Chapter 8

Mapping Applications to a Coarse Grain Reconfigurable System

Yuanqing Guo, Gerard J.M. Smit, Michèl A.J. Rosien, Paul M. Heysters, Thijs Krol, Hajo Broersma

University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, P.O. Box 217, 7500AE Enschede, The Netherlands
{yguo, smit, rosien, heysters, krol}@cs.utwente.nl; h.j.broersma@utwente.nl

Abstract This paper introduces a design method to map applications written in a high level source language program, like C, to a coarse grain reconfigurable architecture, called MONTIUM. The source code is first translated into a control dataflow graph (CDFG). Then after applying graph clustering, scheduling and allocation on this CDFG, it can be mapped onto the target architecture. High performance and low power consumption are achieved by exploiting maximum parallelism and locality of reference respectively. Using our mapping method, the flexibility of the MONTIUM architecture can be exploited.

Keywords: Coarse Grain Reconfigurable Architecture, Mapping and Scheduling

8.1 Introduction

In the CHAMELEON/GECKO project we are designing a heterogeneous reconfigurable System-On-Chip (SoC) [Smit, 2002] for 3G/4G terminals. This SoC contains a general-purpose processor (ARM core), a bit-level reconfigurable part (FPGA) and several word-level reconfigurable parts (MONTIUM tiles). We believe that in future 3G/4G terminals heterogeneous reconfigurable architectures are needed. The main reason is that the efficiency (in terms of performance or energy) of the system can be improved significantly by mapping application tasks (or kernels) onto the most suitable processing entity.

The objective of this paper is to present a design method for mapping processes, written in a high level language, to a reconfigurable platform. The

P. Lysaght and W. Rosenstiel (eds.),
New Algorithms, Architectures and Applications for Reconfigurable Computing, 93–103.
© 2005 *Springer. Printed in the Netherlands.*

method can be used to optimize the system with respect to certain criteria e.g. energy efficiency or execution speed.

8.2 The Target Architecture: MONTIUM

In this section we give a brief overview of the MONTIUM architecture, because this architecture led to the research questions and the algorithms presented in this paper. Figure 8.1 depicts a single MONTIUM processor tile. The hardware organization within a tile is very regular and resembles a very long instruction word (VLIW) architecture. The five identical arithmetic and logic units (ALU1 $\cdots$ ALU5) in a tile can exploit spatial concurrency to enhance performance. This parallelism demands a very high memory bandwidth, which is obtained by having 10 local memories (M01 $\cdots$ M10) in parallel. The small local memories are also motivated by the locality of reference principle. The ALU input registers provide an even more local level of storage. Locality of reference is one of the guiding principles applied to obtain energy-efficiency in the MONTIUM. The MONTIUM has a datapath width of 16-bits and supports both integer and fixed-point arithmetic. Each local SRAM is 16-bit wide and has a depth of 512 positions, which adds up to a storage capacity of 8 Kbit per local memory. A memory has only a single address port that is used for both reading and writing. A reconfigurable address generation unit (AGU) accompanies each memory. The AGU contains an address register that can be modified using base and modify registers.

A single ALU has four 16-bit inputs. Each input has a private input register file that can store up to four operands. The input register file cannot be bypassed, i.e., an operand is always read from an input register. Input registers can be written by various sources via a flexible interconnect. An ALU has two 16-bit outputs, which are connected to the interconnect. The ALU is entirely combinatorial and consequentially there are no pipeline registers within the ALU. Each MONTIUM ALU contains two different levels. Level 1 contains four

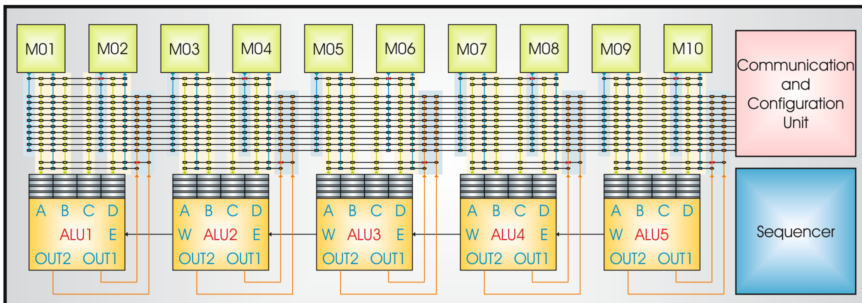

Figure 8.1. MONTIUM processor tile.

function units. A function unit implements the general arithmetic and logic operations that are available in languages like C (except multiplication and division). Level 2 contains the MAC unit and is optimised for algorithms such as FFT and FIR. Levels can be bypassed (in software) when they are not needed.

