CHALMERS

Optimization Engine for the FlexTools Design Space Exploration Platform

Master of Science Thesis in Integrated Electronic Systems Design

SALAR ALIPOUR
BABAK HIDAJI

Chalmers University of Technology
Department of Computer Science and Engineering
Gothenburg, Sweden, February 2011

Optimization Engine for the FlexTools Design Space Exploration Platform

Salar Alipour, Babak Hidaji

Examiner: Prof. Per Larsson-Edefors

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden, February 2011

THESIS FOR THE DEGREE OF MASTER

Optimization Engine for the FlexTools Design Space Exploration Platform

SALAR ALIPOUR, BABAK HIDAJI

Division of Computer Engineering
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2011

Optimization Engine for the FlexTools Design Space Exploration Platform
Salar Alipour, Babak Hidaji

Department of Computer Science and Engineering
VLSI Research Group

Division of Computer Engineering
Chalmers University of Technology
SE-412 96 GOTHENBURG, Sweden
Phone: +46 (0)31-772 10 00

Authors e-mail: *salaralipour84@gmail.com, bhidaji@gmail.com*

Printed by Chalmers Reproservice
Gothenburg, Sweden 2011

Optimization Engine for the FlexTools Design Space Exploration Platform

Salar Alipour, Babak Hidaji

Division of Computer Engineering, Chalmers University of Technology

ABSTRACT

A general-purpose datapath interconnect is designed to make the processor efficient in executing a wide array of diverse applications. An embedded processor, typically working with a limited application domain, does not necessarily utilize the fixed, general-purpose datapath interconnect efficiently. If we consider the interconnect to be a flexible resource, the datapath can be fine tuned to one or a few applications. The addition of an interconnect link between two datapath units has the potential to reduce execution time, while the removal of an unused link can save area and power dissipation. Finding the most energy-efficient datapath interconnect configuration for a software application domain is a time-consuming process, since it involves rescheduling of the targeted application(s) on different datapath implementations. We present an automated optimization engine that is based on a genetic algorithm. This engine aids the designer in finding the most energy-efficient interconnect configuration of a simple processor datapath. We show that an optimized datapath interconnect can offer an energy saving of 38% with respect to a general-purpose datapath reference, if the interconnect links are matched to the need of one application.

Keywords: Embedded, Flexible, Fine-grained, Design Space Exploration, Energy.

Acknowledgments

First of all we have to express our gratitude to our mentor, examiner and supervisor Professor Per Larsson-Edefors for his uncountable advice, enthusiasm and patience. Without his contribution this work would not have been possible.

We are deeply indebted to many of our friends and colleagues to support us during this project:

▷ Kasyab P. Subramaniyan as our supervisor for many technical discussions and being the co-author of our papers.

▷ Lars Svensson and Tung Thanh Hoang for their technical advice and providing appropriate documents related to the project.

▷ Administrators at the CSE department: Lars Kollberg and Peter Helander, Rune L. for IT support and personnel consultants.

▷ All the members of the FlexSoC project for various contributions.

▷ All the other colleagues at the CSE department that we had opportunities to work with.

▷ To our family for being patient, supportive and motivating during our studies.

Salar Alipour, Babak Hidaji
Gothenburg, February 2011

Contents

List of Figures

List of Tables

Part I

Application-Specific Energy Optimization Engine

Chapter 1

Introduction

Embedded processors are typically designed along the lines of a general-purpose processor (GPP) using a RISC design philosophy, in order to make the processor operate efficiently on a wide array of application workloads. The drawback of such an approach is that application flexibility is achieved at the cost of lower efficiency in the context of more specific, embedded workloads.

Common approaches to development of processors for application domains include reconfigurable coprocessors augmented to a standard GPP processor and coarse-grain GPP datapath reconfigurations. We explore another option, more specifically to tailor the GPP to the applications at hand by reworking the interconnect inside the datapath. An interconnect that is flexible may not only allow for a more efficient, application-dependent communication pattern, but it can also support a modular and scalable datapath in which more units can be added. The interaction of datapath circuitry and software application is complex for such a scheme, making design-space exploration (DSE) a challenge. Given some datapath units, how can we tell whether a specific communication path in the interconnect is useful or not? While it may increase performance by reducing cycle count, an extra interconnect link leads to an area overhead and, possibly, an increase in power dissipation.

We present work on energy optimization of a flexible datapath, that in terms of units, has many similarities with a general-purpose 5-stage processor. It is challenging for designers to approach an optimal interconnect configuration, given that this requires an iterative DSE procedure in which long applications need to run on accurate and complex models of many different hardware configurations. As a first step to performing overall processor energy optimization, this thesis work demonstrates an optimization engine that automates the process of finding the datapath interconnect configuration with the lowest possible energy dissipation.

1.1 FlexSoC

FlexSoc (Flexible System-on-Chip) is a project launched in 2003 at Chalmers University of Technology. The project is founded by the Swedish Strategic Research Foundation (SSF) [1]. Most System on Chip architectures use one or more application specific processing elements or accelerators together with a general-purpose processor. The accelerators are used to perform computation-time consuming tasks of the frequently requested applications. The GPP processor will control the accelerators or specialized blocks and also handle the rest of the computations [2].

There are two major considerations towards this approach. The first one is programmability of such design; the processor is fine tuned for the targeted application and if there was a need to do a slight alteration in the application, all the processor would act just like a regular GPP. This means that the benefits of using the specific designed processor would disappear. The second problem is that conventional software development tools may not interact well with system on chip designs [3].

The FlexSoC program provides an alternative approach for processor's design. In this design there is a network of datapath units that are controlled by means of a reconfigurable instruction sets [1].

1.1.1 FlexCore

FlexCore is a processor design based on the FlexSoC architecture in 32-bit GPP architecture fashion. In conventional GPP architectures the number of datapath units and also their connections are fixed, however FlexCore do not follow such scheme. At design time, it is possible to add a number of units (e.g. accelerators, more computational units, ALUs, or any specialized block) to the basic FlexCore architecture. Moreover the interconnect between these units is reconfigurable in order to allow any type of possible data routing between these datapath units.

The reconfigurability of the interconnect is obtained by using several switch boxes that connect outputs of all datapath units to each input. These connections are controlled using a Native Instruction Set (N-ISA). The N-ISA that is specifically developed for for FlexCore consists of two parts; the address bits of the switch boxes and also control bits for the functional units. At runtime the appropriate units for each cycle are connected to each other [3].

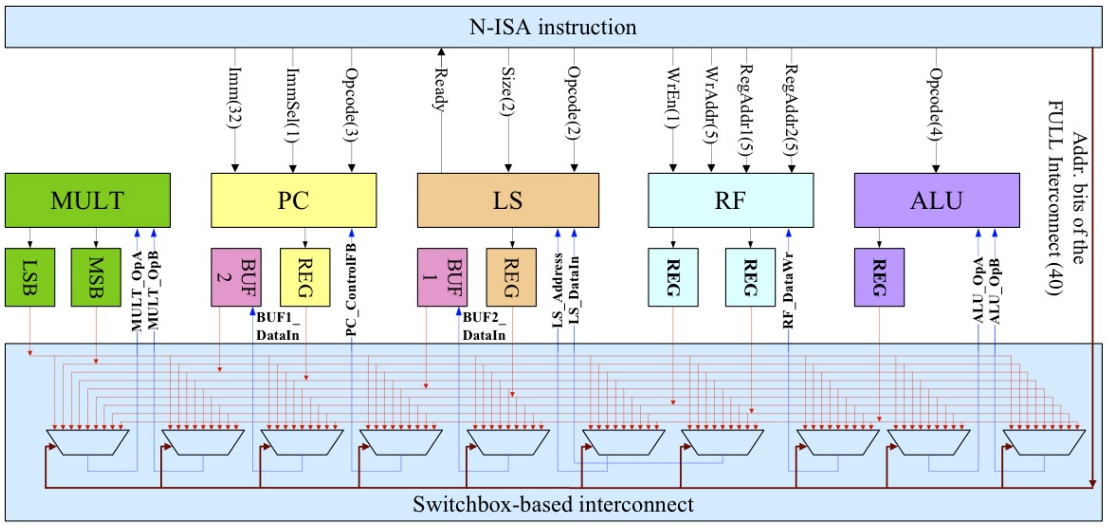

Figure 1.1: *Multiplier-extended FlexCore processor with a full 90-link interconnect [4].*

As shown in the Fig. 1.1 a basic FlexCore that is extended by a multiplier comes with a 90-link interconnect. This results in 2^{90} configurations, however not all these configurations are possible. There are some links that we should always keep connected (in order to keep GPP functionality or just guarantee the compilation) and some that are never used in any type of application.

