# Mapping Applications to a Coarse Grain Reconfigurable System

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

University of Twente, Faculty of Electrical Engineering,
Mathematics and Computer Science
P.O. Box 217, 7500AE Enschede, The Netherlands
Phone: +31 53 4894178 Fax: +31 53 4894590
E-mail: {yguo, smit, broersma, rosien, heysters}@cs.utwente.nl

**Abstract.** This paper introduces a method which can be used to map applications written in a high level source language program, like C, to a coarse grain reconfigurable architecture, MONTIUM. The source code is first translated into a control dataflow graph. Then after applying graph clustering, scheduling and allocation on this control dataflow graph, it can be mapped onto the target architecture. The clustering and allocation algorithm are presented in detail. 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.

## 1 Introduction

In the CHAMELEON/GECKO[1]project we are designing a heterogeneous reconfigurable System-On-Chip (SoC) [12] (see Fig. 1). This SoC contains a general-purpose processor (ARM core), a bit-level reconfigurable part (FPGA) and several word-level reconfigurable parts (MONTIUM tiles; see Section 2). 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 show that a design method can be used to map processes, written in a high level language, to a reconfigurable platform. The methods can be used to optimize the system with respect to certain criteria e.g. energy efficiency or execution speed.

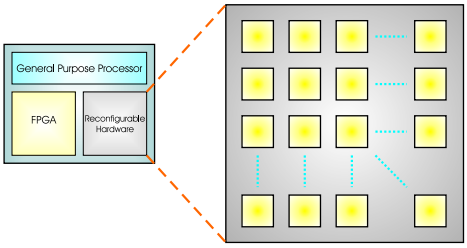



**Fig. 1.** CHAMELEON heterogeneous SoC architecture

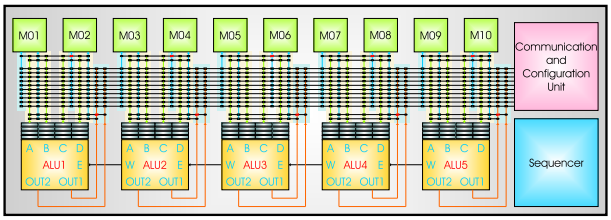



**Fig. 2.** MONTIUM processor tile

## 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. Fig. 2 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. A vertical segment that contains one ALU together with its associated input register files, a part of the interconnect and two local memories is called a processing part (PP). The five processing parts together are called the processing part array (PPA). A relatively simple sequencer controls the entire PPA. The communication and configuration unit (CCU) implements the interface with the world outside the tile. The MONTIUM has a datapath width of 16-bits and supports both integer and fixed-point arithmetic. Each local SRAM

[1] This research is supported by PROGram for Research on Embedded Systems & Software (PROGRESS) of the Netherlands Organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the technology foundation STW.

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.

The configuration of the interconnect can change every clock cycle. There are ten busses that are used for inter-PPA communication. Note that the span of these busses is only the PPA within a single tile. The CCU is also connected to the global busses. The CCU uses the global busses to access the local memories and to handle data in streaming algorithms. Communication within a PP uses the more energy-efficient local busses. 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. The diagram of the MONTIUM ALU in Fig. 3 identifies two different levels in the ALU. Level 1 contains four 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.

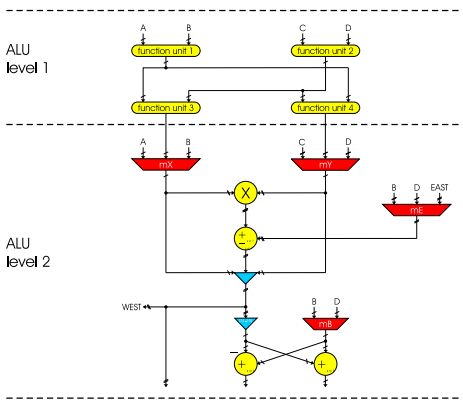



**Fig. 3.** MONTIUM ALU

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 when performing a complex multiplication, or when adding up a large amount of numbers (up to 20 in one clock cycle). The East-West connection does not introduce a delay or pipeline, as it is not registered.

## 3 Approach

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. We propose to decompose this problem into a number of phases: 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.

