Technical Report: Kahn Process Network IR Modeling for Multicore Compilation

Sjoerd Meijer*, Sven van Haastregt*,
Dmitry Nadezhkin*, Bart Kienhuis *,

* *LIACS, University Leiden, Leiden Institute of Advanced Computer Science,*
Niels Bohrweg 1, 2333 CA, Leiden, Netherlands

ABSTRACT

The complexity of embedded applications has reached a point where the performance requirements of these applications can no longer be supported by embedded systems based on a single processor. Instead, emerging embedded System-on-Chip platforms are increasingly becoming multicore architectures. On these platforms two major problems emerge: how to design and how to program such multicore platforms in a systematic and automated way? We focus on the latter and are building compiler support for programming multicore platforms. For uniprocessor systems, auto parallelization techniques such as vectorization, instruction level parallelism, and software pipelining mainly focused on the back-end of the compiler. We believe that for multicore systems this is not sufficient and a difference can be made in the middle-end. Therefore, we present a compiler framework in which we use the Kahn Process Network (KPN) model of computation in the Intermediate Representation (IR) of the compiler. This IR model is crucial and allows us to focus on: 1) code generation for multicore platforms, and 2) KPN profiling and network restructuring transformations, where the former is a prerequisite for the latter. In this paper we focus on 1), and manually apply the network restructuring transformations. We demonstrate a working prototype compiler tool-chain that automatically creates multithreaded code from a sequential program specification and show results for the Cell processor.

1 Introduction

Multicore architectures are being introduced more and more to meet the required compute power of the applications. An example of such a platform is IBM's Cell processor [1]. The availability of these architectures is the first step in meeting the performance requirements. The next step and challenge is to take full advantage of these architectures; applications that were running in a single thread must be carefully partitioned and mapped onto the architecture. The arrival of multicore platforms requires

[1]E-mail: {smeijer,svhaastr,dmitryn,kienhuis}@liacs.nl

that a program can be expressed in partitions that are mapped onto different processors. The questions is how to express and model these partitions, i.e., a higher level of abstraction is required inside a compiler. This model is also a prerequisite to support high-level transformations on the partitions itself like splitting and merging. Typically, a compiler is organized as follows: it consists of a front-end (parsing), middle-end (high-level optimizations), and back-end (code generation) as shown in the right-hand side of Figure 1. The logical place for this higher level model is in the middle-end of the compiler [2]. Since we focus on compiler support for high-performance streaming applications, we propose to use the Kahn process network (KPN) [3] model of computation as the high level intermediate code representation (IR). A choice that is inspired by the work done in the Compaan compiler [4, 5]. The work presented in this paper is a continuation of the Compaan project but focuses on programmable multicore platforms rather than FPGA platforms.

In this paper, we explain the KPN model and show how we implement this model in the CoSy compiler framework, which is an industrial strength compiler framework. Making CoSy generated compilers that are *KPN aware* means creating an abstraction layer in the middle-end of the compiler. This enables us to iterate over high-level constructs like processes and FIFO data-structures that make up the KPN. The CoSy compiler allows us to make views, annotations or extensions onto the existing Intermediate Representation (IR) without breaking the compiler flow. Since the compiler flow is not disrupted, we can immediately use any code generator (e.g., for the Pentium, ARM, or PowerPC) without adaptation. Besides making the CoSy compiler KPN-aware, we also extended CoSy with a high-level *platform description*. This description describes a multi-processor platform in terms of number of processors, type of interconnects used between the cores etc. We also extended CoSy with a *mapping specification* allowing us to express how we should bind processes of the KPN to the components of the platform. Both the Platform and Mapping descriptions are shown in Figure 1.