1.1.2 FlexTools

The FlexTools tool chain is the design and verification environment for FlexCore processors [4]. At the early design stages, FlexTools accepts a user-defined datapath and interconnect configuration, and the C code of the applications to be executed. Subsequently FlexTools performs a simulation of the application code on the given FlexCore datapath configuration to obtain statistics on cycle count and interconnect link utilization.

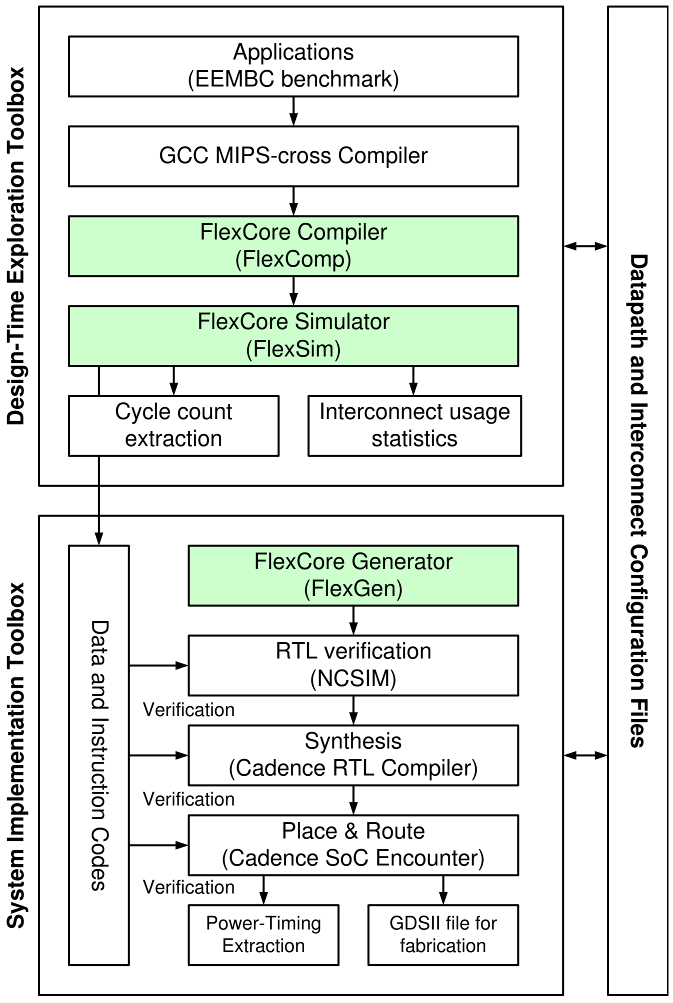
Design-Time Exploration Toolbox
Applications (EEMBC benchmark)
GCC MIPS-cross Compiler
FlexCore Compiler (FlexComp)
FlexCore Simulator (FlexSim)
Cycle count extraction
Interconnect usage statistics
System Implementation Toolbox
Data and Instruction Codes
FlexCore Generator (FlexGen)
RTL verification (NCSIM)
Verification
Synthesis (Cadence RTL Compiler)
Verification
Place & Route (Cadence SoC Encounter)
Verification
Power-Timing Extraction
GDSII file for fabrication
Datapath and Interconnect Configuration Files

Figure 1.2: *Illustration of the complete FlexTools environment [3].*

Fig. 1.2 shows the structure of FlexTools consists of two toolboxes; that of design-time exploration (DTE) and that of system implementation (SI). The role of the DTE toolbox is to perform compilation/scheduling and simulation to extract the application cycle count, which is needed for evaluating performance and energy dissipation, and interconnect usage statistics that is used to recognize unused or rarely used links.

As part of the SI toolbox, the RTL generator generates the top-level VHDL module and a testbench for the chosen hardware configuration and the specified software application. The major duties of the rest parts of the SI toolbox including RTL verification, logic synthesis and place and route are to enable fabrication in different ASIC process technologies and to provide highly accurate area, timing and power dissipation statistics [4].

The FelxGen Tool generates a FlexCore processor by reading the input files, "flex.ini" that provides the datapath units functionality and the number of them, and also the "interconnect.cvs" file that consists of interconnect configuration data. Table 1.1 shows a "flex.ini" file that represents a Multiplier-extended FlexCore processor. The "interconnect.cvs" file depicted in Table 1.2 is the representation of an interconnect configuration called "GPP-lite" that mimics the functionality of a MIPS R2000 processor.

Table 1.1: *A sample "flex.ini" file representing a Multiplier-extended FlexCore processor*

```
alu=1
agu=0
buffers=2
mult=1
read-poert=4
write-ports=2
multi-delay=2
verify=0
```

FlexTools uses a MIPS cross-compiler to translate the applications' C code into the FlexCore N-ISA. This instruction set consist of the control bits for the data path units, the switch boxes' addresses, and also immediate values. The FlexComp tool schedules the processor based on the specific benchmark that is used, number and functionality of the datapath units, and the interconnect configuration.

Since the compiler uses heuristic algorithms, it is not so easy to predict its behavior. Based on the nature of the FlexCore and the need for extra bits in the instruction set to control the switch boxes the size of N-ISA is larger compared to conventional GPPs and this will result in having larger programs. In later versions of the FlexCore this problem is partially addressed by using a compression scheme. The primary ISA is compressed, and at run time by using a decoder at the top level the ISA is decoded to fit its desired size and is used to control the processor.

The next step is simulating the processor for the specific targeted application (The FlexSim tool). The outcome of the simulation consists of valuable data that can be used by the designer.

1. Cycle count: is the number of the cycles that takes for the processor with the current configuration to run the target application. Based on the cycle count and the clock period, the performance of the processor for the specific benchmark is obtainable.

Table 1.2: *An "interconnect.cvs" file, representing a GPP configuration*

	Alu_OpB	Alu_OpA	Regbank_Wr	Ls_Address	Ls_Write	Buf1_Write	Buf2_Write	PC_FB	Mult_OpB	Mult_OpA
Mult_LSW	1	1	1	0	0	0	0	0	0	0
Mult_MSW	1	1	1	0	0	0	0	0	0	0
PC_ImmPC	1	0	1	0	0	1	0	0	0	1
Buf2_Read	1	1	1	1	0	0	0	1	0	0
Buf1_Read	0	0	0	0	1	0	1	0	0	0
Ls_Read	1	1	1	0	0	0	0	1	0	0
Regbank_Out1	0	1	0	0	0	0	1	0	0	1
Regbank_Out2	1	0	0	0	0	1	0	1	1	0
Alu_Rslt	0	1	1	1	0	1	0	1	0	0

2. Links' usage statistics: FlexSim generates a file containing the statistics for each available interconnect link during running the application. A usage statistics file that was generated for an interconnect with all links available (*Full Interconnect*) running autocorrelation benchmark is shown inTable 1.3. The data in the file reveals for instance, the link connecting ALU's output Alu_Rslt to the register bank's input Regbank_Wr was used in 2320 cycles during the simulation.