### 3.1 Some Definitions Regarding a CDFG

For the purpose of formulating our problem in a mathematical context, it is convenient to introduce a new type of graphs called **hydragraphs**[2] to model our directed acyclic CDFGs (CDFGs for short in this paper). This concept should capture and represent the operations, the inputs and outputs, as well as which inputs are used and which outputs are produced by the operations (and which outputs of a certain operation serve as inputs for one or more further operations).

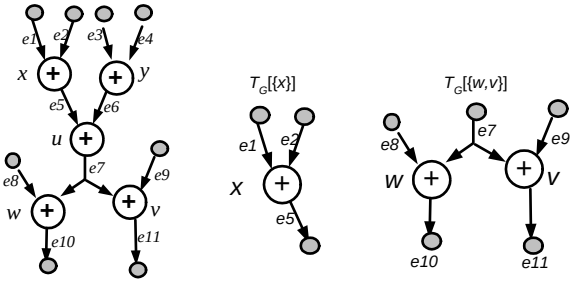
[2] These graphs are named after Hydra, a water-snake from Greek mythology with many heads that grew again if cut off.

A hydragraph $G = (N_G, P_G, A_G)$ 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 Fig. 4(a): 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 Fig. 4(a), for instance, has two heads, $w$ and $v$. The CDFG communicates with external systems through its ports represented by small grey circles in Fig. 4(a).

A node subset $S \in N_G$ generates a hydragraph in the following natural way: For every $v \in S$ consider the following two types of hydra-arcs of $G$ related to $v$:
- $(t_v, H_v)$, so hydra-arcs with tail $v$: if $H_v \not\subset S$, we introduce a new port $p_v$ and replace $(t_v, H_v)$ by $(t_v, (H_v \cap S) \cup \{p_v\})$; otherwise, we keep $(t_v, H_v)$ as it is.
- $(t_u, H_u)$ with $v \in H_u$, so hydra-arcs for which $v$ is one of the heads: if $t_u \notin S$, we introduce a new port $t'_u$ and replace $(t_u, H_u)$ by $(t'_u, H_u \cap S)$; otherwise we keep $(t_u, H_u)$ as it is.

Doing so for all hydra-arcs, e.g. starting from the sources in $S$, we obtain a unique hydragraph which we will refer to as the **template** generated by $S$ in $G$. We denote it by $T_G[S]$ and say that $S$ is a **match** of the template $T_G[S]$. In the sequel we will only consider connected templates without always stating this explicitly. For convenience let us call a template an $i$**-template** if the number of its nodes is $i$. Similarly $i$**-match** and $i$**-node subset** are defined.




(a) A small CDFG

(b) Two templates of the CDFG from Fig. 4(a)

**Fig. 4.** An example.

For example, in Fig. 4(b) we see two templates of the CDFG from Fig. 4(a): the left one is generated by the set $\{x\}$, the right one by $\{v, w\}$. Compared with the original CDFG from Fig. 4(a), in the left one, the newly added port is a head for hydra-arc $e5$, while in the right one the newly added port is a tail for hydra-arc $e7$.

Two hydragraphs $G$ and $F$ are said to be **isomorphic** if there is a bijection $\phi : N_G \cup P_G \to N_F \cup P_F$ such that:

$\phi(N_G) = N_F$, $\phi(P_G) = P_F$, and $(t_v, H_v) \in A_G$ if and only if $(\phi(t_v), \phi(H_v)) \in A_F$.

We use $G \cong F$ to denote that $G$ and $F$ are isomorphic.

We say that $S' \subset N_G$ is a **match for the template** $T_G[S]$ if $T_G[S'] \cong T_G[S]$. A hydragraph $H$ is a **template of the hydragraph** $G$ if, for some $S \subset N_G$, $T_G[S] \cong H$. Of course, the same template could have different matches in $G$.

Note that, in general, a template is not a subhydragraph of a hydragraph, because some nodes may have been replaced by ports. The important property of templates of a CDFG is that they are themselves CDFGs that model part of the algorithm modelled by the whole CDFG: the template $T_G[S]$ models the part of the algorithm characterized by the operations represented by the nodes of $S$, together with the inputs and outputs of that part. Because of this property, templates are the natural objects to consider if one wants to break up a large algorithm represented by a CDFG into smaller parts that have to be executed on ALUs. In this paper, we only consider connected templates.

