of flexibility for organizing the application code is a small penalty, however.

Windowed FIFO communication on single memory. Using windowed FIFOs for communication instead of ordinary FIFOs is particularly advantageous for processes that share the same memory because unnecessary copying of data can be completely avoided by directly accessing the FIFO channel buffer. In distributed memory architectures, usually all the processes executing on a single processor share the same local memory and can thus profit from this implementation. In shared memory architectures, this implementation can even be used for processes executing on different processors.

Fig. 2 illustrates our implementation of the windowed FIFO that consists of a ring-buffer and four pointers to store the state of the windowed FIFO. Data are enqueued at the tail and dequeued from the head whereby each of the four windowed FIFO access routines basically just moves one of the four pointers. Since CAPTURE and RESERVE are required to return pointers to a contiguous piece of memory, CAPTURE and RESERVE limit the window size accordingly if a buffer "wrap-around" would occur.

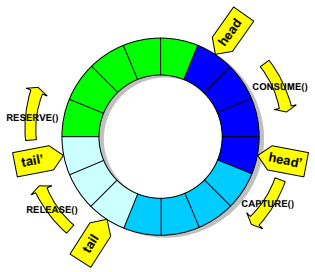

Fig. 2. Implementation of windowed FIFO.

Windowed FIFO communication between memories. While the implementation of the runtime-system using prothreads and the implementation of windowed FIFO communication on a single memory are architecture-independent, the implementation of the windowed FIFO communication between memories is architecture-dependent. In any case, to be able to parallelize computation and communication, an implementation will utilize DMA controllers, if available. Apart from that, there are many possibilities for implementing the windowed FIFO. The FIFO buffer can be implemented in the memory of the sender, the receiver, or both of them. Similarly, either the sender or the receiver could issue the DMA request. Finally, also a third processor could be used for the buffering and coordination required in windowed FIFO communication. In Section VI, a concrete windowed FIFO implementation for the Cell Broadband Engine is discussed.

B. Software Synthesis

The software synthesis for a KPN application specified according to Section IV basically consists of two steps, namely embedding each process into a prothread and creating a MAIN function for each processor.

The result of the first step is illustrated in Listing 4. The process structure p stores the local data of a process, including a variable $linecount$ that is used as the argument for the prothreads macros: Basically, $linecount$ is used to store at which point a function exits which is necessary in the PT_WAIT_UNTIL and PT_END macros. This enables to jump back to that point when reentering the function which is implemented in the PT_BEGIN macro. For further details about these macros, we refer to [2]. Clearly, the variables used in FIRE either have to be declared static or reside in the local data p of the process. Note that to create the code shown in Listing 4, an automated source-to-source code transformation is used.

Listing 4. Implementation of the process shown in Listing 1 based on prothreads.

```
01 procedure FIRE(ProcessData *p)
02   PT_BEGIN(*(p->linecount));
03   PT_WAIT_UNTIL(*(p->linecount),
04     p->wfifo_in->CAPTURE(rbuf, size) == size);
05   PT_WAIT_UNTIL(*(p->linecount),
06     p->wfifo_out->RESERVE(wbuf, size) == size);
07   manipulate();
08   p->wfifo_in->CONSUME();
09   p->wfifo_out->RELEASE();
10   PT_END(*(p->linecount));
11 end procedure
```

In the MAIN function on each processor, basically just windowed FIFO channels and processes are initialized before entering the scheduler that repeatedly calls the FIRE functions of processes bound to a processor as outlined in Listing 5.

Listing 5. MAIN used in the prothreads runtime-system that executes the processes $process1$ and $process2$ bound to a processor.

```
01 procedure main()
02   process1->INIT(process1_data);
03   process2->INIT(process2_data);
04   do while (true)
05     process1->FIRE(process1_data);
06     process2->FIRE(process2_data);
07   end do
08 end procedure
```

Finally, we did not consider a particular way to boot a system, distribute the MAIN functions to each processor, and invoke them. We use existing operating system services for that purpose, as explained in the next section.

VI. Efficient Execution of KPNs on the CBE

In the previous sections, the basic ideas that allow the efficient execution of KPN applications on multi-processor systems have been described. In this section, a concrete implementation of a runtime-system and software synthesis is described. Specifically, the distributed operation layer framework [3] has been extended to execute applications on the Cell Broadband Engine (CBE) [4]. The described framework is available online: http://www.tik.ee.ethz.ch/∼shapes/.