Table 1.3: *Full Interconnect usage statistics file*

	Alu_OpB	Alu_OpA	Regbank_Wr	Ls_Address	Ls_Write	Buf1_Write	Buf2_Write	PC_FB	Mult_OpB	Mult_OpA
Mult_LSW	1	100	0	0	0	0	0	0	0	0
Mult_MSW	0	0	0	0	0	0	0	0	0	0
PC_ImmPC	3870	0	768	669	8	59	98	0	0	0
Buf2_Read	78	99	232	140	7	44	0	0	0	0
Buf1_Read	21	134	210	4	3	0	185	109	0	0
Ls_Read	3	54	1078	1	57	26	57	799	0	100
Regbank_Out1	270	2284	135	827	265	40	24	555	100	0
Regbank_Out2	1585	2128	68	214	258	1	0	7	0	0
Alu_Rslt	101	1129	2320	612	542	402	179	1454	0	0

Having the usage statistics file one can easily recognize those links that were never used during the simulation or those that were rarely used. Since having extra links means larger switch boxes, which consequently result in more power consumption of the switch box, the designer may benefit from removing those links. However as the compile algorithms are heuristic, it is possible to remove an unused link and later during the simulation find out that the link's usage statistics has changed.

Flextools performs RTL verification by utilizing Simrtl tool. The power estimation of the design can be obtained by using the VCD (Value Changed Dump) file that was generated during simulation phase. The tool can perform this estimation either in the synthesis phase called Synvcd or in the place and route phase that is called Socvcd.

1.2 Evolutionary algorithms

The theory of evolutionary algorithms is based on Darwin theory of evolution, which introduces the gradual hereditary change in the individuals of a species. Evolutionary algorithms, as the name implies, pursue the processes identical to biological evolution procedure such as adaption to difficulties offered by environment. Genetic algorithm (GA), Linear genetic programming (LGP), and Interactive evolutionary computation (IEC) considered as three different EA optimization techniques. Genetic algorithm as the most widespread type of evolutionary algorithms has been used to address the optimization problem in this project [5].

1.2.1 Genetic algorithm

In this section some necessary information about genetic algorithm and the different components of GA will be presented. For the sake of clarification we start with brief definitions of some important terms in genetic algorithm.

Search space: A group of permitted values of one specific variable.

Chromosome: A string of digits (genes) that contains the encoded value of variables.

Although different encoding schemes exist, in this project we use binary encoding scheme where the genes take 0 or 1 values.

1.2.2 Components of Genetic algorithm

In order to have good understanding of the genetic algorithm, it is important to grasp the duty of different component of the algorithm. This can also help us to distinguish the different parts of the algorithm during the implementation. In the following parts we explain the characteristics of different components of genetic algorithm and the alternative techniques of developing them .

Initialization

In order to use the data that we would like to optimize in the genetic algorithm, we need to convert them to string of bits which is called chromosome in genetic algorithm's scheme. To do so we assign the value of 0 or 1 to the bits (genes) in the chromosomes. This method is called binary encoding scheme, which was used, in initial genetic algorithm by Holland and others in the 1970s [6].

To generate the initial population, there is a need to generate N random binary strings (chromosome). The typical size of N in genetic algorithm is 30-1000 and it can vary by user's decision based on the targeted problem. At the early stage, this initial population will be evaluated to go through the other essential parts of the genetic algorithm.

Selection

In order to from the new generation we need to select the chromosomes in pair for sexual reproduction, which is explained later in this chapter. The *fitness value* is a factor that determines the probability of a chromosome to be selected to produce the next generation.

In this section we introduce three different selection techniques that can be used in genetic algorithm.

1) **Roulette wheel selection:** In this method, as it has been shown in Fig. 1.3, we assign every chromosome (as a representative of the possible solution) to a slice of a roulette wheel (the size of the slice varies based on the chromosome's fitness value). Then by producing a random value in range of fitness functions we select a chromosome that corresponds to the selected slice of the wheel.

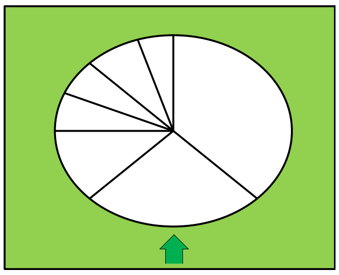

Figure 1.3: *Roulette-Wheel Selection*

Table 1.4 shows an example of Roulette-Wheel Selection technique applied on some FlexCore interconnect configurations. As the table reveals in this example the *selection probability* of different solutions are very close to each other and selecting a proper solution among the others —by applying Roulette-Wheel Selection method— is almost impossible due to same size of the slices of the wheel.

Table 1.4: *An example of Roulette-Wheel Selection technique applied on some FlexCore Interconnect configurations*

Position	Name	Energy (μJ)	Sum of Energy (μJ)	Selection Probability
1	I-100014	4.1591	4.1591	0.1001
2	I-100087	4.15992	8.319	0.1001
3	I-100034	4.16285	12.481	0.1001
4	I-100048	4.17761	16.658	0.1000
5	I-100078	4.21657	20.875	0.0999
6	I-100028	4.22643	25.101	0.0999
7	GPP	4.23918	29.340	0.0998
8	I-100013	4.24141	33.582	0.0998
9	I-100086	4.24867	37.830	0.0998
10	I-100091	4.25037	42.081	0.0998

2) **Tournament selection:** This method is based on selecting two random individuals at the early stage and comparing them to select the best one among this pair based on their fitness value with the probability of P_{tour}. This method can be generalized to apply the selection to more than two individuals. In this general

case, *k* individuals are chosen and then with a probability of P_{tour} we select the best individual. If the best chromosome is not selected, we repeat the procedure this time for *k-1* individuals with removing the best chromosome. The process is continued until we either select the best individual among the group or when the group only contains one chromosome [5].

3) **Rank Selection:** In this method first, we rank the chromosomes based on their evaluation's results. Then a fitness value proportional to their place in the ranked list using the selective pressure will be generated and is assigned to the chromosomes. Selective pressure (SP) is the probability of selecting the best individual comparing the average probability of selecting all individuals [7]. Finally by producing a random value in range of the total fitness values we select the chromosomes for reproduction. Table 1.5 shows an example of Rank Selection method applied on FlexCore optimization algorithm. In this table FV is the fitness value of each individual, Sum Fv is the summation of the fitness value of the individual with the total fitness values of previous individuals, and SP is the selective pressure. To demonstrate the effect of the selective pressure 3 different values of 1, 1.5, and 2 were assigned to SP.

Table 1.5: *Rank Selection technique with demonstrating the effect of Selective pressure (SP)*

Position	Name	Energy (μJ)	SP=1		SP=1.5		SP=2	
			FV	Sum Fv	FV	Sum Fv	FV	Sum Fv
1	I-100014	4.1591	1	1	1.5	1.5	2	2
2	I-100087	4.15992	1	2	1.388	2.888	1.777	3.777
3	I-100034	4.16285	1	3	1.277	4.166	1.555	5.333
4	I-100048	4.17761	1	4	1.166	5.333	1.333	6.666
5	I-100078	4.21657	1	5	1.055	6.388	1.111	7.777
6	I-100028	4.22643	1	6	0.944	7.333	0.888	8.666
7	GPP	4.23918	1	7	0.833	8.166	0.666	9.333
8	I-100013	4.24141	1	8	0.722	8.888	0.444	9.777
9	I-100086	4.24867	1	9	0.611	9.5	0.222	10
10	I-100091	4.25037	1	10	0.5	10	0	10

Comparison of Table 1.4 and 1.5 demonstrates that the Rank Selection method is more appropriate than the Roulette-Wheel Selection for the problems that have populations of individuals with extremely close evaluation's results.