Neighboring ALUs can also communicate directly on level 2. The West-output of an ALU connects to the East-input of the ALU neighboring on the left (the West-output of the leftmost ALU is not connected and the East-input of the rightmost ALU is always zero). The 32-bit wide East-West connection makes it possible to accumulate the MAC result of the right neighbor to the multiplier result (note that this is also a MAC operation). This is particularly useful for performing a complex multiplication, or for adding up a large amount of numbers (up to 20 in one clock cycle). The East-West connection does not introduce pipeline, as it is not registered.

8.3 A Four-Phase Decomposition

The overall aim of our research is to execute DSP programs written in high level language, such as C, by one MONTIUM tile in as few clock cycles as possible. There are many related aspects: the limitation of resources; the size of total configuration space; the ALU structure etc. The main question is to find an optimal solution under all those constraints. Therefore we propose to decompose this problem into a number of phases: Based on the two-phased decomposition of multiprocessor scheduling introduced by [Sarkar, 1989], our work is built on a four-phase decomposition: translation, clustering, scheduling and resource allocation:

1 **Translating the source code to a CDFG**: The input C program is first translated into a CDFG; and then some transformations and simplifications are done on the CDFG. The focus of this phase is the input program and is largely independent of the target architecture.

2 **Task clustering and ALU data-path mapping**, clustering for short: The CDFG is partitioned into clusters and mapped to an unbounded number of fully connected ALUs. The ALU structure is the main concern of this phase and we do not take the inter-ALU communication into consideration;

3 **Scheduling**: The graph obtained from the clustering phase is scheduled taking the maximum number of ALUs (it is 5 in our case) into account. The algorithm tries to find the minimize number of the distinct configurations of ALUs of a tile;

4 **Resource allocation**, allocation for short: The scheduled graph is mapped to the resources where locality of reference is exploited, which is

important for performance and energy reasons. The main challenge in this phase is the limitation of the size of register banks and memories, the number of buses of the crossbar and the number of reading and writing ports of memories and register banks.

Note that when one phase does not give a solution, we have to fall back to a previous phase and select another solution.

8.4 Translating C to a CDFG

A control data flow graph (CDFG) $G = (N_G, P_G, A_G)$ consists of two finite non-empty sets of **nodes** N_G and **ports** P_G and a set A_G of so-called **hydra-arcs**; a hydra-arc $a = (t_a, H_a)$ has one **tail** $t_a \in N_G \cup P_G$ and a non-empty set of **heads** $H_a \subset N_G \cup P_G$. In our applications, N_G represents the operations of a CDFG, P_G represents the inputs and outputs of the CDFG, while the hydra-arc (t_a, H_a) either reflects that an input is used by an operation (if $t_a \in P_G$), or that an output of the operation represented by $t_a \in N_G$ is input of the operations represented by H_a, or that this output is just an output of the CDFG (if H_a contains a port of P_G).

See the example in Figure 8.2: The operation of each node is a basic computation such as addition (in this case), multiplication, or subtraction. Hydra-arcs are directed from their tail to their heads. Because an operand might be input for more than one operation, a hydra-arc is allowed to have multiple heads although it always has only one tail. The hydra-arc $e7$ in Figure 8.2, for instance, has two heads, w and v. The CDFG communicates with external systems through its ports represented by small grey circles in Figure 8.2.

In general, CDFGs are not acyclic. In the first phase we decompose the general CDFG into acyclic blocks and cyclic control information. The control