Note that the approach based on protothreads and windowed FIFOs is architecture-independent and porting the implementation to other multi-processor systems is thus relatively easy.

A. Cell Broadband Engine

A high-level block diagram of the CBE is outlined in Fig. 3. The CBE is a heterogeneous multi-processor system and consists of one 64-bit PowerPC processing element (PPE) and eight synergistic processing elements (SPEs). All processors are clocked at a frequency of 3.2 GHz and are interconnected by a ring bus clocked at 1.6 GHz.

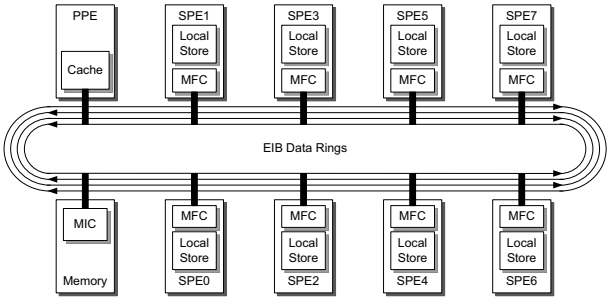

Fig. 3. Simplified block-diagram of the CBE.

While the PPE is a PowerPC processor, the SPEs are RISC processors with 128-bit SIMD organization that can process up to four 32-bit integer or single-precision float values per clock cycle. An SPE can only access its own local store, a small memory of 256 kBytes. A DMA engine enables the access to the main memory and the communication with other SPEs. The element interconnect bus (EIB) that interconnects all processors and the memory interface controller (MIC) consists of four data rings (two running clockwise and two running counterclockwise). The SPEs are connected to the EIB through memory flow controllers (MFC) which are responsible to manage DMA requests of SPEs. Besides the communication via EIB, the PPE and SPEs can exchange 32-bit messages using a mailbox system.

Concerning the implementation of KPNs on the CBE, the following restrictions need to be taken into consideration:

- Preemptive scheduling is fully supported on the PPE by using POSIX threads. Contrary, to the best of our knowledge there is no kernel-space or user-space thread library for the SPEs available which could be used to implement quasi-parallelism.
- Due to the low bandwidth and high delay of mailbox communication, mailboxes can only be efficiently used for exchanging synchronization messages, but not for data transfers.
- Due to the structure of the MFC, the source and the destination buffers of a DMA transfer must be "naturally" aligned. A memory block of N bytes is called "naturally" aligned when the $\log_2 N$ least significant bits of its address are zero.

B. CBE Runtime-System

The runtime-system to execute a KPN application on the CBE follows the principles explained in Section V-A. On the PPE as well as on each SPE, multiple processes are executed in a quasi-parallel fashion using protothreads. Windowed FIFOs that connect processes executing on the same processing element are implemented according to Fig. 2. In case of the PPE, the channel buffer is implemented on the main memory, while for SPEs it is implemented on the corresponding local store. Note that this limits the sum of the size of the windowed FIFO buffers mapped to an SPE to 256 kBytes minus the fraction occupied by program code.

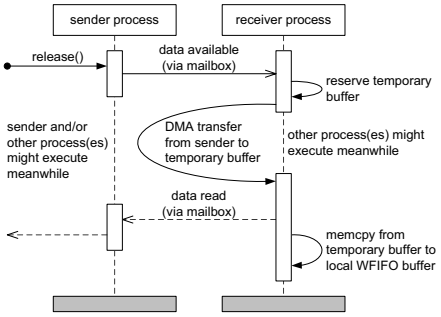

Fig. 4. WFIFO communication protocol between processes located on different processors.

The single processors of the CBE only interact via windowed FIFOs. The developed protocol is depicted in Fig. 4. As Fig. 4 shows, windowed FIFO communication only involves the sending and the receiving processes and processors, respectively, thus not requiring any global coordination. The basic principle underlying the protocol is that data are forwarded as soon as possible from the sender to the receiver. This means that data transfers from the local store of the sender to the local store of the receiver are initiated as soon as the sender has invoked a RELEASE. The receiver regularly checks whether data are ready and carries out the corresponding transfer. Consequently, RESERVE, CAPTURE, and CONSUME are executed completely locally and do not require any synchronization between processors.