Crossover

Reproduction is a process that creates new offspring for the next generation. During reproduction new genetic combination occur and the offspring inherit the features of the parents. Reproduction can be categorize in to two main domains [5]:

- **Asexual:** In asexual reproduction, offspring are produced by one individual of that species (e.g. bacteria).
- **Sexual:** In sexual reproduction two individuals generate the new offspring for next generation.

Crossover is one of the most important components of the genetic algorithm, which is inspired by the reproduction process in nature. Crossover can be implemented by many different techniques. Here we describe three important crossover mechanisms that are used in our genetic algorithm.

1) **One-point crossover:** As shown in Fig. 1.4 in this method both chromosomes are split through one crossing point and their corresponding segments are swapped to generate new chromosomes for the next generation.

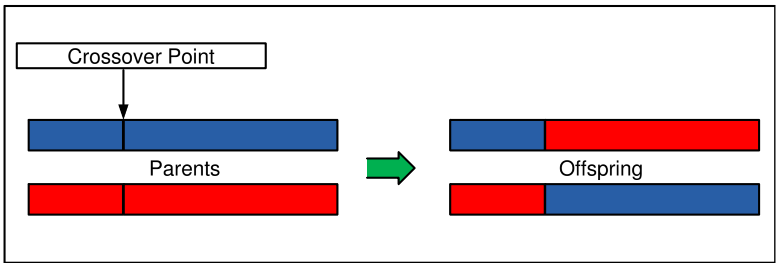

Figure 1.4: *One-Point Crossover*

2) **Two-point crossover:** The second alternative to perform reproduction in genetic algorithm scheme is two-point crossover. In this method both chromosomes are split through two different crossing points and their corresponding segments are swapped to generate new chromosomes for next generation. The complete process of two-point crossover is shown in Fig. 1.5.

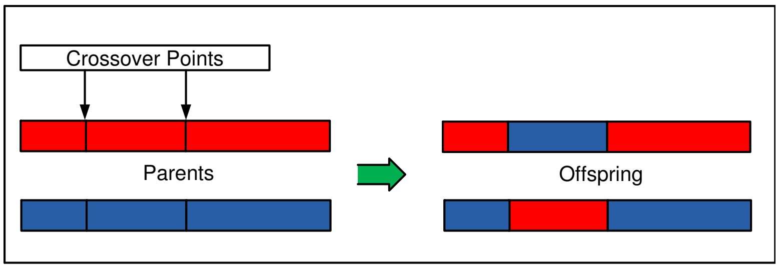

Figure 1.5: *Two-Point Crossover*

3) **Uniform crossover:** Uniform crossover is the most disruptive method among the crossover methods.

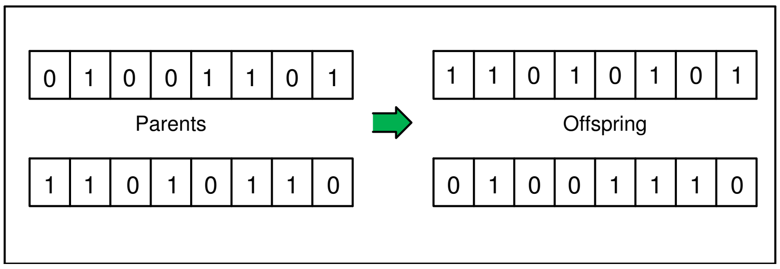

Figure 1.6: *Uniform Crossover*

As it can be seen in Fig. 1.6, in this method some randomly (Usually with probability of 0.5) selected bits swap between parents to create new offspring.

The probability of the crossover being carried out for every two-selected chromosome is called crossover probability (P_c).

Mutation

In binary chromosomes, mutation refers to flipping a specific bit from 1 to 0 or vice versa. These specific bits can be selected randomly by defining a mutation probability (P_{mut}) by the user of genetic algorithm.

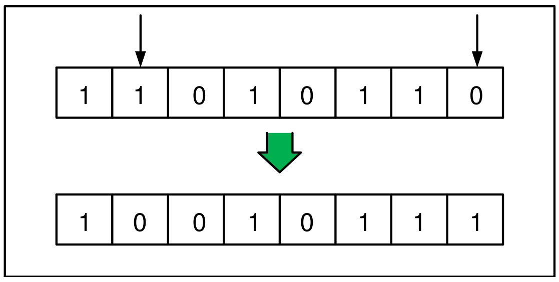

Figure 1.7: *Mutation*

Elitism

The best chromosomes of each generation are called elite members. It is important to keep these chromosomes while discarding the old generation. In order to avoid losing elite members of every generation, we transfer them to next generation without any changes.

Replacement

The aforementioned steps result in appearance of new generation of individuals with new characteristics. The previous generation has to be replaced by the newly generated generation. This process is called replacement.

1.3 Problem Statement

As it is explained before flexibility in FlexCore processor is achieved in two aspects. Flexible interconnect scheme facilitates the exploration of different processors' structures and provides an environment to compare different configuration for a set of applications. Since the power consumption of CMOS circuits is highly dependent on signal activity, the flexible interconnect is an attempt to explore different possible data routing in order to find the most efficient configuration in terms of power. On the other hand a flexible datapath allows inclusion of multiple functional blocks. Additionally, dedicated accelerators such as MAC-units can be made available, to further enhance the flexibility. It is important to consider that addition of dedicated blocks suggests increased area and power dissipation; however, these blocks could execute some functions more quickly than general-purpose blocks which leads to less energy dissipation.

Finding the best hardware configuration among the large number of possible configuration for any specific application is a very complex and time-consuming process. The designer needs to have knowledge both in FlexCore design and application software in order to tune the processor for targeted application.

This project addresses this problem by developing an optimization engine based on genetic algorithm. We hypothesize that the genetic algorithm is an appropriate algorithm to address this optimization problem because of its ability to optimize huge solution spaces. The ultimate goal of the engine is to find the most efficient hardware configuration in terms of energy for the targeted application(s). In addition, the engine potentially could become a very strong tool to explore the FlexCore design space in a more straightforward manner than the previous, manual way.

1.4 Related Work

In [8], Lambrechts *et al.* perform fast simulation and evaluation of coarse-grain reconfigurable architectures using a range of interconnect configurations for functional units. They could identify a subset of interconnect configurations that were associated with higher performance and energy efficiency. The COFFEE compiler and simulator framework [9] was developed to enable fast energy and performance estimations for early exploration of architectures. Mehta *et al.* explore different interconnect configurations for stripe-based reconfigurable fabric architectures, and they presented architecture gains up to 50% reduction on energy, compared to the same architecture with a fully connected interconnect [10]. These methods are limited to coarse-grain reconfigurable architectures of coprocessors, while in this paper we investigate the internal interconnects of a GPP-like datapath.

Other efforts implementing reconfigurable architectures, like the Tensilica Xtensa family of reconfigurable processors [11], leverage the availability of hardware support such as register file size and debug ports to provide design time reconfigurability, while retaining the design philosophy as a traditional RISC processor. In addition, instruction extensions, made available through the design-space exploration (DSE) environment allow the processor to be configured for a specific workload. However, in contrast to the work presented here, reconfigurability is achieved as a function of datapath units alone. The work presented here provides reconfigurability as provided by the Xtensa; in addition interconnect reconfigurability at the datapath unit level is also available.