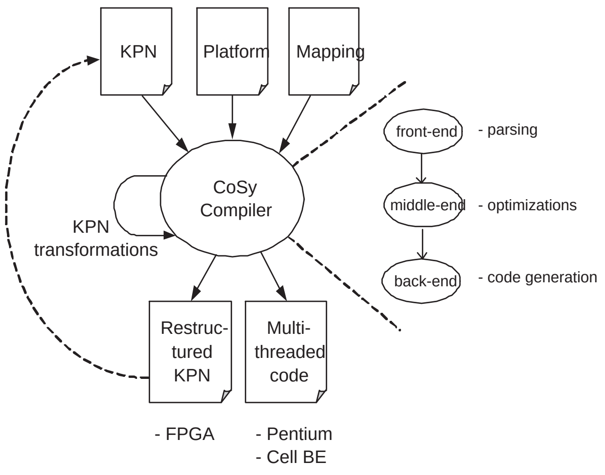

Figure 1: Overview of the CoSy Compiler

This paper is organized as follows; we continue this section by describing the need for high-level transformations and the lack of support in compilers. We present related work and position this work. We also describe the Cell processor which is our model of a multicore architecture. In Section 2, we introduce the CoSy compiler framework and show how we describe a Kahn process network, a multicore architecture and a mapping. In Section 3, we explain how we model the KPN inside CoSy. In Section 4, we explain how we do codegeneration from the KPN model. In Section 5, we show results obtain so far when running a Motion JPEG application on the Cell architecture. We conclude in Section 6.

1.1 Problem Description

Mapping an application written in an imperative language like Matlab, C, or Java onto a multicore architecture is a difficult task. It is difficult as it typically involves manual partitioning of the original code and memory assignments over multiple threads and memory structures while making sure that the threads cooperate correctly. This is a difficult, tedious, and error prone process. Instead, we believe that this partitioning step should be automated. The reason that mapping an application automatically onto a multicore architecture is so difficult, is because the computational model of an imperative language is in complete contrast to the architecture. Imperative languages use the concept of a single thread of control and a large global memory space. Multiprocessor architectures on the other hand, use concepts like autonomous processes and distributed memories. To bridge these two concepts, we believe that the KPN model of computation is very appropriate [6, 7]. The KPN model expresses an applications in terms of autonomous processes that communicate with each other using unbounded FIFOs [3]. The processes synchronize using a blocking read. If a process tries to read data from an empty FIFO channel, the process blocks until data becomes available. Once data becomes available again, the process unblocks and reads the data from the FIFO channel and continues processing.

The purpose of focusing on the KPN model in the middle-end of a compiler is to allow high-level transformations. Examples of these transformations are given in [8]. In this article, a JPEG application specified as a Kahn Process network was executed on a multicore architecture but still did not meet the performance requirements; one process from the network exceeds the average number of cycles greatly and becomes a bottleneck for the entire network. The paper shows that the bottleneck process can be eliminated by applying high-level transformations like *splitting* or *unrolling* and *merging*; the total execution time of the application is (dramatically) reduced. The splitting/unrolling transformation increases parallelism by splitting nodes in a network thereby distributing the workload of a single node over multiple nodes. Similarly, the merging transformation merges nodes in the process thereby decreasing the number of parallel processes. In the paper, all these transformations were handmade in the source-code of the application. To support these transformation inside a compiler, a high-level model is required that expresses the FIFOs, processes and network structure of a KPN. In [8], we have already shown that this high level model is not supported by the GCC compiler and that an implementation is not trivial. Since such model is lacking in GCC, it was very hard to copy or modify complete processes as these constructs cannot be distinguished anymore in the IR since all C++ classes expressing high-level elements like processes and FIFOs are lowered to `structs` in the IR of GCC. This make compiler support of high-level transformations very difficult.

1.2 Related Work and Paper Contribution

Our work was mainly inspired by the Compaan compiler [9, 4, 5]. This compiler translates static affine nested loop programs written in a subset of Matlab into a KPN representation. Compaan uses extensively exact dataflow analysis and the polyhedral model to obtain this KPN representation. From the KPN representation, we can generates VHDL files realizing the KPN in pure hardware using LAURA [10] or as a heterogeneous mix of hardware and microprocessors using ESPAM [12]. Similar to ESPAM, there is the SystemCoDesigner framework [13] that can also map KPNs to FPGAs. However, in contrast to the Compaan framework, an application has to be manual partition into SystemC models implementing the KPN model.

ESPAM has three input specifications: application, architecture, and mapping and uses these specifications to create and map an application onto a specified multicore architecture on a FPGA. ESPAM is not a compiler back-end but a very advanced KPN formatter that generates C-code for the micro-processors and the VHDL code. Similar in style is IMCA [15], which targets the Intel IXP network processors family instead of FPGAs. IMCA formats a KPN into C code that can be compiled by the Intel IXP toolset. The IMCA work is however discontinued in favor of the Cell processor as the IXP is too specific to network traffic processing. Instead, we decided to work on the Cell architecture as it provides a multi-core architecture with a much broader application use.