The protocol works as follows: Whenever the sender has released a window, it informs the receiver that new data are available. Afterwards, the sender can continue as long as the local windowed FIFO buffer is not full. Otherwise, the sender blocks and waits until it receives a notification message that the transfer has been completed. Meanwhile, if other processes are ready to execute, they can be executed, of course. The actual transfer is carried out by the receiver which

checks whether new data are available on every invocation of CAPTURE or whenever a process has finished its FIRE routine. If a message about the availability of new data is present, the receiver sets up a corresponding DMA transfer to copy the data from the remote memory to its own local store. Due to the alignment restriction, however, the MFC cannot directly copy data between the involved windowed FIFO buffers. To deal with this issue, a naturally aligned temporary buffer is used at the receiver that serves as the destination for the DMA transfer. If the window of the sender is not naturally aligned, the source address is changed to the largest naturally aligned address that is smaller than the source address, while increasing the number of bytes to transfer. After the DMA transfer has been set up, the receiver blocks until the transfer has been completed. Like on the sender side, other processes can run meanwhile. Finally, the receiver informs the sender about the completion of the transfer, possibly unblocking a RESERVE invoked earlier at the sender. To complete the transfer, the received data are copied from the (aligned) temporary buffer to the (potentially unaligned) buffer of the windowed FIFO, discarding the bytes that have been transferred due to the alignment restriction.

Finally, to bootstrap a KPN application on the CBE, routines provided by the CBE SDK [34] and Linux are used. Basically, running Linux on the PPE allows the proposed runtime-system to run as a single Linux process. During bootstrapping, the runtime-systems for the SPEs are dispatched by the PPE. Afterwards, an application can execute in a completely distributed fashion.

Besides a runtime-system for the CBE, runtime-systems for executing KPN applications on single-processor systems have been implemented for the experiments. Specifically, single-processor runtime-systems based on pthreads, SystemC, YAPI, and protothreads have been implemented. Since the approach is the same as for a single processor in a multi-processor system, we refrain from giving further details.

C. Software Synthesis for the CBE

The goal of software synthesis is to implement a given high-level specification of a KPN application on the runtime-system described above. Fig. 5 shows an overview of this step: Given a process network description, the source code of the processes, and the mapping (binding) of processes to processors, the architecture and runtime-system dependent source code and Makefiles for their compilation are automatically created in software synthesis.

For this purpose, we use the distributed operation layer (DOL) framework which defines XML schemata for describing the topology of KPN applications and their mapping to multi-processor systems, an API, and a set of coding rules for the source code of processes (see Section IV). Written in Java, the basic components of the DOL are parsers for the XML specification files, corresponding data structures that can be conveniently accessed, and code generators. Based on this framework, the software synthesis steps described in Section V-B have been implemented in less than 2000 lines of Java code.

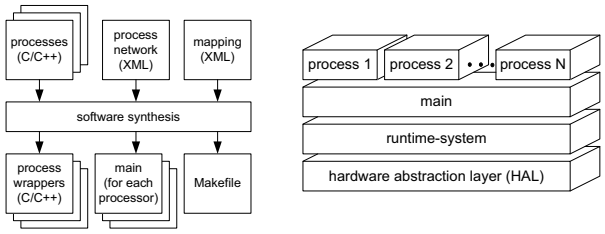

Fig. 5. Software synthesis (left) and the structure of the resulting software stack for a single processor (right).

VII. Experimental Results

In this section, we present several experiments demonstrating the performance of KPN applications implemented using the proposed runtime-system based on protothreads and windowed FIFO communication. We also compare the performance to other runtime-systems for KPNs based on the pthreads, SystemC, and YAPI library, respectively, for different configurations and single-processor as well as multi-processor systems. The goal is to demonstrate the main advantages of our approach which are: (1) low runtime overhead for a single-processor implementation, (2) high bandwidth and speed-up close to the theoretic limit in the case of a multi-processor implementation, (3) execution of fine-grained KPNs with the efficiency of a coarse-grained implementation, and finally, (4) implementation in a small memory footprint.

To enable the replication of the experiments, a package containing all the required sources and scripts is available online: http://www.tik.ee.ethz.ch/∼shapes/.

A. Single-Processor Performance

Experimental Setup

To determine the single-processor performance, we execute synthetic KPN applications using the four runtime-systems mentioned above under Linux (kernel 2.6.28) running on an Intel Xeon processor clocked at 3.06 GHz. The used compiler is GCC 4.1.2 and the optimization level -O2 is used for all experiments. (The reason for running the single-processor experiments on an Intel Xeon processor rather than on the PPE or the SPE of a CBE is that suitable ports of the SystemC and YAPI libraries were not available for the CBE.)