Stochastic optimization algorithms have previously been applied to the problem of DSE. Ant-colony algorithms were used for mapping and scheduling tasks to a heterogeneous multiprocessor system made up by four processing elements: a DSP, an ARM core, a PowerPC GPP, and an FPGA [12]. Results showed that the approach was 64% better than simulated annealing and 55% better than tabu search. Particle swarm optimization was used to tune a parameterized embedded System-On-Chip architecture to a set of applications and constraints [13]. A system was described by parameters, for example, issue width, number of integer/floating point functional units and cache parameters, with results within 70% of a full search in 1/5 of the time. Genetic algorithms, on a similar problem as above, were targeting a MIPS R3000 processor with 19 parameters resulting within 1% of optimum in 80% less time [14]. In [15] it was noted that genetic algorithms emerged as the best search policy with performance around 0.1% of exhaustive search while being 23,000 times faster when applied to multi-core DSE. All the methods mentioned above target performance as an improvement criterion, while considering the functional units and their parameters as inputs for optimization. In our scheme, however, we consider the interconnect configuration between the datapath units as an input for optimization, while targeting performance as well as power dissipation for improvement.

Chapter 2

Solution

The Optimization Engine (i.e. OE) toolbox is added to the FlexTools environment to find an optimal interconnect configuration in terms of energy. As it is shown in Fig. 2.1, this toolbox is associated with the other two toolboxes and collects essential data to perform optimization on the FlexCore interconnect configuration. In this chapter the entire structure of the OE toolbox is explained in detail.

To evaluate a configuration, we have to go through a time-consuming procedure of compiling the application on the datapath, generating RTL code, performing functional verification, and simulating the datapath to extract cycle count, link utilization, and information on area, timing, power and energy dissipation. In the later stages, high fidelity estimates are obtained via place and route, and then the whole process takes almost 30 minutes on an Intel Core 2 Duo E8400 3.0 GHz with 4 GB RAM running Linux 2.6.18.

In our engine we typically use the power consumption obtained from post synthesize, which takes about 10 minutes, if targeting autocorrelation benchmark, in order to have faster evaluation on each interconnect configuration compared to post place and route power estimation.

Our solution to the optimization engine is mainly based on two methods to reduce the solution space size (for instance a reduction from 2^{90} possible configurations to 2^{36} for autocorrelation benchmark) and then using genetic algorithm to search through the solution space and find an energy efficient configuration. Due to some technical problems in the FlexTools compiler the optimization engine can only consider interconnect configuration at the moment. However the compiler is currently being developed in another project so optimizing the processor with a decoder, and exploring datapath variations are postponed to future.

The decoder size is related to the number of the links as this will affect the N-ISA size. Having fewer number of links shrinks the size of the decoder that consequently results in less power consumption in the decoder. We suspect that using the decoder in the engine results in finding the optimal configurations, in a population of interconnects that tends to have fewer number of links.

FlexTools is run on the linux environment and to interact with the tools, there is a need to develop bash linux scripts. In the first version of the OE the whole program was written in bash linux, however on the next version, we migrated to C++ to have more control over exploiting parallelism.

As mentioned earlier the algorithms used in FlexTools are heuristic, and this usually prevents us from

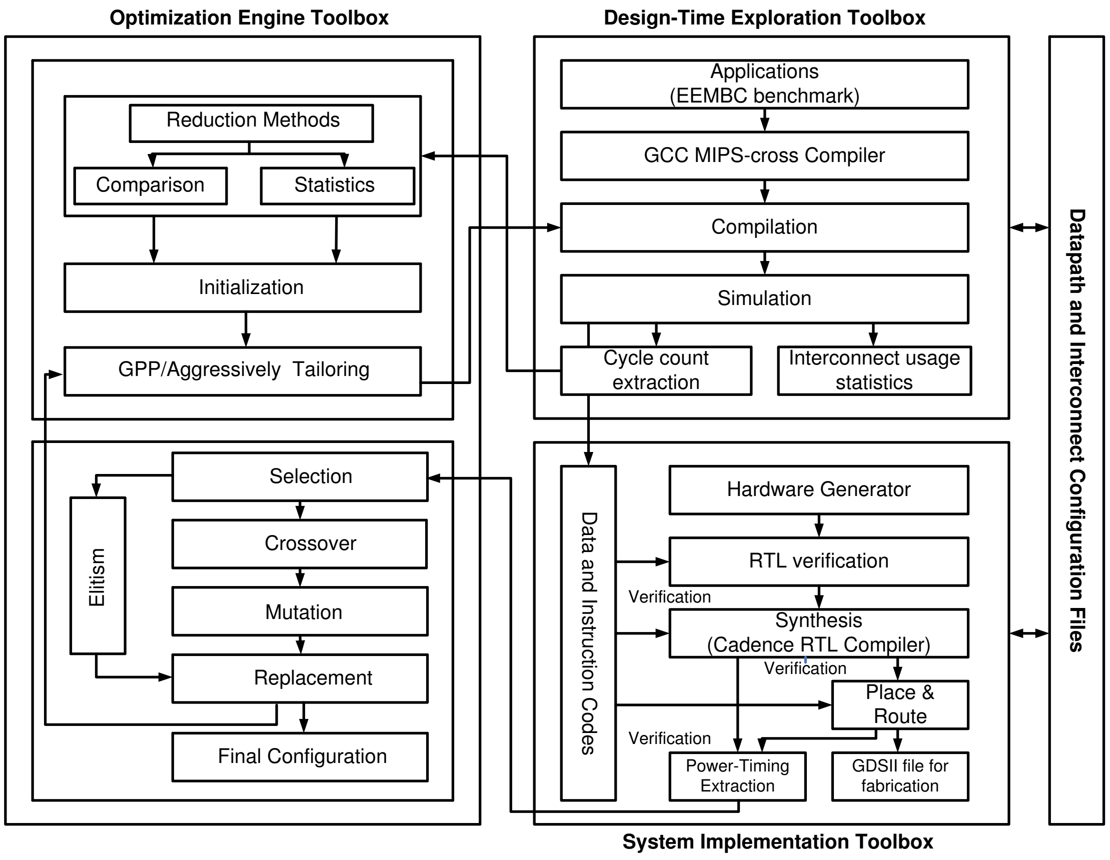
Optimization Engine Toolbox
Design-Time Exploration Toolbox
Reduction Methods
Comparison
Statistics
Initialization
GPP/Aggressively Tailoring
Selection
Elitism
Crossover
Mutation
Replacement
Final Configuration
Applications (EEMBC benchmark)
GCC MIPS-cross Compiler
Compilation
Simulation
Cycle count extraction
Interconnect usage statistics
Data and Instruction Codes
Hardware Generator
RTL verification
Verification
Synthesis (Cadence RTL Compiler)
Verification
Place & Route
Verification
Power-Timing Extraction
GDSII file for fabrication
Datapath and Interconnect Configuration Files
System Implementation Toolbox

Figure 2.1: *The FlexTools environment extended by the OE toolbox.*

predicting the outcome of making slight variations in the interconnect configuration. So using analytical methods to find the optimal configuration were not possible.