The Compaan compiler lacks a proper C front-end. An environment that can handle C code and generate KPNs is called `PN` [11]. This tool uses the SUIF compiler just as a front-end for parsing C files. However, no middle-end model is created inside SUIF and no (assembly) code generation is performed; it serves as an alternative front-end for ESPAM.

The Graphite project [14] introduces exact dataflow analysis and the polyhedral model in the GCC compiler. However, it is unknown whether the Graphite project will introduce a special middle-end model in GCC; this project is still work in progress.

By introducing the KPN model inside CoSy, we move away from using advanced KPN formatters as done in ESPAM and IMCA. By using CoSy, we get access to all classical compiler passes and optimizations based on the Control Dataflow Graph (CDFG) model. By making the KPN model of computation an integral part of the IR of CoSy, we can exploit both the strength of the KPN mode and the CDFG model.

1.3 Target Architecture

As a model of a multicore architecture, we looked at the IBM Cell processor (CELL). The CELL runs at 3.2 GHz and consist of one Power Processor Element (PPE) and 8 Synergistic Processor Elements (SPE). Each SPE has 256 kb of local memory that can be used for data and instructions. The PPU has access to a large memory space. The PPU and SPEs are connected via the Element Interconnect Bus (EIB). This bus runs at half the processor's core frequency and can perform 4 parallel transfers in any cycle. A SPEs cannot directly access the large shared memory of the PPU, but must transfer data form the large memory using DMA transfers over the EIB to its local memory.

A systematic mapping of a KPN onto the Cell is possible. A key aspect of the KPN model of computation is that no global scheduler is used; each process follows its own local schedule. This model fits well with the Cell as the PPU and SPEs can be considered to be autonomous processors. Therefore, mapping a KPN on the Cell means that processes become threads that are either mapped on the PPU or one of the SPEs. Another key aspect of the KPN model is that it does not assume a shared memory model. All exchange of data between processes happens over FIFO channels. These FIFO channels are easily mapped onto the EIB in case of PPU to SPE or SPE to SPE communication.

2 Tool-flow Overview

As compiler framework, we use the CoSy framework. It is a mature, well-established compiler generation framework that provides compiler engineers with a complete and solid foundation; CoSy is not a compiler, but a framework that enables one to quickly create a compiler using standard components (engines). Furthermore, parts of the IR and standard functionality to manipulate it, is automatically generated based on IR specifications. As described in [16], CoSy uses a compilation model that is different from compilers like GCC or Open64. This compilation model uses a set of engines that work in parallel on a single global data structure that makes the exchange of information/results with other engines easy. Access to the data is done using access routines from a generated library and macro package, the Data Manipulation and Control Package (DMCP). The engines operate on data structures that have been defined in a language called *fSDL*. This language specifies the permissions and views an engine has on the Intermediate Representation (IR). Also, extensions can be made and specified in the language. Based on the fSDL language specification, code is generated for accessing and manipulating the IR. We exploit the fSDL language concept to realize the middle-end extension in CoSy as will be explained in Section 3.

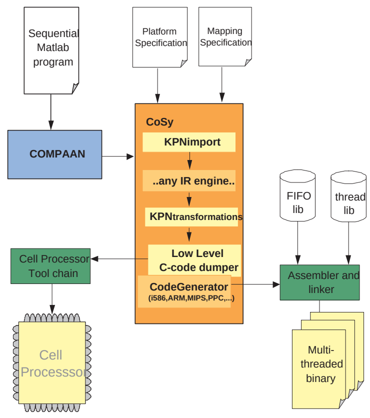

Figure 2: Tool-flow overview

To compile sequential applications to a multicore platform, we realized the tool flow as shown in Figure 2, which is a more detailed picture of Figure 1. We start with a sequential application written in a subset of the Matlab language. It is analyzed by the Compaan compiler which generates a KPN specification. This specification is a file describing the different processes of the network and how they communicate over the FIFO channels in an XML format. This XML file is parsed by the *kpnxmlimport* engine that we have created. During the parsing of the XML file, the KPN model is built as a high level Intermediate Representation (IR). We create annotations such that the Processes and FIFO channels can be recognized in the standard IR of the compiler. Due to the special way CoSy allows us to create these annotations in the IR, other engines or optimizations can continue to run without modification. In Section 3, we explain this annotation technique in more detail.

We used the *i586cg* code generation engine to create assembly output for a Pentium platform; this generates functionally correct multithreaded x86 assembly code that is very useful for debugging purposes. Although the Pentium is not our target architecture, it shows that we can have a high-level model in the IR and still generate x86 assembly code without any modifications in the back-end. By replacing the Pentium code generation engine by another code generation engine, we can generate code for other platforms.