To measure the context switching and communication times, we have designed two synthetic applications, referred to as SINGLEFIFO and PINGPONG. The SINGLEFIFO application (see Listing 6) has been designed to measure FIFO access times, its basic functionality being to write/read to/from a FIFO channel. As only a single process is executed, no context switches occur. On the other hand, the PINGPONG application (see Listing 7) has been implemented to measure context switching times. When running PINGPONG, the runtime-system is forced to switch between the two processes during each invocation of FIRE because the processes communicate via FIFO channels with a capacity that equals the size of the communicated tokens. The context switching time is then estimated by subtracting the FIFO access times from the total execution

time. For comparison, we implemented both applications using standard and windowed FIFOs.

Listing 6. SINGLEFIFO application for measuring FIFO access times.

```
01 procedure process_fire(ProcessData *p)
02   for (i = 0 to ITERATIONS - 1)
03     p->fifo->write(*buf, size);
04     p->fifo->read(*buf, size);
05   end for
06   for (i = 0 to ITERATIONS - 1)
07     p->wfifo->RESERVE(*buf, size);
08     p->wfifo->RELEASE();
09     p->wfifo->CAPTURE(*buf, size);
10     p->wfifo->CONSUME();
11   end for
12 end procedure
```

Listing 7. PINGPONG application for measuring context switching times.

```
01 procedure process1_fire(ProcessData *p)
02   for (i = 0 to ITERATIONS - 1)
03     p->fifo1->write(*buf, size);
04     p->fifo2->read(*buf, size);
05   end for
06 end procedure
07 procedure process2_fire(ProcessData *p)
08   for (i = 0 to ITERATIONS - 1)
09     p->fifo1->read(*buf, size);
10     p->fifo2->write(*buf, size);
11   end for
12 end procedure
```

Context Switching Time

In Fig. 6, the context switching time and the (windowed) FIFO access time are compared for the four APIs. As expected, the protothread implementation introduces the smallest context switching overhead. When compared to a user-space thread implementation (YAPI or SystemC), a protothread context switch is about 8 to 18 times faster. When compared to pthreads, a protothread context switch it is approximately 200 times faster. Note that the measured context switching times include the execution of the scheduler which is likely to consume the majority of the context switching time.

Windowed FIFO Communication

To measure and compare the execution time for accesses to windowed and standard FIFOs, the SINGLEFIFO application was used. The results are shown in Fig. 6 where FIFO access stands for a single READ or WRITE and windowed FIFO access stands either for RESERVE, RELEASE or for CAPTURE, CONSUME. (For both FIFO types, the read and write primitives take approximately the same time.) While the standard and windowed FIFO access times are similar for small accesses, the windowed FIFO is considerably more efficient for large accesses: In case of the windowed FIFO, basically only the pointers shown in Fig. 2 have to be updated. The duration of the involved operations is thus independent from the actual number of transmitted bytes. In case of the standard FIFO, however, the data to transmit actually need to be copied into the FIFO buffer, increasing the access time as the number of transmitted bytes increases.

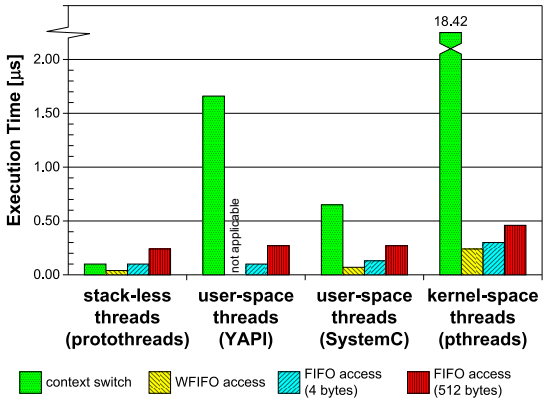

Fig. 6. Comparison of execution times for a context switch and (windowed) FIFO access in different thread implementations. The duration of a context switch and a windowed FIFO access in the protothread implementation amount to roughly 300 and 150 clock cycles, respectively, thus enabling the efficient execution of rather fine-grained process networks.

Code Size

Table II shows that in addition to high performance, a runtime-system based on protothreads introduces only a small overhead in terms of application size.

