

Parallelization using Polyhedral Analysis

White Paper

ACE Associated Compiler Experts by

Version: 2008.3
Date: March 10, 2008
Status: release
Confidentiality: public
Reference: cosy-8153-polyhedral

Parallelization using Polyhedral Analysis

by ACE Associated Compiler Experts bv.

© Copyright 2008 by ACE Associated Compiler Experts bv,
Amsterdam, the Netherlands.

© Copyright 2008 by ACE Associated Computer Experts bv,
Amsterdam, the Netherlands.

All rights reserved. No part of this document may be copied, photocopied, reproduced or translated in any way, without prior written consent of ACE Associated Compiler Experts bv.

Every care has been taken in manufacturing the supplied product and its documentation. ACE Associated Compiler Experts bv will neither assume responsibility for any damages caused by the use of its products, nor accept warranty or update claims, unless stated explicitly otherwise in a special agreement.

The information contained in this document is subject to change without notice.

Printed in the Netherlands, March 10, 2008.

Many of the designations used by manufacturers and vendors to distinguish their product are trademarks. ACE Associated Compiler Experts bv has made every attempt to supply trademark information of manufacturers and their products mentioned in this document. ACE Associated Compiler Experts bv also recognizes any trademarks used in this document but not mentioned below.

Trademark notices

CoSy[®] is a registered trademark of ACE Associated Computer Experts bv.

Chapter 1

Introduction

Multicore architectures are being introduced more and more to meet the compute power required by applications. An example of such a platform is IBM's Cell processor. 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. 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. Because the computational model of an imperative language does not match that of parallel architectures, automatically mapping an application to a multicore architecture is not commonplace. Imperative languages use the concept of a single thread of control and a single memory space. Multiprocessor architectures are built from autonomous processes and distributed memories. The automatic parallelization and the automatic mapping of sequential applications to multi-core architectures is an important research topic for which a general compiler-oriented solution does not yet exist.

However, there are compilation techniques and analyses that greatly help application engineers in solving this difficult problem. This paper discusses the Polyhedral and Kahn Process Network models that are being integrated into the CoSy compiler framework.

This paper describes our experiments in semi-automatic coarse-grain parallelization of an MJPEG application and its mapping to the CELL architecture. This experiment is described in Chapter 5. That chapter is self-contained so impatient readers can jump to it immediately. For background information, Chapter 2 contains a small overview of CoSy, Chapter 3 deals with the polyhedral model that is used to calculate exact data dependences and to create a Polyhedral Reduced Dependence Graph (PRDG). In Chapter 4, the automatic derivation of the Kahn Process Network (KPN) from a PRDG is presented. Furthermore, the extensions to CoSy's Intermediate Representation (IR) to represent the components of the KPN model to facilitate network restructuring transformations are discussed.

Chapter 2

CoSy

The CoSy compiler development system is a well-established commercial compiler building framework [2]. It is used to build compilers for a wide variety of architectures ranging from simple RISC processors to highly complex non-interlocked VLIW processors. CoSy is flexible, modular, highly retargetable and generates high performance compilers.

CoSy is a modular compiler framework built around a single Intermediate Representation (IR). A large number of so-called *engines* is provided with CoSy. The engines take care of analysis, transformation and optimization of the IR. CoSy users can build their own engines to extend or specialize CoSy functionality. CoSy includes a number of generators that take care of some key components in compiler construction. These include generators for the code generator, the supervisor that controls engine ordering and the IR access library. All the work in this paper was done using standard CoSy interfaces. CoSy's IR is not only fully specified and accessible to the compiler writer, it is also extensible. For the work in this paper the CoSy IR is extended with representations for PRDGs and KPNs as described next.

Chapter 3

Polyhedral Model

The polyhedral model is a model to represent and manipulate loop nest structures and their program statements. In this model, the iteration space of a program statement is represented by a single geometrical object—a polyhedron. Data dependence analysis and loop restructuring transformations such as loop fusion, loop fission and strip-mining can be efficiently implemented using polyhedral loop models. Polyhedral models can be manipulated efficiently with existing tools such as PolyLib, the Parma Polyhedral Library, and Cloog [1, 4, 3, 6]. A polyhedral model can be derived from loop nests with the following properties:

- loops have a constant step size, and constant or affine loop bounds (linear combination of loop iterators)
- if-statements must have affine conditions
- index expressions of array references are affine constructs of the enclosing loop iterators, program parameters, and constants
- data flow between statements in the loop must be explicit. This prohibits that two statements that contain function calls communicate through shared variables invisible to the compiler.