We like to measure the performance of a KPN after applying high-level transformations on a multicore platform; in particular the Cell processor. Since no code generation engines exists yet for the SPEs (for the PowerPC a back-end exist) of the Cell processor, we had to take another route. We generated low-level C code that is optimized by the CoSy engines (see Figure 2). The C-code is compiled for the Cell using the available compiler chain based on GCC. When a SPE code generator becomes available, we can follow the same route as taken for the Pentium platform.

2.1 Kahn Process Network Specification

In the tool-flow given in Figure 2, a sequential program written in a subset of the Matlab language is first processed by the Compaan compiler. In [4], a four step approach is given to convert the complete class of Static Affine Nested Loop Program (SANLP) into a KPN. These steps are based on solving many Parametric Integer Linear Problems within a polyhedral framework. A SANLP is a program in which the loop bounds, array index expressions, and conditions of if-statements are expressed as affine functions of the loop iterators and program parameters. Applications in the domain of of streaming audio and video applications are naturally express as SANLPs.

Each assignment statement in the SANLP becomes a process in the KPN. Array references in the SANLP become FIFO channels after calculating the exact data dependences between read and write operations on the Arrays.

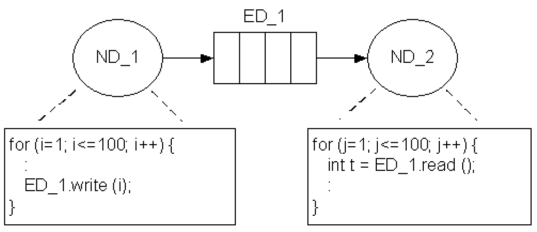

Figure 3: KPN Producer Consumer Pair

The simplest KPN is a producer-consumer pair. In Figure 3, such a producer-consumer pair is given as a KPN. Producer `ND_1` generates a token at each iteration point i and writes it to FIFO channel `ED_1`. In the same manner consumer `ND_2` reads data from the FIFO channel at each iteration point j.

2.2 Platform and Binding Specification

Traditional machine descriptions in compilers consist of files describing the number of registers, tree pattern matching rules, the processor pipeline, etc., as described for example in the GCC internals [18]. For uniprocessor systems this description suffices. For multicore platforms however, additional information is required about the different cores. An example of a platform description is given below:

```
<platform name="CELLBE20">
  <processor name="PPC1" type="ppc">
    <threading libname="pthread"/>
  </processor>
  <processor name="SPE1" type="altivec"/>
</platform>
```

The platform specification describes a platform named `CELLBE20`. It consists of one PowerPC core named `PPC1` and a core named `SPE1` that has altivec vector instructions. Furthermore, we indicated that we use the POSIX thread library (pthread) for implementing the processes of the KPN as threads. The mapping specification specifies which process of a KPN will run on which platform component. There is no automatic decision making involved in this process. The designer has to manually bind a process to a processor. An example of a mapping specification is given below:

```
<mapping name="myMapping">
  <processor name="PPC1">
    <process name="ND_1" />
    <process name="ND_2" />
  </processor>
  <processor name="SPE1">
    <process name="ND_3" />
  </processor>
</mapping>
```

Suppose we have a network consisting of 3 nodes. The mapping specification describes that nodes `ND_1` and `ND_2` will be mapped onto `PPC1`, and node `ND_3` onto `SPE1`. The communication between processes is deferred from the assignment of processes to processors.

3 KPN Modeling and Transformations

A Kahn process network is modeled as a graph and is defined as $G = (V, E)$, where V is the set of all processes and E the set of all FIFO channels. Using the fSDL language, we modeled the process network structure and introduced new data structures for the processes and FIFOs. For example, a process V is described by type `mirKPNprocess` and a FIFO channel E is described by type `mirKPNfifo`. The graph G is described by type `mirKPN`. Using these data structures, we reconstruct the process network in the IR of CoSy from the XML input generated by the Compaan compiler (see Figure 2).