## 4 Phase1: Translating C to a CDFG

In general, CDFGs are not acyclic. In the first phase we decompose the general CDFG into acyclic blocks and cyclic control information. In this paper we only consider acyclic graphs. 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. Fig. 5 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.

## 5 Phase2: Clustering

The input for clustering and data-path mapping is a CDFG. 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 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

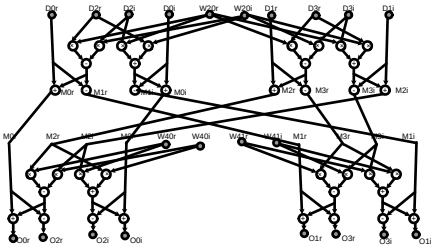


**Fig. 5.** The generated CDFG of a 4-point FFT after complete loop unrolling and full simplification.

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.

The clustering phase is implemented by a graph-covering algorithm [6]. The distinct configurations corresponds to distinct templates and the clusters corresponds to matches in [6]. The procedure of clustering is the procedure of finding a cover for a CDFG.

We say that a collection $(T_1, \ldots, T_k)$ of hydragraphs is a **$k$-tiling** of the hydragraph $G$ if there exists a partition of $N_G$ into mutually disjoint sets $S_1, \ldots, S_k$ such that $T_G[S_i] \cong T_i$ for all $i \in \{1, \ldots, k\}$. In that case we call $S_1, \ldots, S_k$ a **$k$-cover** of $G$. A **$(k, \ell)$-tiling** is a $k$-tiling in which at most $\ell$ nonisomorphic hydragraphs appear. Similarly, we define a **$(k, \ell)$-cover**.

**Problem 1: Hydragraph Covering Problem**
Given a CDFG $G$, find an optimal $(k, \ell)$-cover $S_1, S_2, \ldots, S_k$ of $G$. It is clear that we cannot expect to solve this complex optimization problem easily. We would be quite happy with a solution concept that gives approximate solutions of a reasonable quality, and that is flexible enough to allow for several solutions to choose from. We propose to start the search for a good solution by first generating all different matches (up to a certain number of nodes because of the restrictions set by the ALU-architecture) of nonisomorphic templates for the CDFG. The second step tries to find an efficient cover for an application graph with a minimal number of distinct templates and minimal number of matches.

**Problem 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.

**Problem B: Template Selection Problem**
Given a CDFG $G$ and a set of (matches of) templates, find a 'optimal' $(k, \ell)$-cover of $G$.

### 5.1 Template Generation

A clear approach for the generating procedure is:

1 Generate a set of connected $i$-node subsets by adding a neighbor node to the $(i-1)$-node subsets.
2 For all $i$-node subsets, consider their generated $i$-templates. Choose the set of nonisomorphic $i$-templates and list all matches of each of them.
3 Starting with the 1-node subsets, repeat the above steps until all templates and matches op to *maxsize* nodes have been generated.

In step 1, an $i$-node subsets can be obtained by different $(i-1)$-node subsets, which will result in unnecessarily many computations. To avoid this, we use a clever labelling of the nodes during the generation process depicted in detail in [6]:

- Each hydragraph node is given a unique serial number.
- A leading node is defined within each node subset $S$, which is the one with the smallest serial number.
- Within a subset $S$, each graph node $n \in S$ is given a **circle number**, denoted by Cir($n|S$), which is the distance between the leading node and $n$ within $S$, i.e., Cir($n|S$)=Dis($S$.LeadingNode, $n|S$).

If a $(i-1)$-node subset $S$ and one of its neighbor node *Nei* satisfy the following conditions, $S' = S \cap \{Nei\}$ will be considered as a $i$-node subset, otherwise $S'$ is thrown away.

1 $S$.LeadingNode.Serial<*Nei*.Serial;
2 Dis($S$.LeadingNode, $Nei|S \cup \{Nei\}$) is not smaller than Cir($n|S$) for any $n \in S$;
3 For each $n$ which satisfies $n \in S$ and
Cir($n|S$) = Dis($S$.LeadingNode,$Nei|S \cup \{Nei\}$), $n$.Serial< *Nei*.Serial.