Program parts that have these properties are called *static control parts*. Although the polyhedral model does impose restrictions on the input program, in many application domains it is natural to express time critical parts of the applications in this form. Examples are DSP and audio/video stream-based applications in consumer electronics, and also modeling and simulation applications in high performance computing. Therefore, the polyhedral model is highly relevant because it enables efficient loop restructuring in many applications.

3.1 Parametric Integer Linear Programming

In the polyhedral model, an iteration vector is associated with each program statement. The length of the vector is equal to the number of loops that enclose the statement. The i_{th}

component of the vector corresponds to the value of a loop iterator at depth i . The *iteration space* of a statement corresponds to all values of the iteration vector associated with the statement.

Thus, the iteration space of a statement is described by a set of linear inequalities defining a polyhedron in an n -dimensional space, where n corresponds to the length of the iteration vector. The bounds of the polyhedron are defined by the lower and upper bounds of the loop counters.

In effect, the polyhedral model of the iteration space of a statement is just a set of linear equations. Because of this it is possible to use Linear Programming techniques to compute dependences between statements. Parametric Integer Linear Programming (PILP) is used to find the lexicographic maximum integer point of the iteration space of the producer statement in a dependence relation, as described in [10] and [11]. This is illustrated with the following running example. Suppose there are two statements that read and write to the same array in two subsequent loops:

```

for (i=1; i<=9; i++)
    a[i] = i;    // S1 writes to a[i]
for (i=1; i<=9; i++)
    F(a[i-1]);  // S2 reads from a[i-1]

```

Figure 3.1: Producer Statement $S1$, Consumer Statement $S2$

To calculate the data dependence between the statements requires the construction of a constraint system that can be given to the PILP solver. The constraint system for the example is depicted in Table 3.1.

Objective	$max_{lex}\{i_p\}$
Subject to	$1 \leq i_p \leq 9$ $1 \leq i_c \leq 9$ $(i_p) \prec (i_c)$ $(i_p) = (i_c - 1)$

Table 3.1: Constraint system for calculating dependences

The objective function describes the dependence relation—it must find the last iteration in the producer’s iteration space that wrote to a specific memory location. This is equivalent to finding the lexicographically maximum iteration point of the producer statement. The objective is subject to the constraints that come from the polyhedral models of the statements: the iterators of the producer and consumer statements range from 1 to 9 (denoted by $1 \leq i_p, i_c \leq 9$); data production must occur before data consumption which is translated to $(i_p) \prec (i_c)$; and finally, the subscript index expressions of both statements must address the same location: $i_p = i_c - 1$.

At this point, polyhedral models are generated in CoSy for loop nests and an existing PILP solver is used that allows to solve constraint systems as depicted in Table 3.1. Solving these

results in a data dependency functions (DD) that map an iteration point from the consumer iteration space to a point of the producer where the data is produced. In the example, data is produced at each previous iteration of i such that: $DD(i_c) = (i_c - 1)$.

3.2 Polyhedral Reduced Dependence Graph

Two statements are said to be dependent if they address the same memory location, the first statement occurs before the second, and at least one of them writes to that memory location. These dependences form a partial order on the program statements that should be respected in statement reordering transformations and by parallelization algorithms. In a reduced dependence graph there is a vertex for every statement and an edge between dependent statements (vertices). The dependence graph is reduced because a single vertex represents all iterations of the statement and a single edge represents any number of iteration points where the statements are dependent. No precise information is kept on which iteration points the dependence occurs.

In a Polyhedral Reduced Dependence Graph (PRDG), every edge is additionally labeled with a dependence polyhedron P . These dependence polyhedra are the result of calculating exact data dependences of scalar and array references as described in Section 3.1. A CoSy compiler engine was developed that: 1) scans the input code for static control parts, 2) calculates the dependences, and 3) creates the PRDG. The engine depends on the standard CoSy engine `loopanalysis` to find loops and attribute their properties.