TABLE II
SIZE OF THE BINARY OF THE PINGPONG APPLICATION IMPLEMENTED USING DIFFERENT KPN RUNTIME-SYSTEMS.

	protothreads	YAPI	SystemC	pthread
size [kBytes]	42	1443	1343	461
size after stripping [kBytes]	9	113	221	32

B. Multi-Processor Performance

The first part of this section has shown that a runtime-system based on protothreads together with windowed FIFO communication enables the efficient execution of KPNs on single-processor systems. We now examine the performance on a multi-processor system.

Experimental Setup

A Sony PlayStation 3 running Yellow Dog Linux 6.1 (kernel 2.6.23) is used for the experiments. The Sony PlayStation 3 contains a single CBE whereby the PPE and six SPEs are available for user applications. We used the GCC compilers PPU-G++ 4.1.1 and SPU-G++ 4.1.1 with all optimizations enabled.

Context Switching and FIFO Communication on the CBE

Using the same experiments as in the single-processor evaluation, we measure the context switching time and compare the standard and windowed FIFO access times for the PPE and SPEs. In Fig. 7, the execution times of a context switch and of various FIFO accesses are outlined. The plot clearly indicates that the PPE and the SPE behave in a similar way as the Intel Xeon architecture used in the single-processor evaluation.

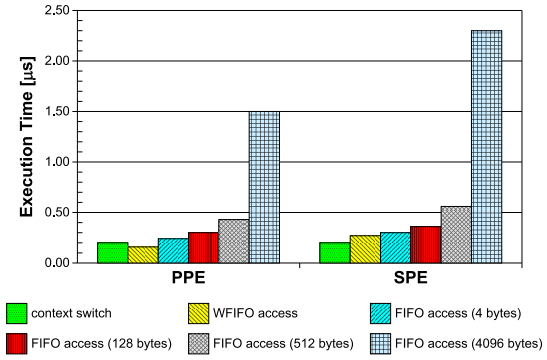

Fig. 7. Comparison of the execution times of a context switch and various FIFO accesses on the CBE.

Inter-SPE Data Transfer Rate

To measure the peak data transfer rate between SPEs, two experiments are performed: First, a chain of six processes was mapped onto the six SPEs (one process per SPE). Second, a chain of 12 processes was mapped onto the six SPEs (two processes per SPE). To connect the processes in the chains, windowed FIFO channels with a size of 16384 bytes are used which corresponds to the maximum size of a single DMA transfer on the CBE.

Fig. 8 shows the aggregate inter-SPE data transfer rates for the two chains whereby the number of bytes transmitted in a single windowed FIFO access is varied between 512 bytes and 16384 bytes. The observed peak data rates are 9.8 Gbytes/s when one process is executed on each SPE, and 10.9 Gbytes/s when two processes are executed on each SPE. In the latter case the data rate is higher because the data transfers initiated by the two processes on each SPE can be partially overlapped.

Even higher data rates have been reported in [30], where in a similar test setup a data rate of approximately 17.3 Gbytes/s has been achieved for an MPI implementation on the CBE. That implementation assumes naturally aligned source and destination buffers, thereby saving the memory copy operations required in our implementation.

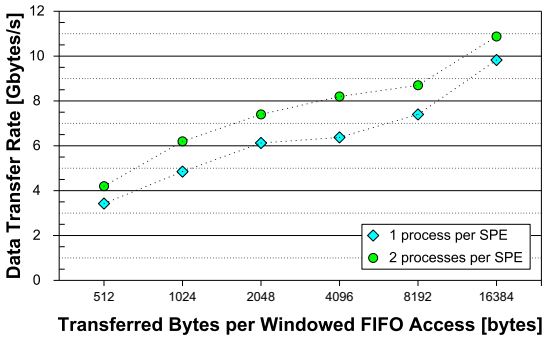

Fig. 8. Aggregate inter-SPE data transfer rate.

Speed-Up due to Parallelization

To evaluate the speed-up of applications that can be achieved using the proposed framework, a motion JPEG (MJPEG) decoder is used. MJPEG is a video codec in which each video frame is separately compressed as a JPEG image. The process network and mapping used for this evaluation are depicted in Fig. 9. We distinguish between a coarse-grained and a fine-grained implementation: In the coarse-grained implementation, a complete frame is decoded in each of the processes called “decode frame”. In the fine-grained implementation, “decode frame” splits a frame into segments of 40 macroblocks that are then subjected to inverse quantization, zigzag scan, and inverse discrete cosine transform. In both cases, the process “split stream” reads the video stream from a file and dispatches single video frames to the subsequent processes. “merge stream” collects the decoded frames and displays them. Each “decode frame” process (with its associated “child processes” in the case of the fine-grained implementation) is mapped to an SPE. “split stream” and “merge stream” are mapped to the PPE.