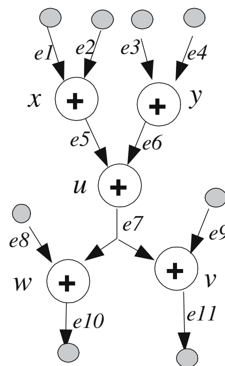

Figure 8.2. A small CDFG.

```
const int n = 4;
float x_re[n] ; float x_im[n] ; float w_re[n/2]; float w_im[n/2];
void main() {
 int xi, xip;
 float u_re, u_im, x_re_tmp_xi, x_re_tmp_xip, x_im_tmp_xi, x_im_tmp_xip;
 for (int le = n / 2; le > 0; le /= 2) {
  for (int j = 0; j < le; j++) {
   int step = n / le;
   for (int i = 0; i < step/2; i++ ) {
    xi = i + j * step; xip = xi + step/2; u_re = w_re[le * i]; u_im = w_im[le * i];
    x_re_tmp_xi = x_re[xi]; x_re_tmp_xip = x_re[xip];
    x_im_tmp_xi = x_im[xi]; x_im_tmp_xip = x_im[xip];
    x_re[xi] = x_re_tmp_xi + (u_re *  x_re_tmp_xip - u_im *  x_im_tmp_xip);
    x_re[xip] = x_re_tmp_xi - (u_re *x_re_tmp_xip - u_im * x_im_tmp_xip);
    x_im[xi] = x_im_tmp_xi + (u_re * x_im_tmp_xip + u_im * x_re_tmp_xip);
    x_im[xip] = x_im_tmp_xi  - (u_re * x_im_tmp_xip+ u_im * x_re_tmp_xip);
   }
  }
 }
}
```

Figure 8.3. C code for the n-point FFT algorithm.

information part of the CDFG will be handled by the sequencer of the MONTIUM. In this paper we only consider acyclic parts of CDFGs. To illustrate our approach, we use an FFT algorithm. The Fourier transform algorithm transforms a signal from the time domain to the frequency domain. For digital signal processing, we are particularly interested in the discrete Fourier transform. The fast Fourier transform (FFT) can be used to calculate a DFT efficiently. The source C code of a n-point FFT algorithm is given in Figure 8.3 and Figure 8.4 shows the CDFG generated automatically from a piece of 4-point FFT code after C code translation, simplification and complete loop expansion. This example will be used throughout this paper.

8.5 Clustering

In the clustering phase the CDFG is partitioned and mapped to an unbounded number of fully connected ALUs, i.e., the inter-ALU communication is not yet considered. A cluster corresponds to a possible configuration of an ALU data-path, which is called **one-ALU configuration**. Each one-ALU configuration has fixed input and output ports, fixed function blocks and fixed control signals. A partition with one or more clusters that can not be mapped to our MONTIUM ALU data-path is a failed partition. For this reason the procedure of clustering should be combined with ALU data-path mapping. Goals of clustering are 1) minimization of the number of ALUs required; 2) minimization of the number of distinct ALU configurations; and 3) minimization of the length of the critical path of the dataflow graph.

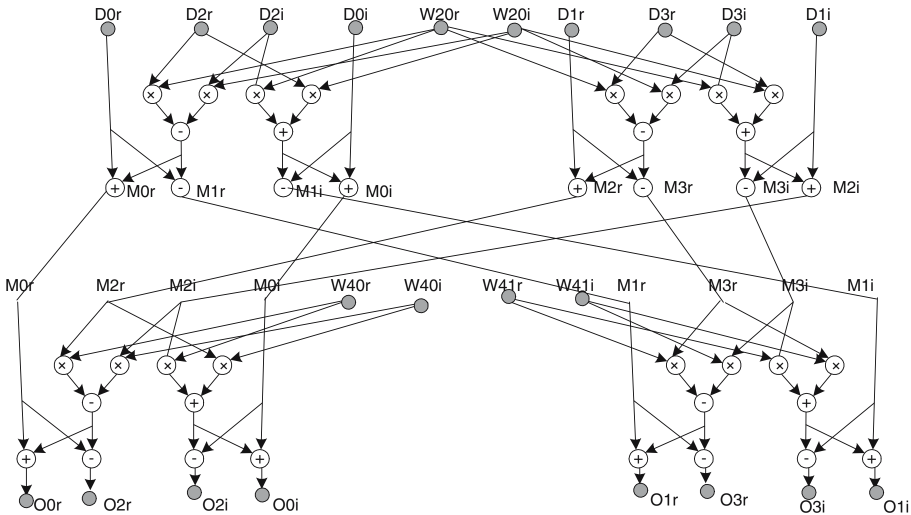

Figure 8.4. The generated CDFG of a 4-point FFT after complete loop unrolling and full simplification.

The clustering phase is implemented by a graph-covering algorithm [Guo, 2003]. The procedure of clustering is the procedure of finding a cover for a CDFG which is implemented in two steps:

Step A: Template Generation Problem

Given a CDFG, generate the complete set of nonisomorphic templates (that satisfy certain properties, e.g., which can be executed on the ALU-architecture in one clock cycle), and find all their corresponding matches (instances).