For the code fragment of the producer statement $S1$ and consumer statement $S2$ from Section 3.1, the complete dependence graph for the full iteration space is shown in Figure 3.2.

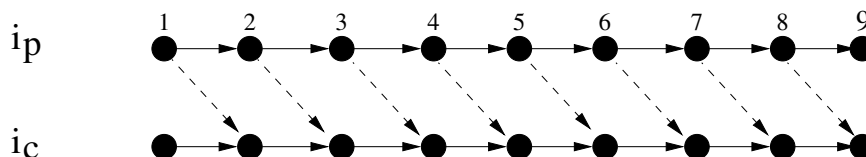


Figure 3.2: The full dependence graph for producer-consumer pair ($S1, S2$)

Figure 3.2 depicts the one-dimensional iteration spaces of statements $S1$ and $S2$. The solid arrow denotes the execution order and the dotted arrow the dependences between points in the iteration spaces of $S1$ and $S2$. As the full dependence graph can get very large, the PRDG is used to accurately describe the dependences in a compact notation. The PRDG for the example consists of two nodes corresponding to the program statements $S1$ and $S2$, and one edge labeled with a dependence polyhedron. The dependence polyhedron consists of all consumer iterations except the first one where $i_c = 1$. The PRDG is shown in Figure 3.3.

The iteration domains of nodes $S1$ and $S2$ are represented by polyhedra as well—the constraints come from their surrounding loop bounds and thus $S1$ has a polyhedron of the form $\{i \mid 1 \leq i \leq 9\}$.

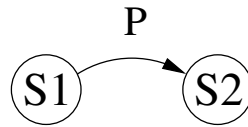


Figure 3.3: The PRDG for producer-consumer pair $(S1, S2)$

3.3 Loop Parallelization Algorithms

Loop parallelization algorithms can be characterized by the different RDGs they analyze. Edges labeled with dependence levels are input for Allen and Kennedy's algorithm which mark loops as DOALL or DOSEQ [5]. This is used in generating vector code. Wolf and Lam's algorithm can be used for detecting parallelism in RDGs where edges are labeled with direction vectors [17]. RDGs and edges can also be labeled with dependence polyhedra. In this case, Darte and Vivien's algorithm and Feautrier's algorithm can be used to detect the degree of parallelism in applications [8, 7]. Two reduced dependence graphs are available in CoSy: one RDG with dependence levels (a classical representation), and one with dependence polyhedra (the PRDG). These RDGs allow implementation of above mentioned parallelization algorithms, or any other parallelization algorithm that requires dependency information. In [14, 16], for example, it is shown that a Kahn Process Network (KPN) can automatically be derived from a PRDG. KPN is a model of computation consisting of autonomous processes that communicate over FIFO channels. Next sections will deal with this KPN model and our ongoing effort to integrate it into the CoSy framework. This allows the translation, in a fully automatic and analytical way, of sequential applications to a parallel program specification.

Chapter 4

Kahn Process Networks

The KPN model expresses an applications in terms of autonomous processes that communicate with each other using unbounded FIFOs [12]. 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, the process unblocks and reads the data from the FIFO channel and continues processing.

KPNs are very suitable for systematic mapping onto multi-core architectures [15]. Since a KPN can be derived from a PRDG in a fully analytical way, it is very attractive to use it for code generation and mapping of applications onto multi-core platforms.