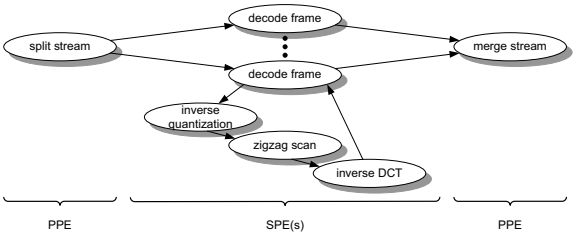

Fig. 9. KPN of the MJPEG decoder. The lower part of the KPN indicates the structure of the fine-grained implementation.

In Fig. 10, the time for decoding 3100 frames using the MJPEG algorithm is compared for implementations on a different number of SPEs. Obviously, the peak-performance can be achieved when a “decode frame” process is mapped to each of the six SPEs. In that case, a grayscale video of 320×240 pixels can be decoded with a frame rate of 942 frames/s. Mapping all processes to the PPE leads to a frame rate of 136 frames/s, thus a speed-up of almost seven can be achieved when using the PowerPC and all six SPEs. The plot also shows that by leveraging protothreads and communication via windowed FIFOs the overhead of the fine-grained version compared to the coarse-grained version is only about 5%.

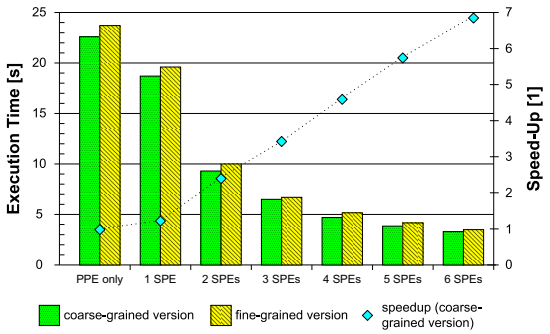

Fig. 10. Execution time and speedup of the MJPEG application for a varying number of SPEs. The speed-up of the coarse-grained implementation refers to the right y-axis.

VIII. Conclusion

In this paper, we have demonstrated that Kahn process networks can be efficiently executed on multi-processor systems. To achieve this goal, we propose to use protothreads to implement quasi-parallelism on the single processors in the system and windowed FIFOs to implement the communication between processes. This keeps the runtime-overhead very low such that speed-ups close to the theoretical bound can be achieved. Our solution can be easily ported to different architectures because only the implementation of the windowed FIFO communication between different processors is architecture-dependent. We demonstrated the viability of the proposed approach by running KPNs on the Cell Broadband Engine achieving speed-ups close to seven on seven processors.

Acknowledgment

This work was supported by the EU Framework Programme projects SHAPES, COMBEST, and PREDATOR under grant numbers 026825, 215543, and 216008.

References

[1] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. IFIP Congress*, Stockholm, Sweden, Aug. 1974, pp. 471–475.

[2] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," in *Proc. Int'l Conf. on Embedded Networked Sensor Systems (SENSYS)*, Boulder, CO, USA, Nov. 2006, pp. 29–42.

[3] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping Applications to Tiled Multiprocessor Embedded Systems," in *Int'l Conf. on Application of Concurrency to System Design (ACSD)*, Bratislava, Slovak Republic, Jul. 2007, pp. 29–40.

[4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, 2005.

[5] E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proc. of the IEEE*, vol. 83, no. 5, pp. 773–799, May 1995.

[6] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. P. Llopis, and P. Lippens, "C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems," *Design Automation for Embedded Systems*, vol. 7, no. 3, pp. 233–270, Oct. 2002.

[7] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A Retargetable Parallel-Programming Framework for MPSoC," *ACM Trans. on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 39:1–39:18, Jul. 2008.

[8] A. D. Pimentel, C. Erbas, and S. Polstra, "A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[9] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and Automated Multiprocessor System Design, Programming, and Implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 542–555, Mar. 2008.