Step B: Template Selection Problem

Given a CDFG *G* and a set of (matches of) templates, find an 'optimal' cover of *G*.

See [Guo, 2003] for the details of the graph-covering algorithm.

Each selected match is a cluster that can be mapped onto one MONTIUM ALU and can be executed in one clock-cycle [Rosien, 2003]. As an example Figure 8.5 presents the produced cover for the 4-point FFT. The letters inside the dark circles indicate the templates. The graph is completely covered by three templates. This result is the same as our manual solution. It appears that the same templates are chosen for a n-point FFT ($n = 2^d$).

8.6 Scheduling

To facilitate the scheduling of clusters, all clusters get a **level** number. The level numbers are assigned to clusters with the following restrictions:

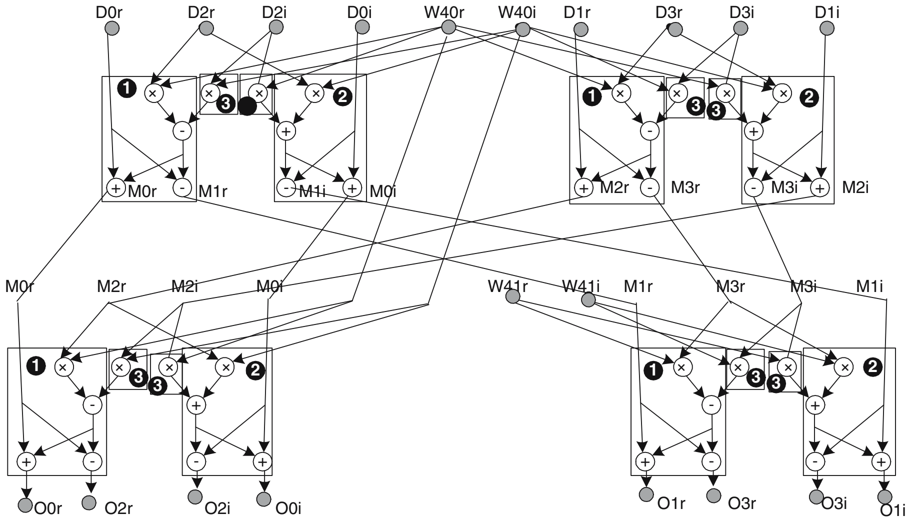

Figure 8.5. The selected cover for the CDFG in 8.4.

- For a cluster A that is dependent on a cluster B with level number i, cluster A must get a level number $> i$ if the two clusters cannot be connected by the west-east connection.
- Clusters that can be executed in parallel may have equal level numbers.
- Clusters that depend only on in-ports have level number one.

The objective of the clustering phase is to minimize the number of different configurations for separate ALUs, i.e. to minimize the number of different one-ALU configurations. The configurations for all five ALUs of one clock cycle form a **5-ALU configuration**. Since our MONTIUM tile is a very long instruction word (VLIW) processor, the number of distinct 5-ALU configurations should be minimized as well. At the same time, the maximum amount of parallelism is preferable within the restrictions of the target architecture. In our architecture, at most 5 clusters can be on the same level.

If there are more than 5 clusters at some level, one or more clusters should be moved one level down. Sometimes one or more extra clock cycles have to be inserted. Take Figure 8.5 as an example, where, in level one, the clusters of type 1 and type 2 are dependent on clusters of type 3. However, by using the east-west connection, clusters of type 1/2 and type 3 can be executed on the same level. Because there are too many clusters in level 1 and level 2 of Figure 8.5, we have to split them. Figure 8.6(a) shows a possible scheduling scheme where not all five ALUs are used. This scheme consists of only one 5-ALU