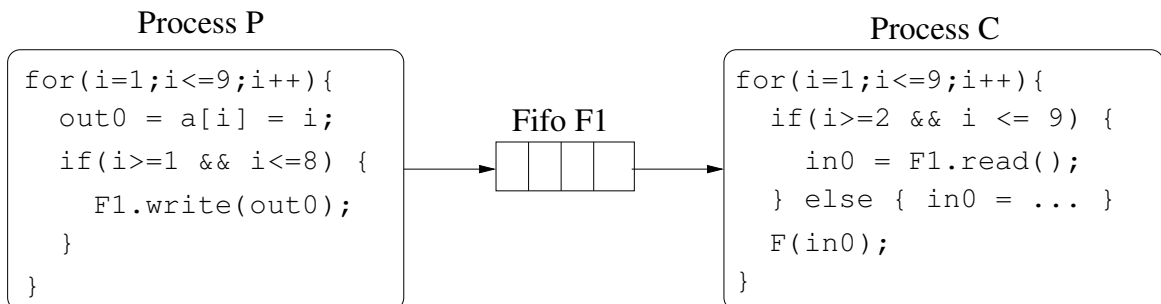


Figure 4.1: KPN for statement $S1$ and $S2$

The KPN for the producer-consumer example is given in Figure 4.1. It consist of two processes P , C and one FIFO channel. Process P corresponds to statement $S1$ in the program code and C to statement $S2$.

The internal structure of a process consist of a nested loop, input and output port statements and the statement or function call that defines this process. The nested loop corresponds to the loop (and if-then-else) structure that surrounds the statement in the original program. The input and output ports correspond to the exact solution of the dependence relations between statements as defined in the PRDG. Typically, an input (output) port is read only

for a subset of the iteration space. This is achieved by enclosing the input (output) port execution by if-statements that match the subset.

For the producer-consumer example, the output port of the consumer process corresponds to the result of the dataflow analysis as discussed in Section 3.2. As can be seen from Figure 4.1 data is produced at each iteration except for the last iteration point. Thus, the output port for the producer is defined only for iterations 0 to 8, as can be seen for the if-statement surrounding the write.

4.1 KPN IR Annotation

Automatic derivation of KPNs from sequential code is a first step in the parallelization process. KPN restructuring transformations can be performed to further improve performance results. Examples of these transformations are given in [13]. In that article, a JPEG application specified as a KPN was executed on a multicore architecture but 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 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. Extending the GCC compiler with a KPN model was explored but not successful. To support these transformation in a compiler, a high-level model is required that expresses the FIFOs, processes and network structure of a KPN.

To enable these type of high-level transformations, the KPN model is integrated into CoSy. CoSy's IR is extended with representations of the different components of a KPN. Thanks to CoSy's modular engineering it was possible to do this without effecting the overall flow of the compiler. This provides us the framework for: 1) code generation for multicore platforms, 2) high-level restructuring transformations such as merging and splitting nodes in order to decrease or increase parallelism respectively.

The Kahn process network is modeled as a graph of processes and FIFO channels. Using the IR description language called *fSDL*, the process network structure is modeled and new data structures are introduced for the processes and FIFOs. For example, a process is described by type `mirKPNprocess` and a FIFO channel is described by type `mirKPNfifo`. The graph is described by type `mirKPN`. The IR definitions of `mirKPN`, `mirKPNprocess` and `mirKPNfifo` are as follows:

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

```
    >
  } ;

  domain mirKPNprocess: {
    mirKPNprocess <
      Process : mirProcGlobal,
    >
  } ;

  domain mirKPNfifo: {
    mirKPNfifo <
      fifoid: INT
    >
  } ;
```

Based on these two IR specifications, functionality is generated for traversing and manipulating the data structures. For example, `GRAPH_KPN_VERTEX_LOOP` and `GRAPH_KPN_EDGE_LOOP` are generated macros to visit all processes and FIFOs respectively. For the `mirKPNprocess` type, the `Process` field contains a pointer to a `mirProcGlobal`. This data structure is used by CoSy to represent procedures and functions in the default IR. Instances of the `mirKPNprocess` data structure are populated when the KPN is created. Since `mirProcGlobal` is just a procedure definition in CoSy, the normal compiler-flow is kept. As a consequence, the KPN view does not interfere with the default IR and therefore all available CoSy engines can operate without changes or validation. Using the `mirKPN`, `mirKPNprocess` and `mirKPNfifo` data types, it becomes straightforward to make copies of processes, create new FIFOs, and thus restructure the KPN. These transformations are now being automated.

Chapter 5

An Experiment with a Real Application

The implementation is currently capable of automatically mapping static control parts to a coarse grain parallel architecture. In the experiment described here, there are still some manual steps needed in particular for the tuning of the grain size.