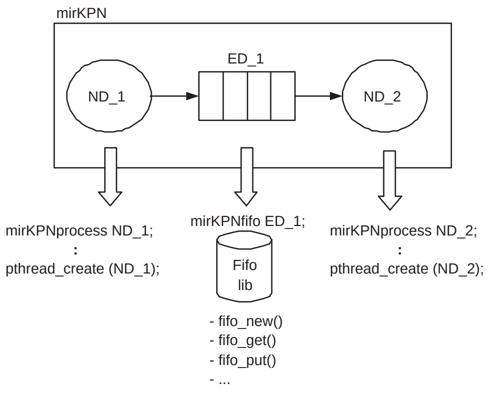

Figure 4: Overview of the KPN modeling

Figure 4 shows again the producer-consumer example in which `ND_1` is a producer process and `ND_2` a consumer process. They communicate over FIFO channel `ED_1`. The processes are instances of the new data structures created in the middle-end IR: the `mirKPNprocess` data structure represents a node of the Kahn process network. Likewise, the FIFO channel `ED_1` is an instance of the `mirKPNfifo` data structure. The entire Kahn process network is contained in another structure, called `mirKPN`. In the next sections, we discuss these new middle-end IR structures in more detail and explain how the FIFO functions are implemented.

3.1 The mirKPNprocess data structure

In the KPN model of computation, a process retrieves data from incoming FIFO channels, processes the data, and sends data to outgoing FIFO channels. Therefore, the `mirKPNprocess` data structure is defined in the fSDL language as follows:

```
domain mirKPNprocess :
<
  Process  : mirProcGlobal,
  Outfifos : LIST (mirKPNfifo),
  Infifos  : LIST (mirKPNfifo)
>
```

The `Process` field contains a pointer to a `mirProcGlobal`. This datastructure is used by CoSy to represent procedures and functions in the default IR. We populate instances of the `mirKPNprocess` data structure when parsing the KPN specification in XML format as described in Section 2.1. Since `mirProcGlobal` is just a procedure definition in CoSy, we do not break the normal compiler-flow; we model the processes as procedures and use function calls to a FIFO library in order to implement interprocess communication. We use existing data structures, but do not change them by embedding the components of a KPN in a view on the IR. As a consequence, the KPN view never interferes with the default IR and therefore all available CoSy engines can operate without changes or validation. A `mirKPNprocess` has fields `Outfifos` and `Infifos`, which contain lists of FIFO channels to which a process write and read tokens, respectively. These fields are both implemented as a `LIST`, which in CoSy terminology is called a *functor*. Based on this specification, macros for easy accessing of fields, traversal, and modification of a list is automatically generated.

3.2 The mirKPNfifo data structure

The FIFO channels in a Kahn process network connect the different processes to each other. In the KPN model of computation, both the input and the output side of a FIFO channel are connected to exactly one process and therefore the `mirKPNfifo` data structure is defined in the fSDL language as follows:

```
domain mirKPNfifo :
<
  Source : mirKPNprocess,
  Dest   : mirKPNprocess,
  Name   : NAME
>
```

The `Source` and `Dest` fields are pointers to the processes writing and reading to/from the FIFO channel. Thus, we can easily access the corresponding `mirKPNprocess` structure. The `Name` field contains the name of the channel, which is provided in the original XML input, and is used to identify the FIFO channel. It is important to realize that we do not really implement the FIFOs. We only identify them by their names, create a corresponding `mirKPNfifo` and declare them as local variables in the different procedures. By the time we do the code generation for a particular platform, we can use the most optimal implementation for the FIFO for that platform. This may include that the FIFO is realized over a bus as done for the Cell processor, using shared memory, or other available communication structures.

To execute the KPN on the Pentium or Cell platform, we have created FIFO libraries that implements the FIFO channels by providing the following functions:

We import these functions using the `extern` keyword and link our binary against this FIFO library. We use two initializing routines that create and destroy the FIFO channels using the functions `fifo_new` and `fifo_delete`. In the IR, we create function call nodes to `blocking_fifo_get` and `blocking_fifo_put` in order to implement the FIFO communication.