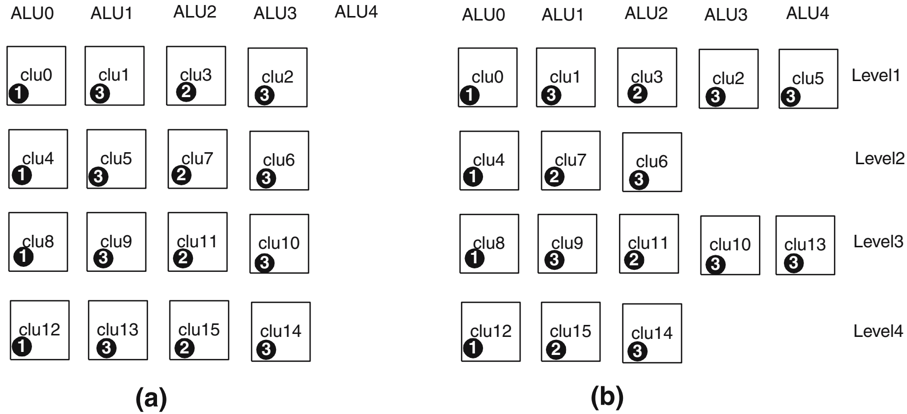

Figure 8.6. Schedule the ALUs of Figure 8.5.

configuration: $C1 = \{$❶❸❷❸○$\}$. As a result, with the scheme of 8.6(a), the configuration of ALUs stays the same during the execution. The scheduling scheme of Figure 8.6(b) consists of 4 levels as well, but it is not preferable because it needs two distinct 5-ALU configurations: $C2 = \{$❶❸❷❸❸$\}$ and $C3 = \{$❶❷❸○○$\}$. Switching configurations adds to the energy and control overhead.

8.7 Allocation

The main architectural issues of the MONTIUM that are relevant for the resource allocation phase are summarized as follows:

(1) The size of a memory is 512 words; (2) Each register bank includes 4 registers; (3) Only one word can be read from or written to a memory within one clock cycle; (4) The crossbar has a limited number of buses (10); (5) The execution time of the data-path is fixed (one clock cycle); (6) An ALU can only use the data from its local registers or from the east connection as inputs.

After scheduling, each cluster is assigned an ALU and the relative executing order of clusters has been determined. In the allocation phase, the other resources (busses, registers, memories, etc) are assigned, where locality of reference is exploited, which is important for performance and energy reasons. The main challenge in this phase is the limitation of the size of register banks and memories, the number of buses of the crossbar and the number of reading and writing ports of memories and register banks. The decisions that should be made during allocation phase are:

- Choose proper storage places (memories or registers) for each intermediate value;

- Arrange the resources (crossbar, address generators, etc) such that the outputs of the ALUs are stored in the right registers and memories;
- Arrange the resources such that the inputs of ALUs are in the proper register for the next cluster that will execute on that ALU.

Storing an ALU result must be done in the clock cycle within which the output is computed. When the outputs are not moved to registers or memories immediately after generated by ALUs, they will be lost. For this reason, in each clock cycle, storing outputs of the current clock cycle takes priority over using the resources. Preparing an input should be done at least one clock cycle before it is used. However, when it is prepared too early, the input will occupy the register space for a too long time. A proper heuristic is starting to prepare an input 4 clock cycles before the clock cycle it is actually used by the ALU. If the inputs are not well prepared before the execution of an ALU, one or more extra clock cycles need to be inserted to do so. However, this will decrease the runtime of the algorithm.

When a value is moved from a memory to a register, a check should be done whether it is necessary to keep the old copy in the memory or not. In most cases, a memory location can be released after the datum is fed into an ALU. An exception is when there is another cluster which shares the copy of the datum and that cluster has not been executed.

We adopt a heuristic resource allocation method, whose pseudocode is listed in Figure 8.7. The clusters in the scheduled graph are allocated level by level (lines 0–2). Firstly, for each level, the ALUs are allocated (line 4). Secondly, the outputs are stored through the crossbar (line 5). As said before, storing outputs is given highest priority. The locality of reference principle is employed again to choose a proper storage position (register or memory) for each output. The

```
//Input: Scheduled Clustered Graph G
//Output: The job of an FPFA tile for each clock cycle
0  function ResourseAllocation(G) {
1        for each level in G do  Allocate(level);
2  }
3  function Allocate(currentLevel) {
4    Allocate ALUs of the current clock cycle
5    for each output  do store it to a memory;
6    for each input of current level
7    do  try to move it to proper register at the clock cycle which is four steps
8           before; If failed,  do it three steps before; then two steps before;  one
9           step before.
10     if  some inputs are not moved successfully
11    then insert one or more clock cycles before the current one to load inputs
12 }
```

Figure 8.7. Pseudocode of the heuristic allocation algorithm.