In this experiment, the MJPEG encoder is mapped onto the CELL architecture. MJPEG is a relatively simple video encoding technique that applies JPEG encoding to each frame. The CELL architecture is a multi-core architecture with a single PowerPC (PPE) and 8 stream-oriented processors (SPEs) that are connected by a fast ring network. The large PowerPC side memory allows shared access from the SPEs, but the bandwidth to do this is limited and it has a high latency. Every SPE also has a fast local memory of 256 Kb. It is not straightforward to efficiently map applications on the CELL for parallel processing. Ideally, such applications make optimal use of local processing on the SPEs and communicate through the fast ring network.

For the polyhedral analysis to work, the MJPEG application has to be put into the form as described in Section 3. Compiler transformations can also help to do this so as to allow more freedom in the input form of the program.

To give an idea of the data flow of the program, the next fragment shows a part of the simplified C-source code. It was simplified for this presentation by reducing the data structures passed around. Specifically, the luminance, chrominance and Huffman tables used in the variable length encoder (VLE) are omitted. For the actual experiment, the full MJPEG application was used.

```

for (j=0; j<IMAX; j++) {
  for (i=0; i<JMAX; i++) {
    DCT( block[j][i],
        &(block[j][i]) );
  }
}
for (j=0; j<IMAX; j++) {
  for (i=0; i<JMAX; i++) {
    Quant( block[j][i],
          &(block[j][i]) );

    VLE( block[j][i], varEndOfField,
        &Statistics, &packets );

    ControlF1( Ctrl, Statistics,
              &varEndOfField, &Ctrl );

    VidOut( packets );
  }
}

```

The polyhedral framework requires that each of the functions `DCT`, `Quant`, `VLE`, `ControlF1` and `VidOut` do not communicate data in any other way than is apparent from the fragment above. This property can be analyzed by the compiler although it relies, for more complicated functions, on program wide alias analysis.

The KPN associated with the simplified MJPEG application is shown in Figure 5.1.

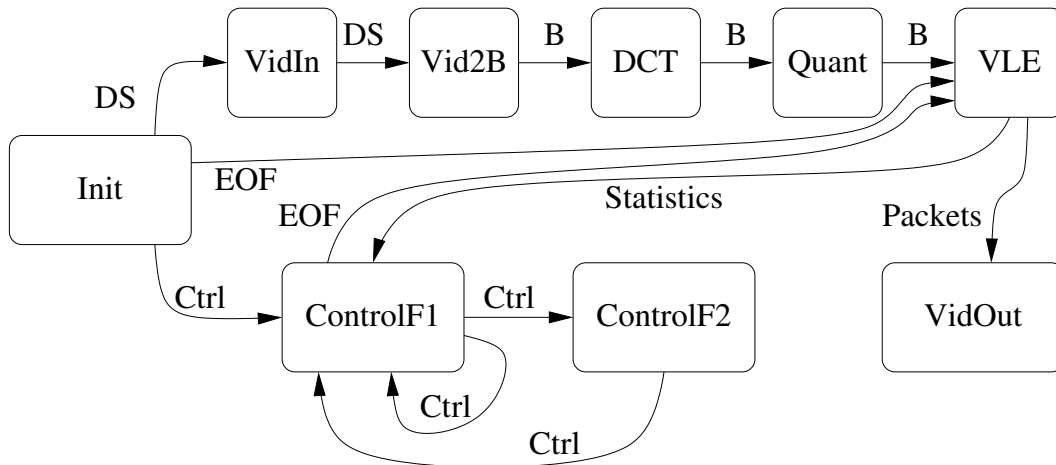


Figure 5.1: KPN for simplified MJPEG. The loop shown above corresponds to the blocks `DCT`, `Quant`, `VLE`, `ControlF1` and `VidOut`

In the figure, `DS` is the input data stream, `B` are 8x8 image data blocks, `EOF` are End of

Field markers and `Ctrl` is a control variable.

The computation-intensive parts are in the blocks for the DCT, Quantization and Variable Length Encoding. The polyhedral analysis has found that the data flow between these blocks is very simple—it is a pipeline of transformations on 8x8 blocks of image data. This demonstrates the crucial step that is enabled by polyhedral analysis. In the original C source program, these three blocks operate sequentially on the same piece of image data stored at one place in memory: `block[j][i]`. The KPN shows that the image data can be passed by-value between the computational blocks. This liberates the computational blocks from their bounds to specific shared memory locations.