2.1 Mapping of Application on Datapath

The 32-bit processor datapath template we consider here (Fig. 2.2) has the units of a MIPS R2000 processor [16], that is, a program counter (PC), a load-store unit (LS), a register file (RF), an arithmetic logic unit (ALU), and two buffers (BUF1 and BUF2). Integer multiplication is common in embedded applications, so a 32-bit multiplier unit (MULT) is also included. In this datapath template, there are nine output ports from the units and ten input ports. Thus, if each output port can communicate with all input ports, there are 90

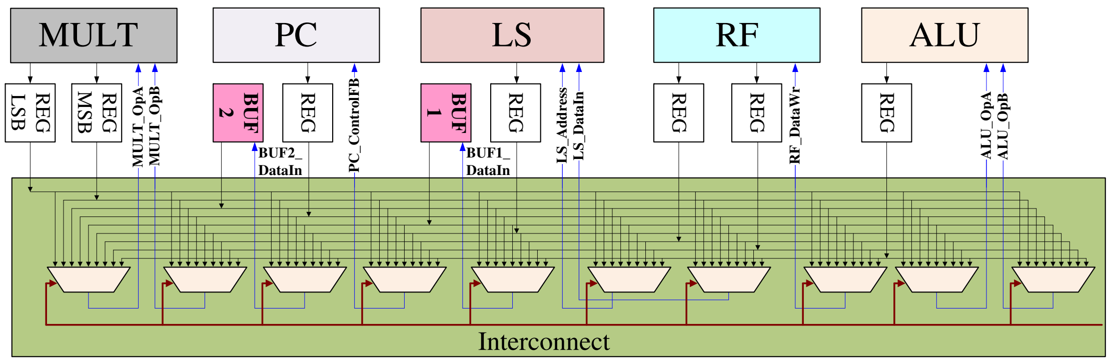

Figure 2.2: *Processor datapath template. The full 90-link interconnect is shown in the switchbox schematic.*

interconnect links altogether; this is called *full interconnect* configuration. Running application code on the full interconnect allows a compiler to freely choose any communication path to obtain a minimal cycle count. Indeed, this full interconnect yields the lowest execution time, however, it is also associated with a significant area and power penalty, since the interconnect multiplexers and the wiring become complex. During the optimization carried out in later sections, initially the full interconnect configuration is assumed, but in the process of fine tuning the datapath to the particular application(s), many interconnect links can be removed.

The interconnect configuration can be represented by a set of bits, where each bit corresponds to a communication path between two specific ports; a ’1’ means a link exists, whereas a ’0’ represents absence of a link. Table 2.1 shows the fields of the interconnect configuration for the datapath in Fig. 2.2.

If we implement the 33 links marked GPP, we can guarantee that the processor will successfully run any application. However, as each specific application has its own characteristics, not all these 33 links are needed; for instance, only 27 out of the 33 links are required to execute the EEMBC autocorrelation benchmark [17] that is used to demonstrate the optimization engine (OE) tool.

The design flow, which maps application code to a datapath, accepts a user-defined datapath and interconnect configuration and the C code of the applications to be executed. Subsequently, a simulation of the application code is performed on the given datapath configuration to obtain statistics on cycle count and interconnect link utilization. Fig. 2.1 shows the design flow, including the OE tool, which is the main contribution

Table 2.1: *Representation of interconnect configuration for the datapath in Fig. 2.2. The 33 links that are labeled GPP represent the GPP-lite configuration and emulate the interconnect of a MIPS R2000.*

	Alu_OpB	Alu_OpA	Regbank_Wr	Ls_Address	Ls_Write	Buf1_Write	Buf2_Write	PC_FB	Mult_OpB	Mult_OpA
Mult_LSW	GPP	GPP	GPP	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Mult_MSW	GPP	GPP	GPP	0/1	0/1	0/1	0/1	0/1	0/1	0/1
PC_ImmPC	GPP	0/1	GPP	0/1	0/1	GPP	0/1	0/1	0/1	GPP
Buf2_Read	GPP	GPP	GPP	GPP	0/1	0/1	0/1	GPP	0/1	0/1
Buf1_Read	0/1	0/1	0/1	0/1	GPP	0/1	GPP	0/1	0/1	0/1
Ls_Read	GPP	GPP	GPP	0/1	0/1	0/1	0/1	GPP	0/1	0/1
Regbank_Out1	0/1	GPP	0/1	0/1	0/1	0/1	GPP	0/1	0/1	GPP
Regbank_Out2	GPP	0/1	0/1	0/1	0/1	GPP	0/1	GPP	GPP	0/1
Alu_Rslt	0/1	GPP	GPP	GPP	0/1	GPP	0/1	GPP	0/1	0/1

of this thesis work. The other parts of the flow perform application scheduling and simulation to extract the application cycle count, which is needed for evaluating performance and energy dissipation, and interconnect usage statistics, to recognize unused or rarely used links. Also, there exists an RTL generator that creates datapath VHDL and a corresponding testbench based on the software application.

2.2 Datapath Energy Optimization

In order to find the solution with the lowest energy dissipation[1] we need to evaluate a large number of solutions (in this case 2^{90} possible configurations). Due to the large number of possible solutions, it is impossible to perform an exhaustive search through the solution space. The constraint of design time, the huge size of the solution space, and the difficulty in formulating the problem of energy minimization as a mathematical function forces us to use evolutionary algorithms instead of deterministic classical optimization methods [18]. In the process of optimizing the interconnect configuration, evolutionary algorithms can find the optimal solutions without evaluating the entire solution space.

The binary string representing the interconnect configuration corresponds to a chromosome in a genetic algorithm. The concept of a genetic algorithm is based on improving the solution space in future generations by inheriting the appropriate features and discarding the improper characteristics of the previous generation. Such algorithms have been proven to be reliable and powerful for many problems [18].

2.2.1 Optimization Engine

The process of identification and removal of the unused links for specific application(s) is called *tailoring*. We compile and simulate the application(s) on a proposed interconnect configuration, after which we extract the

[1]The metric of total energy dissipation is defined in a straightforward manner as power dissipation $\times$ clock period $\times$ cycle count.

link usage statistics. Based on the obtained statistics, we propose a new interconnect configuration that lacks the links that were never used. Tailoring can be categorized as follows:

GPP-tailoring: During the process of tailoring, in order to guarantee that the original GPP functionality is maintained, we have to use the 33 GPP links of Table 2.1.

Aggressive tailoring: If GPP functionality is not necessary, all links that are not essential to the specific application(s) can be removed.

Search Space Reduction Techniques

In order to decrease the execution time of the Optimization Engine, we need to reduce the size of the search space. In this section we present two techniques that we used to reduce the search space size. Also, we provide a framework to allow for compilation of each possible configuration, either for a datapath that supports GPP functionality or one that does not.

Reduction Based on Comparison: In this technique we compare the GPP chromosome, which corresponds to the GPP-lite interconnect configuration (Table 2.1), with a full interconnect configuration chromosome that has been aggressively tailored; the *full-tailored* chromosome. The comparison is done bit by bit and the result based on Table 2.2 is stored in a new chromosome, called the *frame chromosome*, that acts as a frame to produce new chromosomes. The '0' and '1' bits in the frame chromosome remain intact. Moreover, to maintain GPP functionality, 'G' is set to '1'.

Table 2.2: *a) Assigned value in the frame chromosome. b) Reduction based on comparison technique.*

GPP	Full-tailored	Frame	# Links
0	0	0	28
0	1	F	29
1	0	G	9
1	1	1	24

Full-tailored chromosome	01101100111	90 bits
GPP chromosome	10100101001	90 bits
Frame chromosome	1F10F101FF1	90 bits
Compact chromosome	0110	29 bits
Full-size chromosome	10101101101	90 bits

In the example of Table 2.2(a) there are totally 61 bits representing '1', '0', and 'G' that are permanently set to either '1' or '0' in the frame chromosome. The remaining 29 bits are used to form a smaller chromosome called the *compact chromosome*, which is the primary input of our algorithm. During optimization, the algorithm generates different compact chromosomes, which are then split bit by bit, where each bit is placed back in its corresponding empty place (F) in the frame chromosome. The newly created chromosome, called the *full-size chromosome*, is then evaluated to obtain the corresponding configuration's energy dissipation and the result is reported to the algorithm. Table 2.2(b) shows an example of a complete process of extracting the full-size chromosome from a full interconnect configuration that has been tailored for the EEMBC autocorrelation benchmark.