For each $i$-template $S'$, these conditions chose a unique pair ($S$,*Nei*) such that $S' = S \cap \{Nei\}$. Thus multiple copies of $S'$ are discarded. The proof can be found in [6].

### 5.2 The Template Selection Algorithm

Given $G$, $\Omega = \{T_1, T_2, \cdots, T_p\}$ and the matches $M(\Omega)$, the objective is to find a subset $C$ of the set $M(\Omega)$ that forms a 'good' cover of $G$. Here by 'good' cover we mean a $(k, \ell)$-cover with minimum $k$ and $\ell$.

Since the generated set $M(\Omega)$ can be quite large, the template and match selection problem is computationally intensive. We adopt a heuristic based on maximum independent set, and apply it to a conflict graph related to our problem, similarly as it was done in [8][10].

A **conflict graph** is an undirected graph $\tilde{G} = (V, E)$. Each match $S \in M(\Omega)$ for a template of the CDFG $G$ is represented by a vertex $v_S$ in the conflict graph $\tilde{G}$. If two matches $S_1$ and $S_2$ have one or more nodes in common, there will be an

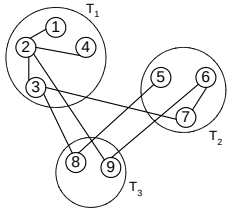



**Fig. 6.** A conflict graph. The weight of each node is 4.

edge between the two corresponding vertices $v_{S_1}$ and $v_{S_2}$ in the conflict graph $\tilde{G}$. The **weight** $w(v_S)$ of a conflict graph vertex $v_S$ is the number of CDFG nodes $|S|$ within the corresponding match $S$. The vertex set of the conflict graph are partitioned into subsets, each of which corresponds to a certain template (see Fig. 6). Therefore, on the conflict graph, vertices of the same subset have the same weight. The **maximum independent set (MIS) for a subset** $T \subset V(\tilde{G})$ is defined as the largest subset of vertices within $T$ that are mutually nonadjacent. There might exist more than one MIS for $T$. Corresponding to each MIS for $T$ on $\tilde{G}$, there exists a set of node-disjoint matches in $G$ for the template corresponding to $T$; we call this set of matches a **maximum non-overlapping match set**(MNOMS). To determine a cover of $G$ with a small number of distinct templates, the templates should cover a rather large number of CDFG nodes, on average.

An MNOMS corresponds to a MIS on the conflict graph. Finding a MIS in a general graph, however, is an NP-hard problem [5]. Fortunately, there are several heuristics for this problem that give reasonably good solutions in practical situations. One of these heuristics is a simple minimum-degree based algorithm used in [7], where it has been shown to give good results. Therefore, we adopted this algorithm as a first approach to finding 'good' coverings for the CDFGs within our research project.

For each template $T$, an **objective function** is defined by:

$$g(T) = g(w, s),$$

where $w$ is the weight of each vertex and $s$ is the size of an approximate solution for a MIS within the subset corresponding to $T$ on the conflict graph. The outcome of our heuristic will highly depend on the choice of this objective function, as we will see later.

The pseudo-code of the selection algorithm is shown in Fig. 7. This is an iterative procedure, similar to the methods in [1][3][10]. At each round, after computing an approximate solution for the MISs within each subset, out of all templates in $\Omega$, the heuristic approach selects a template $T$ with a maximum value of the objective function, depending on the weights and approximate solutions for the MISs. After that, on the conflict graph, the neighbor vertices of the selected approximate MIS and of the approximate MIS itself are deleted. This

procedure is repeated until the set of matches $C$ corresponding to the union of the chosen approximate MISs, covers the whole CDFG $G$.

1 Cover $C = \phi$;
2 Build the conflict graph;
3 Find a MIS for each group on the conflict graph;
4 Compute the value of objective function for each template; The $T$ with the largest value of objective function is the selected template. Its MIS is the selected MIS. The MNOMS corresponding to the MIS are put into $C$. On the conflict graph, delete the neighbor vertices of the selected MIS, and then delete the selected MIS;
5 Can $C$ cover CDFG totally? If no, go back to 3; if yes, end the program.