The power of polyhedral analysis is also shown in the KPN graph around the `ControlF1` block. This block updates the shared variable `Ctrl`. This shared variable is initially written by the `Init` block that runs once for the entire application. The variable is also updated by the `ControlF2` block that runs once per image frame. `ControlF2` runs once per 8x8 image data block. Due to the exact nature of the polyhedral analysis, the edges in the KPN contain precise information on when, i.e. at which iteration, the value of the `Ctrl` variable is passed between the nodes in the graph. This makes the KPN fully deterministic.

5.1 Mapping and KPN Transformations

The power of the KPN is that it makes the program dataflow completely explicit and lifts the data from its shared memory location. However, this is not yet enough to get an efficient parallel implementation. The mapping to an actual parallel architecture still needs to be done.

In the simplified MJPEG KPN, the nodes `VLE` and `ControlF1` are closely connected. Every execution of the `VLE` node creates one output on the `Statistics` edge. This value is consumed by `ControlF1` and leads to an update of the `EOF` value, which is in turn consumed by `VLE`. Given this dataflow pattern, there is no parallelism to be gained in executing `VLE` and `ControlF1` on separate processors—each will be waiting while the other one runs. Therefore, the KPN is more efficiently implemented by merging `VLE` and `ControlF1` into a single node. This compiler transformation is like function inlining—when put together, the code can be more efficiently compiled. At the level of the KPN, it requires re-routing of the data flow and correction of the iteration spaces associated with the edges. This is possible due to the deterministic nature of the KPN.

A second transformation is that of node splitting, which is also called node unrolling in [9]. Consider that in MJPEG, the DCT node is more computationally expensive than the Quantization and the VLE. Given that the DCT node is purely functional, i.e. its result only depends on its input image block, it makes sense to run more than one invocation of DCT in parallel. This is possible using a node splitting transformation as demonstrated in Figure 5.2.

Such splitting transformations can in principle be generalized to functional sub-graphs of the KPN too. The complicated part of the splitting transformation is to adjust the iteration spaces of the edges along which the image data blocks are sent. Because of the deterministic nature of the KPN, all “even” image data will be sent to the top block and all “odd” image

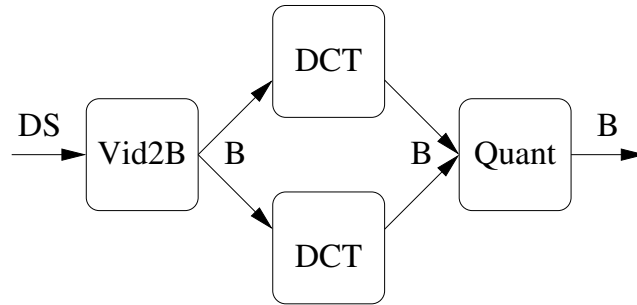


Figure 5.2: Splitting the DCT node

data will be sent to the bottom block. Note that if the run-time of the DCT nodes would vary depending on the image data, it makes sense to allow a run-time choice of the DCT processor. This cannot be expressed in the purely deterministic KPN formalism.

In order to apply the splitting transformation, the compiler must ascertain that the node has purely functional behavior. As described above, it is possible for the compiler to classify nodes. While it is not possible for nodes to communicate under-the-hood, polyhedral analysis does allow nodes having internal state. This is the case with the `VidOut` node in the simplified KPN. It acts as a sink for packets and somehow processes them further. How this is done is not visible in the KPN graph. All inputs to this node must be processed in exactly the same order as in the original sequential program such that the internal state updates are also done in the same order. It prohibits splitting of the `VidOut` node.

5.2 Results

When taking the full MJPEG source code, it translates to a KPN with 58 edges and 9 nodes. Mapping this directly to the CELL architecture does not result in any speedup because of the additional communication overhead. In fact, the program runs 4 times slower than the original running on the PPU due to the overhead of the 58 FIFOs. Merging transformations are necessary to get rid of this overhead. To improve this, the code was manually modified to reduce the number of nodes in the KPN. First, all nodes are merged into a single node. Then, multiple DCT processes were offloaded to the SPEs by applying the splitting transformation as explained in the previous section.