Reduction Based on Statistics: The second technique relies on evaluating several random interconnect configurations. We produce a large number of full-size chromosomes ($\sim$20,000) and go through compilation and simulation steps to extract the utilization statistics of each link. Our investigations into several simulations' results show that for the aforementioned situation, that is, the autocorrelation benchmark operating on the full 90-link configuration, there are 60 bits that always have the same value, either '0' or '1'. These settings leave 30 bits for optimization in the OE tool.

The drawback of this technique is that, although the compilation and logic simulation are relatively fast compared to later, circuit-oriented evaluation steps, performing these steps for a large number of configurations increases the execution time of the engine. The advantage is that it can create a solution space for the engine to optimize the entire interconnect configuration. Since we have statistically set 60 links to '0' and '1' and we are confident that these links always have the same value, optimizing the remaining 30 links could be equal to optimizing all 90 links. After creating the frame and compact chromosomes, the rest of the process is the same as the first technique.

Initialization

The initial step of the algorithm is to provide the first generation with a population of N random chromosomes. In practical genetic algorithms, the typical value for the population size is 30-1,000 [18]. It is important to assign an appropriate population size that largely covers the possible energy range of different interconnect configurations. To find out a suitable population size, different populations of 50, 100, and 3,000 individuals were evaluated to obtain energy dissipation based on post synthesis power estimation[2].

Fig. 2.3 shows the distribution of individuals in different energy ranges for each population. As the figure suggests, for 100 and 3,000 populations the percentage of individuals in different ranges of energy dissipation is almost equal. Furthermore, the difference of the best and the worst energy dissipation in 100 chromosomes covers more than 80% of the population of 3,000 individuals. Based on the results obtained from this diagram, we have assigned a population size of 100 to our algorithm.

To create the initial generation, first a random compact chromosome is created—for instance, 29 random bits for the aforementioned comparison technique, like '1 0101 1000 0001 1001 0110 0110 0111'— and a unique chromosome name, based on its generation and creation order, is assigned to the chromosome. Then its corresponding full-size chromosome is created, as shown in Table 2.2(b).

The procedure is performed N times ($N = 100$) to complete the initialization process. These N chromosomes are evaluated in terms of power dissipation, cycle count and area.

Selection

To produce the next generation, we need to select the chromosomes in pairs for the sexual reproduction. The fitness value is a factor that determines the probability of a chromosome to be selected to produce the next generation.

[2]Using a 90-nm ASIC technology, the autocorrelation benchmark, a clock period of 2500 ps and a nominal temperature of 25 C.

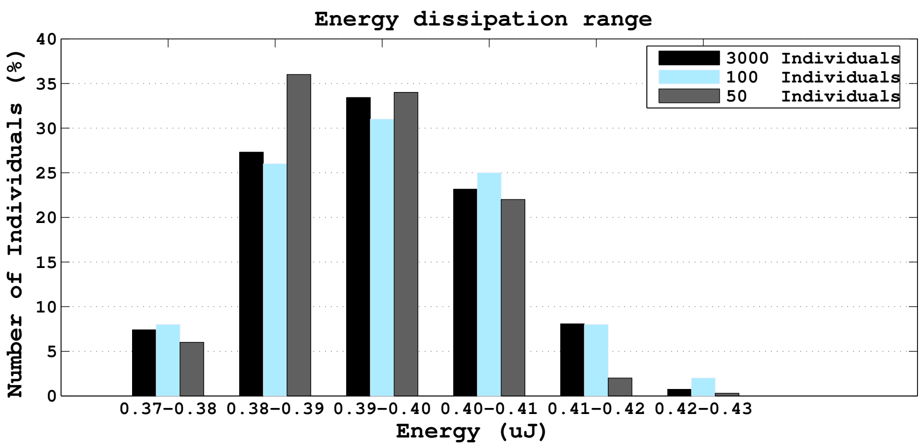

Figure 2.3: *Energy dissipation range.*

Since the energy dissipation of different chromosomes is close to each other, we use a rank selection method to assign fitness function to the chromosomes. Here we first rank the chromosomes based on their energy dissipation and then each chromosome is assigned a fitness value (fv) proportional to its place in the sorted list. Each chromosome corresponds to a range of $\sum_{i=1}^{m-1} fv_m$ to $\sum_{i=1}^{m} fv_m$, where $\sum_{i=1}^{m-1} fv_m$ is the sum of fitness values of chromosomes that take higher places than chromosome m in the sorted list. Finally, a random value is generated and the chromosome that has the random value in its range is chosen. While this method tends to select chromosomes with higher fitness values more often, other chromosomes also have the chance to be selected.

Crossover and Mutation

The crossover process provides an environment for two selected chromosomes to marry and produce offsprings with new characteristics, providing new solutions based on partial inheritance [18]. We have analyzed three major crossover methods; one-point, two-points and uniform methods. The first two methods rely on splitting the pair of chromosomes at one or two points, and swapping the respective segments of the parent chromosomes. In the uniform crossover method, bits of parent chromosomes are often swapped with a probability of 0.5. Our investigation shows that this method tends to produce completely random chromosomes, which makes it inappropriate for our engine, which instead uses one-point or two-points crossover methods. The crossover points are randomly chosen for each pair, to reduce the probability of producing the same chromosomes.

Mutation may change the direction of evolution, as it happens in nature. It may lead to a better or worse offspring and it is the duty of the natural selection process to keep the better and discard the worse. The mutation step is done by changing the state of one randomly selected bit in the chromosome. In our engine, the mutation probability is set to 0.05.

Elitism and Replacement

In order to avoid losing the best individuals during selection and crossover, we directly copy the two best chromosomes, called the *elite members*, to the next generation. These two chromosomes, together with $N - 2$ that come from the previous step, form a new generation. The first population can be discarded and replaced by the N new individuals.

The entire process is repeated on the new generation, until we reach the desired number of generations. The best chromosome in the last generation is the final output of the engine. Based on our experiments, we have set the number of generations to ten. The engine employs parallelism so the total evaluation time scales down with an increasing number of CPUs.

In the next chapter we will present the results that were obtained by running the optimization engine.

Chapter 3

Results

The following results are obtained based on the methods explained in Sec. 2.2.1. The EEMBC autocorrelation and FFT benchmarks are used as driving applications.

3.1 Evaluating 1,000 Random Chromosomes

The result of evaluating 1,000 non-tailored chromosomes and their GPP-tailored counterparts, in a reduced solution space, is shown in Table 3.1. As the table reveals, if a chromosome possesses a higher fitness value, it does not necessarily lead to its tailored version having a better fitness value among other tailored chromosomes. Therefore, in the process of searching for an optimal solution, we need to initialize the population by tailored chromosomes. In this table, *#Links* is the number of interconnect links and *Rank* is the position of the chromosome in the sorted list of chromosomes.

Table 3.1: *Comparing best tailored and non-tailored chromosomes.*

Non-Tailored			Tailored		
Rank	#Links	Energy (μJ)	Rank	#Links	Energy (μJ)
1	47	0.389	28	40	0.383
2	50	0.391	133	43	0.390
70	51	0.399	1	42	0.372
104	50	0.402	2	43	0.377

Table 3.2 shows a number of chromosomes sorted based on their post-place-and-route energy dissipation. Chromosomes with A-T at the end of their names are aggressively tailored and do not necessarily provide GPP functionality; those with GPP-T are GPP-tailored. Config-60 is a 60-link configuration that has been manually optimized by an experienced designer [19]. The table shows that although the ratio of post-synthesis and post-place-and-route energy is not linear, searching for an optimal configuration based on post-synthesis results is satisfactory to the optimization process. This can result in a huge reduction in the execution time of the engine.