`fifo_new`	Create a new FIFO buffer
`fifo_delete`	Free a FIFO buffer
`blocking_fifo_get`	Read a token (blocking)
`blocking_fifo_put`	Write a token (blocking)
`is_fifo_empty`	Check if buffer is empty
`is_fifo_full`	Check if buffer is full

3.3 The mirKPN data structure

The structure of the network is stored as a graph using the `GRAPH` functor:

```
domain mirKPN :
<
  KPNid    : INT,
  KPNgraph : GRAPH (mirKPNprocess, mirKPNfifo)
>
```

Now we have the entire Kahn process network structure available in the middle-end of our compiler framework, which allows us to reason about it. This is necessary since the automatically generated network may not give the desired performance as discussed in [8, 19] and a number of transformations need to be available to allow a designer to play with parallelism; 1) either to increase parallelism, or 2) decrease parallelism. To realize, for example, the splitting transformation as discussed in [8] we now operate on the introduced KPN abstraction nodes. Using the `mirKPN`, `mirKPNprocess` and `mirKPNfifo` data types, it becomes straightforward to make copies of processes, create new FIFOs, and thus restructure the KPN. Although this is not fully implemented yet, we show the usefulness of the model and the IR specification with an example. For the *mirKPN* node, the CoSy framework automatically generates functions to manipulate and visit the IR node. For example, `GRAPH_KPN_VERTEX_LOOP` and `GRAPH_KPN_EDGE_LOOP` are generated macros to visit all processes and FIFOs respectively. The following code fragment illustrates how this all could be used to realize a simple profiler that can provide information steering the network transformations:

```
mirKPNprocesses p;
mirBasicBlock  bb;

GRAPH_KPN_VERTEX_LOOP (kpn, p) {
  CFG cfg = p->mirProcGlobal.Cfg;
  int bbcounter = 0;
  CFG_BB_LOOP (cfg, bb) {
    bbcounter += bb->UseEstimate;
  } CFG_BB_ENDLOOP;
} GRAPH_KPN_VERTEX_ENDLOOP;
```

In the example, a simple *KPN profiler* is implemented that indicates how many times a basic block was executed. After the KPN is built and executed, profile information is created that is loaded back into the compiler. This information is used to calculate the actual use of a basis block. This profile framework is part of CoSy. In the code fragment, we loop over all processes from the KPN. Then, for each process, a pointer to the Control Flow Graph (CFG) is obtained. All its basic blocks are visited and the sum of the basic block counter is calculated. This number is a first indication which processes would be candidates for the splitting or merging transformation.

4 Code Generation

To test and verify the KPN model, we generated multithreaded Pentium (x86) assembly code from the IR based on the middle-end data types. The code generation is straightforward, since the CoSy framework already includes the `i586cg` code generation engine for the Pentium processor. This engine takes the IR as input and outputs an assembly file which can be assembled and linked. To realize concurrent processes, we rely on the POSIX Thread (Pthread) library [20]. To execute the network, the assembly code needs to contain some additional code to set up the process network, i.e., initializing the FIFO buffers and creating the threads. This is done in the `main` function which is describe next.

4.1 Main Function

The `main` function is called upon the start of the binary created by our tool chain. The following IR code fragment shows the main function that is generated for the producer-consumer example described in Section 3.

```
EXPORT PROC main
  int5: pthread_t_ND_1
  int5: pthread_t_ND_2
  int4: exitcode
BEGIN
bb0:
  begin
  ED_1 <- fifo_new(0, 100)
  pthread_init()
  pthread_create(pthread_t_ND_1, NIL, ND_1, NIL)
  pthread_create(pthread_t_ND_2, NIL, ND_2, NIL)
  exitcode <- fifoObserver()
  call destroyNet()
  return exitcode^ to bb1
bb1:
  endproc
END
```

The code fragment given above is a direct dump of the middle-end IR. It is not meant to be compilable, but provides a convenient high-level view that we used to check the correctness of the IR. The code fragment is the lowest level of detail we got exposed to in the compiler. The CoSy code generation back-end converts the code to assembly code for the right target platform.

The `main` function starts by creating all FIFO buffers of a particular size (e.g., 100 locations) using a call to `fifo_new`. Next, a thread is created for each process in the network. In our case, we use the Pthread library and this library is initialized using a call to `pthread_init`. An individual thread is created by invoking `pthread_create`. After the creation of the threads, the main thread invokes the FIFO observer function. The purpose of the FIFO observer is to detect the termintation of the network. At this point, the process net is up and running. The Pthread library takes care of scheduling the processes. Once a process needs to wait for a FIFO buffer due to either a blocking read or write operation, the process calls the `sched_yield` function to switch execution to another process. A KPN finished its execution when the FIFO observer returns to the `main` function. The value returned by the FIFO observer indicates whether the execution was successful or not. The `main` function releases the resources allocated for the process net using a call to `destroyNet` and then returns the exit code to the operating system.