In the experiment, the main thread was running on the PPU, communicating with the DCT processes and SPEs through software implemented ring buffers for the FIFOs. With one thread on the PPU, and two DCTs on the SPEs, a 21% improvement in execution time was achieved compared to the sequential application running on the PPU. Although the speedup is significant, it is not linear to the 2 extra CPUs that are used. However, two remarks have to be made. First, the software FIFO library is a basic implementation that should be optimized to minimize communication costs. Secondly, in the experiments the main thread was both the source and sink for the offloaded tasks. As a result, the main thread was stalled while waiting for data. Therefore, the source and sink should be two threads in order to have,

in a pipelined fashion, a better streamed and performance efficient application. Due to the manual transformation steps involved, only a few different instances of the KPN were tried. However, the potential of this approach is demonstrated and there are many opportunities to further improve on the performance results.

Chapter 6

The Road Ahead

This paper demonstrates a number of extensions to the CoSy framework that are stepping stones towards a parallelization framework for sequential C programs. This framework can solve the problem of automatic parallelization for some programs. Additionally, a much larger range of programs can be effectively parallelized using additional input from the application developer. To this effect, the KPN formalism as implemented here provides an important interface in the mapping to the target architecture. A KPN interface, possibly attributed with profiling results, is an intuitive model that can be presented to the application developer for review. At the same time, KPN is a suitable formalism to transform the application such that it maps better to the target architecture.

The exciting first step in the process is the integration in CoSy of dependence analysis based on the polyhedral representation of loop nests. Unlike traditional dependence analysis, which represents conservative dependence assumptions, polyhedral based analysis enables exact dependence representation and, hence, exact knowledge of the data flow in the loop nest. The resulting liberation of the data from its memory location is the crucial step that allows great freedom in the mapping of the performance critical parts of the application to multi-core architectures.

Our vision is to provide an integrated framework in CoSy that allows the discovery, representation, manipulation and mapping of parallelism from sequential programs. Key components of this framework were demonstrated in this paper.

About ACE

ACE Associated Compiler Experts bv (a wholly owned subsidiary of ACE Associated Computer Experts bv, Amsterdam, the Netherlands) is a world leader in tools and services for professional compiler development. Its open CoSy compiler development system gives compiler developers the ability to achieve a leading edge position in the construction of better and faster optimizing compilers for architectures ranging from 4-bit micro-controllers to 24 bit DSPs, 256-bit VLIW processors and multicore processor systems. CoSy accommodates a wide range of programming languages including C, Embedded C, DSP C, C++, Fortran and Java. SuperTest is the most comprehensive test and validation suite for C/C++ compilers. Based upon 30+ years of ACE experience in compiler construction and validation, SuperTest provides a unique level of compiler test coverage. More information on ACE and its products and services is available at www.ace.nl.

Bibliography

- [1] <http://icps.u-strasbg.fr/polylib/>.
- [2] <http://www.ace.nl/>.
- [3] <http://www.cloog.org/>.
- [4] <http://www.cs.unipr.it/ppl/>.
- [5] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.
- [6] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.
- [7] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Parallel Comput.*, 24(3-4):421–444, 1998.
- [8] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–496, 1997.
- [9] Erwin de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. 15th Int. Symposium on System Synthesis (ISSS'2002)*, pages 68–73, Kyoto, Japan, October 2-4 2002.
- [10] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.
- [11] Paul Feautrier. Dataflow Analysis of Scalar and Array References. *Int. Journal of Parallel Programming*, 20(1):23–53, 1991.
- [12] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

- [13] Sjoerd Meijer, Bart Kienhuis, Alex Turjan, and Erwin de Kock. A process splitting transformation for Kahn Process Networks. In *DATE*, Nice, France, 2007.
- [14] Edwin Rijpkema. Modeling Task Level Parallelism in Piece-wise Regular Programs, 2002. PhD thesis, Leiden University, The Netherlands.
- [15] 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*, pages 7–12, New York, NY, USA, 2002. ACM Press.
- [16] Alexandru Turjan. Compiling nested loop programs to process networks, 2007. PhD thesis, Leiden University, The Netherlands.
- [17] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.