**Fig. 7.** Pseudo-code of the proposed template selection algorithm

We currently use the following objective function:

$$g(T) = w^{1.2} \cdot s = ws \cdot w^{0.2}. \tag{1}$$

In this function, for a template $T$, $ws$ equals the total number of CDFG nodes covered by a MNOMS, which expresses a preference for the template whose MNOMS covers the largest number of nodes. Furthermore, due to the extra $w^{0.2}$ factor, the larger templates, i.e., the templates with more template nodes, are more likely to be chosen than the smaller templates.

Each selected match is a cluster that can be mapped onto one MONTIUM ALU and can be executed in one clock-cycle. As an example Fig. 8 presents the produced cover for the 4-point FFT. The letters inside the dark circles indicate the templates. For this CDFG, among all the templates, three have the highest objective function value. The hydragraph is completely covered by them. This result is the same as our manual solution. It appears that the same templates are chosen for a $n$-point FFT ($n = 2^d$). After clustering, we get a clustered graph shown in Fig. 9.

## 6 Phase3: Scheduling

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

- 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 (see Fig. 3).
- Clusters that can be executed in parallel can 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

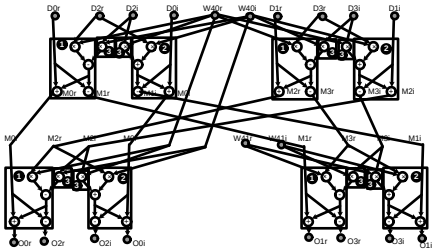


**Fig. 8.** The selected cover for the CDFG in 5

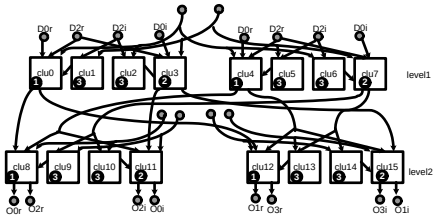



**Fig. 9.** The clustering result for the CDFG from Fig. 5.

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 Fig. 9 as an example, where, in level one, the clusters of type 1 and type 2 are dependent on clusters of type 3. Therefore, type 3 clusters should be executed before the corresponding type 1 or type 2 cluster, or they are executed by two adjacent ALUs in the same clock cycle, in which case type 3 clusters must stay east to the connected type 1 or type 2 cluster. Because there are too many clusters in level 1 and level 2 of Fig. 9, we have to split them. Fig. 10(a) shows a possible scheduling scheme where not all five ALUs are used. This scheme consists of only one 5-ALU configuration: $C1$={❶❸❷❸○}. As a result, with the scheme of 10(a), the configuration of ALUs stays the same during the execution. The scheduling scheme of Fig. 10(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.

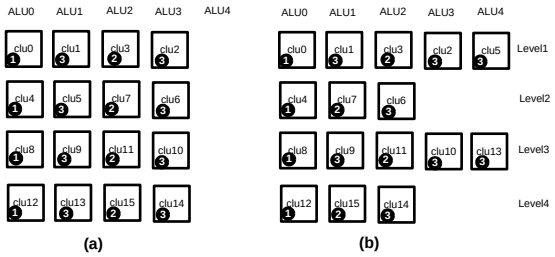



**Fig. 10.** Schedule the ALUs of Fig. 9

## 7 Phase4: Allocation

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

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

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. Preparing an input should be done 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 solution in practise is starting to prepare an input 4 clock cycles before the clock cycle it is actually used by the ALU. 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. If the inputs are not well prepared before the execution of an ALU, one or more extra clock cycles can be inserted to do so. However, this will decrease the speed.

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 that 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 Fig. 11. The clusters in the scheduled graph are allocated level by level (lines

```
//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 }
```

**Fig. 11.** Pseudocode of the heuristic allocation algorithm

0-2). Firstly, for each level, the ALUs are allocated (line 4). Secondly, the outputs are stored through the crossbar (line 5). Storing outputs is given priority because the outputs will be lost when they are not moved to registers or memories immediately after generated by the ALUs. The locality of reference principle is employed again to choose a proper storage position (register or memory) for each output. The 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 are not put in place by the preceding steps (lines 10-11).