[10] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, and T. D. Hämäläinen, "UML-Based Multiprocessor SoC Design Framework," *ACM Trans. Embedded Comp. Systems*, vol. 5, no. 2, pp. 281–320, May 2006.

[11] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, "Multiprocessor Systems Synthesis for Multiple Use-Cases of Multiple Applications on FPGA," *ACM Trans. on Design Automation of Electronic Systems*, vol. 31, no. 3, pp. 40:1–40:27, Jul. 2008.

[12] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF — A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems," in *Proc. First Swedish Workshop on Multi-Core Computing (MCC)*, Uppsala, Sweden, Nov. 2008.

[13] S. A. Edwards and O. Tardieu, "SHIM: A Determinstic Model for Heterogeneous Embedded Systems," *IEEE Trans. VLSI Syst.*, vol. 14, no. 8, pp. 854–867, Aug. 2006.

[14] S. A. Edwards, N. Vasudevan, and O. Tardieu, "Programming Shared Memory Multiprocessors with Deterministic Message-Passing Concurrency: Compiling SHIM to Pthreads," in *Proc. Design, Automation and Test in Europe (DATE)*, Munich, Germany, Mar. 2008, pp. 1498–1503.

[15] "eCos Home Page," http://ecos.sourceware.org/.

[16] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers, "YAPI: Application Modeling for Signal Processing Systems," in *Proc. Design Automation Conference (DAC)*, Los Angeles, CA, USA, Jun. 2000, pp. 402–405.

[17] K. Huang, D. Grünert, and L. Thiele, "Windowed FIFOs for FPGA-based Multiprocessor Systems," in *Proc. Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, Montreal, Canada, Jul. 2007, pp. 36–41.

[18] P. van der Wolf, E. A. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach," in *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004, pp. 206–217.

[19] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proc. Int'l Conf. on Compiler Construction*, Grenoble, France, Apr. 2002, pp. 179–196.

[20] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe, "A Lightweight Streaming Layer for Multicore Execution," *SIGARCH Comput. Archit. News*, vol. 36, no. 2, pp. 18–27, May 2008.

[21] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[22] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the Memory Hierarchy," in *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, Tampa, FL, USA, Nov. 2006.

[23] "OpenMP," http://openmp.org/.

[24] "Message Passing Interface," http://www.mpi-forum.org/.

[25] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagné, and G. Nicolescu, "Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia," *IEEE Trans. VLSI Syst.*, vol. 14, no. 7, pp. 667—680, Jul. 2006.

[26] K. Kim, J. Lee, H.-W. Park, and S. Ha, "Automatic H.264 Encoder Synthesis for the Cell Processor from a Target Independent Specification," in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, Atlanta, GA, USA, Oct. 2008, pp. 95–100.

[27] D. Nadezhkin, S. Meijer, T. Stefanov, and E. Deprettere, "Realizing FIFO Communication when Mapping Kahn Process Networks onto the Cell," in *Proc. Int'l Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*, Samos, Greece, Jul. 2009, pp. 308–317.

[28] N. Vasudevan and S. A. Edwards, "Celling SHIM: Compiling Deterministic Concurrency to a Heterogeneous Multicore," in *Proc. ACM Symposium on Applied Computing (SAC)*, Honolulu, Hawaii, USA, Mar. 2009, pp. 1626–1631.

[29] M. Ruggiero, M. Lombardi, M. Milano, and L. Benini, "Cellflow: A Parallel Application Development Environment with Run-Time Support for the Cell BE Processor," in *Proc. EUROMICRO Conf. on Digital System Design (DSD)*, Parma, Italy, Sep. 2008, pp. 645–650.

[30] S. Pakin, "Receiver-initiated Message Passing over RDMA Networks," in *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, Miami, FL, USA, Apr. 2008, pp. 1–12.

[31] T. M. Parks, "Bounded Scheduling of Process Networks," Ph.D. dissertation, University of California, Dec. 1995.

[32] M. Geilen and T. Basten, "Requirements on the Execution of Kahn Process Networks," in *Proc. European Symposium on Programming (ESOP)*, Warsaw, Poland, Apr. 2003, pp. 319–334.

[33] T. Basten and J. Hoogerbrugge, "Efficient Execution of Process Networks," in *Proc. Communicating Process Architectures (CPA)*, Bristol, UK, Sep. 2001, pp. 1–14.

[34] "IBM SDK for Multicore Acceleration," http://www.ibm.com/developerworks/power/cell/.