4.2 Node Threads

The `main` function first creates the FIFO buffer `ED_1` and then makes `pthread_create` function calls with the processes `ND_1` and `ND_2` of the KPN as actual parameters. The code that has to be executed in each node is described in the KPN specification in the XML format. A particular characteristic of the KPN is that processes are expressed as polytopes. These polytopes describe regions in an iteration space defined by for-loops in the original sequential program. The regions describe which tokens need to be read (ipdstatements) or written (opdstatements) from a particular FIFO. The polytopes are calculated by the Compaan compiler. Each polytope is converted into a list of conditionals in which each conditional is a linear expression. This list of conditionals together with the read and write procedures and the function to be executed in a process make up the `mirProcGlobal` procedure. The code generator in the back-end transforms these procedures into optimal assembly code for target processors.

The following code fragment shows the IR dump for the producer process shown in Figure 3.

```
STATIC PROC ND_1
DECLARE
  int4: i
  ptr18: out_0
BEGIN
bb0:
  begin
  i := 1
  goto bb1
bb1:
  if i <= N then bb2 else bb4
bb2:
  call GenerateToken(out_0)
  call blocking_fifo_put(ED_1, out_0)
  goto bb3
bb3:
  i := i+1
  goto bb1
bb4:
  call pthread_exit(NIL)
  goto bb5
bb5:
  endproc
END
```

The code calls the function `GenerateToken` for each iteration point i and writes each generated (“produced”) token to the FIFO channel designated by ED_1. After N iterations, the thread is terminated using a call to `pthread_exit`. Basic block $bb1$ represents a conditional obtained from the polytope representation in the XML file.

The IR code for the consumer process of the producer-consumer example is identical to that of the producer process, except that block `bb2` should be replaced by the following code:

```
...
bb2:
  in_0 <- blocking_fifo_get(ED_1)
  call PrintToken(in_0)
  goto bb3
...
```

This thread first reads a token from the FIFO channel designated by ED_1, that is, the FIFO channel to which the producer thread has written its tokens. Next, this token is sent to the function `PrintToken`, which “consumes” it.

5 Experiments and Results

The tool-flow shown in Figure 2, enables us to automatically compile stream-based applications written in a subset of Matlab into a parallelized binary. The experiment we conducted was to map a M-JPEG application onto the Cell architecture. The M-JPEG application performs JPEG compression on each image coming in from a video stream without inter-frame predictive coding. The Kahn process network for the M-JPEG application consists of 9 nodes (threads) and 54 FIFOs. We wrote the code that implements a FIFO buffer on the Cell as well as the read and write primitives. The CoSy compiler dumped for each process in the KPN low-level optimized C-code. This code together with the FIFO implementation was compiled for the Cell using the available GCC compiler. We first mapped the sequential M-JPEG application onto the PPU only. It needed 59.72 million cycles to finish. This provided the base line. Next, we generated threads for each node in the KPN and executed the threads only on the PPU. It needed 95.55 million cycles to finish. We observed a slow down caused by the slow FIFO communication between threads and the occurrence of many context switches.

experiment	**# Threads**	**PPU / SPEs**	**# Cycles**	**Speed-up**
sequential	1	1 PPU	59.72	1
parallel	9	1 PPU	95.55	.62
merge	2	1 PPU, 1 SPE	63.83	0.94
merge/unroll 2x	3	1 PPU, 2 SPEs	49.27	1.21
merge/unroll 3x	4	1 PPU, 3 SPEs	49.88	1.20

Table 1: M-JPEG running on the Cell

To improve the performance of the M-JPEG application on the Cell, we therefore apply the high-level transformations of merging and unrolling using the techniques discussed in [19]. The results are shown in Table 1. The DCT calculation in M-JPEG is the most compute expensive and we first move the DCT thread to a single SPE. This can easily be expressed using the Mapping specification. The remaining 8 threads on the PPU are merged to a single thread. We are faster then the parallel specification but slower then the sequential version due to the communication overhead between the PPU and the SPE. Next we unroll the DCT thread 2 times. The threads on the PPU are again merged. We observe a speed-up of 1.21. Mapping three DCT threads on three SPEs does not further reduce the execution time. The communication between SPEs and the PPU becomes dominant. A better FIFO implementation should be able to take advantage of additional SPEs.