The resource allocation result for the 4-point FFT CDFG is listed in table 1. Before the execution of Clu0, Clu1, Clu2 and Clu3, an extra step (step 1) is needed to load there 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.

**Table 1.** The resource allocation result for the 4-point FFT CDFG

| Step | Actions |
|---|---|
| 1 | Load inputs for clusters of level 1 in Fig. 10 |
| 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. |

## 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. We introduced a new type of graph (hydragraph) to represent a CDFG.

To date, the work does not deal with CDFGs with loops and branches, which will be done in the future work. Furthermore, the scheduling and resource allocation steps will be investigated in more detail.

## 9 Related work

There have been published many related research efforts in the areas of FPGA logic synthesis. Many systems use the SUIF compiler of Stanford [13].

For multiprocessor systems, Sarkar [11] presents a clustering algorithm based on a scheduling algorithm on unbounded number of processors. Our MONTIUM is a VLIW processor instead of multiprocessor. To simplify the problem, we still employ a four phase decomposition algorithm based on the two-phased decomposition of multiprocessor scheduling introduced by Sarkar [11].

Clustering is the key parts in our decomposition. In [2][4], a template library is assumed to be available and the template matching is the focus of their work. However, this assumption is not always valid, and hence an automatic compiler must determine the possible templates by itself before coming up with suitable matchings.

[9][10] give some methods to generate templates. These approaches choose one node as an initial template and subsequently add more operators to the template. The drawback is that the generated templates are highly dependent on the choice of the initial template. The heuristic algorithm in [8] generates and maps templates simultaneously, but cannot avoid ill-fated decisions.

The algorithms in [1][3] provide all templates of a CDFG. The central problem for template generation algorithms is how to generate and enumerate all the (connected) subgraphs of a CDFG. The methods employed in [3] and [1] can only enumerate the subgraphs of specific shapes (tree shape, single output or single sink) and as a result, templates with multiple outputs or multiple sinks cannot be generated. In the MONTIUM architecture, each ALU has three outputs, so the existing algorithms cannot be used.

## References


1. Srihari Cadambi, and Seth Copen Goldstein, "CPR: A Configuration Profiling Tool", *IEEE Symposium on FPGAs for Custom Computing Machines,* 1999.
2. Timothy J.Callahan, Philip Chong, Andre DeHon, and John Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs", *Proc. of International Sysp. of Field Programmable Gate Arrays,* 1998.
3. Amit Chowdhary, Sudhakar Kale, Phani Saripella, Naresh Sehgal, Rajesh Gupta, "A General Approach for Regularity Extraction in Datapath Circuits", *Proc. of Internaltional Conference on Computer-Aided Design (ICCAD)* San Jose, CA, 1998, pp.332-339.
4. Miguel R. Corazao, Marwan A. Khalaf, Lisa M.Guerra, Miodrag Potkonjak and Jan M. Rabaey, "Performance Optimization Using Templete mapping for Datapath-Intensive High-Level Synthesis", *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems*, vol.15, No.8, August 1996, pp.877-888.
5. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.
6. 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.
7. Magnús M. Halldórsson, Jaikumar Radhakrishnan, "Greed is good: Approximating independent sets in sparse and bounded-degree graphs", *ACM Symposium on the Theory of Computing*, 1994.
8. Ryan Kastner, Seda Ogrenci-Memik, Elaheh Bozorgzadeh and Majid Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems", *Proc. of International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, November, 2001.
9. Thomas Kutzschebauch, "Efficient Logic Optimization Using Regularity Extraction", *Proc. of the 1999 Internaltional Workshop on Logic Synthesis*, 1999.
10. D. Sreenivasa Rao, and Fadi J. Kurdahi, "On Clustering For Maximal Regularity Extraction", *IEEE Transactions on Computer-Aided Design*, vol.12, No.8,August,1993, pp.1198-1208.
11. Vivek Sarkar. *Clustering and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, 1989.
12. 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.
13. SUIF Compiler system, `http://suif.stanford.edu`.