Table 8.1. The resource allocation result for the 4-point FFT CDFG

Step	Actions
1	Load inputs for clusters of level 1 in Figure 8.6.
2	Clu0, Clu1, Clu2 and Clu3 are executed; Save outputs of step 2; Load inputs for clusters of level 2.
3	Clu4, Clu5, Clu6 and Clu7 are executed; Save outputs of step 3; Load inputs for clusters of level 3.
4	Clu8, Clu9, Clu10 and Clu11 are executed; Save outputs of step 4; Load inputs for clusters of level 4.
5	Clu12, Clu13, Clu14 and Clu15 are executed; Save outputs of step 5.

unused resources (busses, registers, memories) of previous steps are used to load the missing inputs (lines 6–9) for the current step. Finally, extra clock cycles might be inserted if some inputs could not be put in place by the preceding steps (lines 10–11).

The resource allocation result for the 4-point FFT CDFG is listed in Table 8.1. Before the execution of Clu0, Clu1, Clu2 and Clu3, an extra step (step 1) is needed to load their inputs to proper local registers. In all other steps, besides saving the result of current step, the resources are sufficient to loading the inputs for the next step, so no extra steps are needed. The 4-point FFT can be executed within 5 steps by one MONTIUM tile. Note that when a previous algorithm already left the input data in the right registers, step 1 is not needed and consequently the algorithm can be executed in 4 clock cycles.

8.8 Conclusion

In this paper we presented a method to map a process written in a high level language, such as C, to one MONTIUM tile. The mapping procedure is divided into four steps: translating the source code to a CDFG, clustering, scheduling and resource allocation. High performance and low power consumption are achieved by exploiting maximum parallelism and locality of reference respectively. In conclusion, using this mapping scheme the flexibility and efficiency of the MONTIUM architecture are exploited.

8.9 Related work

High level compilation for reconfigurable architectures has been the focus of many researchers, see [Callahan, 2000]. Most systems use the SUIF compiler of Stanford [http://suif.stanford.edu].

The procedure of mapping a task graph to a MONTIUM tile has NP complexity just like the task scheduling problem on multiprocessors systems [Rewini, 1994]. [Kim, 1988] considered linear clustering which is an important special case of clustering. [Sarkar, 1989] presents a clustering algorithm based on a scheduling algorithm on unbounded number of processors. In [Yang, 1994] a fast and accurate heuristic algorithm was proposed, the Dominant Sequence Clustering. The ALUs within our MONTIUM are interconnected more tightly than multiprocessors. This difference prevents us from using their solution directly.

To simplify the problem, we use a four-phased decomposition algorithm based on the two-phased decomposition of multiprocessor scheduling introduced by [Sarkar, 1989].

Acknowledgments

This research is conducted within the Chameleon project (TES.5004) and Gecko project (612.064.103) supported by the PROGram for Research on Embedded Systems & Software (PROGRESS) of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the technology foundation STW.

References

T.J. Callahan, J.R. Hauser, and J. Wawryzynek, "The Garp Architecture and C compiler" in *IEEE Computer*, 33(4), April 2000.

Yuanqing Guo, Gerard Smit, Paul Heysters, Hajo Broersma, "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System", *2003 ACM Sigplan Conference on Languages, Compilers, and Tools for Embedded Systems(LCTES'03)*, California, USA, June 2003, pp. 199–208.

S.J. Kim and J.C. Browne, A General Approach to Mapping of parallel Computation upon Multiprocessor Architectures, *International Conference on Parallel Processing,* vol 3, 1988, pp. 1–8.

Hesham EL-Rewini, Theodore Gyle Lewis, Hesham H. Ali, *Task scheduling in parallel and distributed systems,* PTR Prentice Hall, 1994.

Michel A.J. Rosien, Yuanqing Guo, Gerard J.M. Smit, Thijs Krol, "Mapping Applications to an FPFA Tile", *Proc. of Date 03*, Munich, March, 2003.

Vivek Sarkar. *Clustering and Scheduling Parallel Programs for Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, 1989.

Gerard J.M. Smit, Paul J.M. Havinga, Lodewijk T. Smit, Paul M. Heysters, Michel A.J. Rosien, "Dynamic Reconfiguration in Mobile Systems", *Proc. of FPL2002,* Montpellier France, pp. 171–181, September 2002.

SUIF Compiler system, `http://suif.stanford.edu`.

Tao Yang; Apostolos Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors", *IEEE Transactions on Parallel and Distributed Systems,* Volume:5 Issue:9, Sept. 1994 Page(s): 951–967.