6 Conclusion and Future Work

Our goal was to introduce a model in the middle-end of a compiler to come eventually to a multicore compiler that can handle network restructuring transformations. We have selected the Kahn process network model as our middle-end model, based on the work done in the Compaan compiler project. To represent the different components of a KPN (processes, FIFOs), we have exploited the 'view' concept in the CoSy compiler. We could model the elements of the Kahn process networks in the middle-end quickly, without effecting the overall flow of the compiler. This provides us the framework in which we can do: 1) code generation for multicore platforms, 2) high-level restructuring transformations such as merging and splitting nodes.

Using our developed tool-flow, we have shown that we could obtain a speed-up of 21% for the M-JPEG application running on the CELL platform. We could automatically generate a multi-threaded version of M-JPEG and assign threads to different processors available on the Cell. We also showed the effects of the high-level transforms.

The presented framework provides a foundation to come to a multicore programming environment for high-performance stream-based applications. Future work is to migrate the Compaan compiler directly in CoSy. This means moving exact dataflow analysis and the polyhedral model into CoSy. When this step is done, we can benefit from the ANSI C-compliant front-end and all available optimizations. Also, we will continue to improve and extend the high-level transformation engines in CoSy and to extend the type of multicore platforms supported by the back-end.

References

[1] D. Pham et al., "The design and implementation of a first-generation cell processor," in *In ISSCC Digest of Technical Papers*, 2005, pp. p. 184–5.

[2] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[3] Gilles Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. of the IFIP Congress 74*. 1974, North-Holland Publishing Co.

[4] Alexandru Turjan, "Compiling nested loop programs to process networks," 2007, PhD thesis, Leiden University, The Netherlands.

[5] Edwin Rijpkema, "Modeling Task Level Parallelism in Piece-wise Regular Programs," 2002, PhD thesis, Leiden University, The Netherlands.

[6] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere, "System Design using Kahn Process Networks: The Compaan/Laura Approach," in *Proc. Int. Conference Design, Automation and Test in Europe (DATE'04)*, Paris, France, Feb. 16-20 2004, pp. 340–345.

[7] E. A. de Kock, "Multiprocessor mapping of process networks: a JPEG decoding case study," in *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, New York, NY, USA, 2002, pp. 68–73, ACM Press.

[8] Sjoerd Meijer, Bart Kienhuis, Alex Turjan, and Erwin de Kock, "A process splitting transformation for kahn process networks," in *DATE*, Nice, France, 2007.

[9] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere, "Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures," in *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000.

[10] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere, "LAURA: Leiden Architecture Research and Exploration Tool," in *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, Sept. 1-3 2003.

[11] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov, "Improved Derivation of Process Networks," in *4th Workshop on Optimization for DSP and Embedded Systems, ODES-4*, New York, USA, Mar. 2006.

[12] Hristo Nikolov, Todor Stefanov, and Ed Deprettere, “Multi-processor system design with ESPAM,” in *CODES+ISSS ’06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, New York, NY, USA, 2006, pp. 211–216, ACM Press.

[13] Christian Haubelt, Joachim Falk, Joachim Keinert, Thomas Schlichter, Martin Streubühr, Andreas Deyhle, Andreas Hadert, and Jürgen Teich, “A systemc-based design methodology for digital signal processing systems,” *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 15–15, 2007.

[14] S. Pop, G.-A. Silber, A. Cohen, C. Bastoul, S. Girbal, and N. Vasilache, “GRAPHITE: Polyhedral analyses and optimizations for GCC,” Tech. Rep. A/378/CRI, Centre de Recherche en Informatique, École des Mines de Paris, Fontainebleau, France, 2006.

[15] Sjoerd Meijer, Bart Kienhuis, Johan Walters, and David Snuijf, “Automatic partitioning and mapping of stream-based applications onto the intel ixp network processor,” in *SCOPES ’07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, New York, NY, USA, 2007, pp. 23–30, ACM.

[16] Martin Alt, Uwe Asmann, and Hans van Someren, “Cosy compiler phase embedding with the cosy compiler model,” in *Computational Complexity*, 1994, pp. 278–293.

[17] Paul Feautrier, “Dataflow Analysis of Scalar and Array References,” *Int. Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[18] http://gcc.gnu.org/onlinedocs/gccint, ,” .

[19] Todor Stefanov, Bart Kienhuis, and Ed Deprettere, “Algorithmic transformation techniques for efficient exploration of alternative application instances,” in *CODES ’02: Proceedings of the tenth international symposium on Hardware/software codesign*, New York, NY, USA, 2002, pp. 7–12, ACM Press.

[20] F. Mueller, “Pthreads Library Interface,” 1993, Technical report, Florida State University, July 1993.