Table 3.2: *Comparison of post-synthesis result of a big population.*

Name	#Links	#Cycles	Energy post synthesis (μJ)	Energy post P&R (μJ)
I-000194A-T	42	16111	0.379	0.540
I-002345A-T	43	16133	0.384	0.553
I-000432	47	16226	0.398	0.639
I-000194GPP-T	50	16111	0.397	0.647
I-000004	50	16112	0.406	0.652
Config-60	60	16118	0.403	0.660
I-001267GPP-T	50	16133	0.413	0.663
I-000994	47	16290	0.399	0.665
I-000066	52	17552	0.452	0.716

3.2 Effect of Cycle Count and Power on Energy

Our main focus in the OE is to reduce the energy consumption of the engine, which is obtained by $Power * CycleCount * ClockPeriod$. With a fixed clock period energy is dependent on cycle count and average power consumption. This implies that an energy efficient configuration should have relatively low power and cycle count at the same time. However our experiments do not concur with this idea.

The figures are based on generating 100 random FlexCore configurations and evaluating the Power, Cycle count and consequently Energy dissipation of these configuration running FFT benchmark.

In Fig.3.1 cycle count and power consumption of the chromosomes are plotted versus the energy dissipation. As the figure suggests the chromosomes with lower energy dissipation have lower power consumption and not necessary lower cycle count.

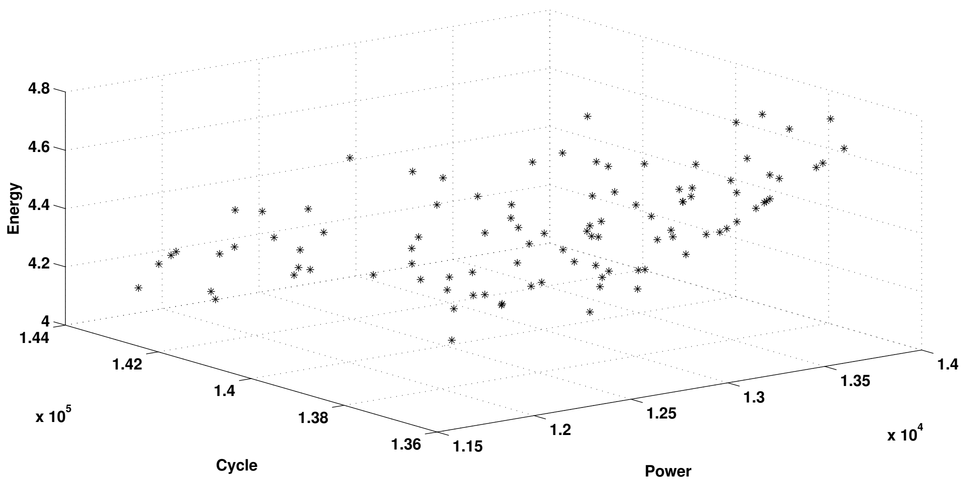

Figure 3.1: *Cycle Count and Power vs. Energy Dissipation of 100 Random chromosomes running FFT benchmark*

Fig.3.2 is the cycle count against energy dissipation of the same chromosomes and as it can clearly be seen the lower cycle count does not result in lower energy dissipation, whereas the situation is quite different in Fig.3.3. A trend that shows the nearly linear relation between power consumption and energy dissipation is visible. Unfortunately the time consuming part of the evaluation is the one related to extracting power consumption, and it is not possible to discard the configurations that are associated with high energy dissipation at early stages of the evaluation.

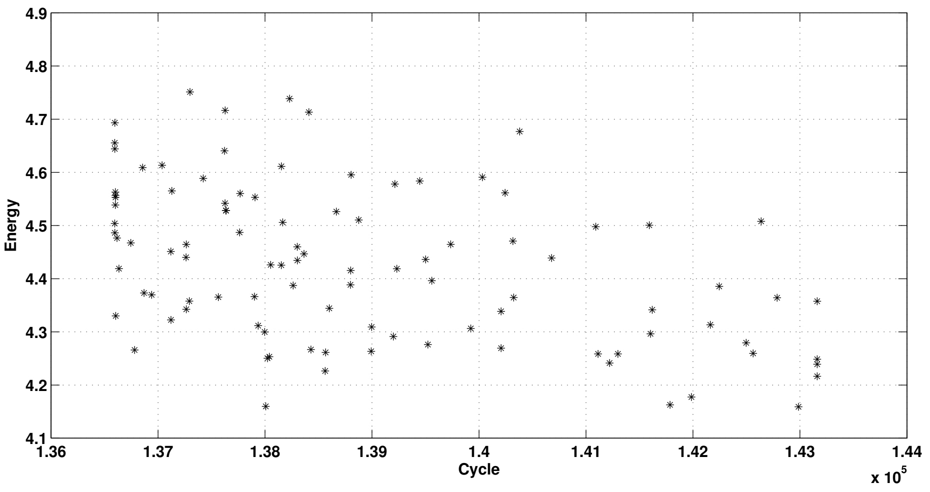

Figure 3.2: *Cycle Count vs. Energy Dissipation of 100 Random chromosomes running FFT benchmark*

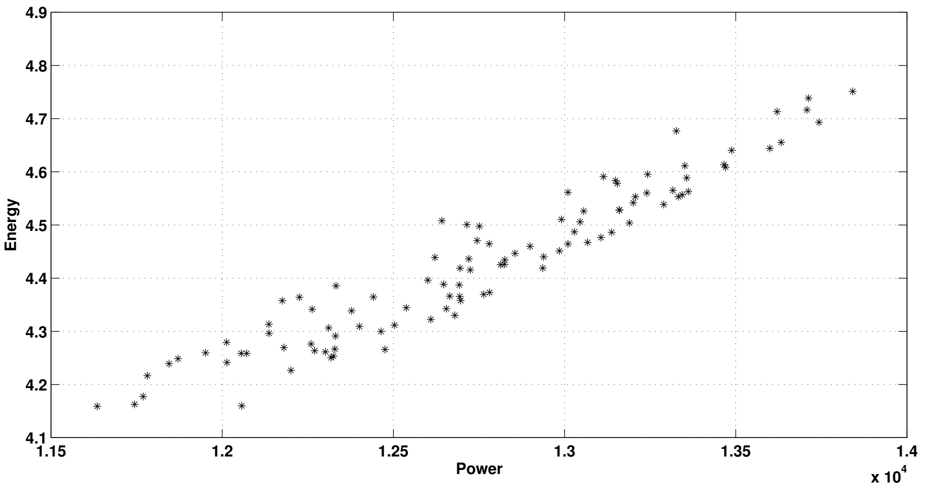

Figure 3.3: *Power vs. Energy Dissipation of 100 Random chromosomes running FFT benchmark*

3.3 Optimization Engine Result

Fig. 3.4 shows the average energy dissipation as well as the elite members of subsequent generations, obtained by running the optimization engine in a reduced solution space. The average energy dissipation for two populations of 100 non-tailored and 100 GPP-tailored random chromosomes, generated in a 2^{90} solution space, are also shown for reference. The effect of tailoring and use of reduction methods to decrease the energy dissipation of the initial population can clearly be seen. Furthermore, applying the genetic algorithm on the subsequent steps of the OE tool contributes largely to this reduction. The result was obtained from a population size of 100 chromosomes, using a clock period of 2500 ps and a 90-nm ASIC technology.

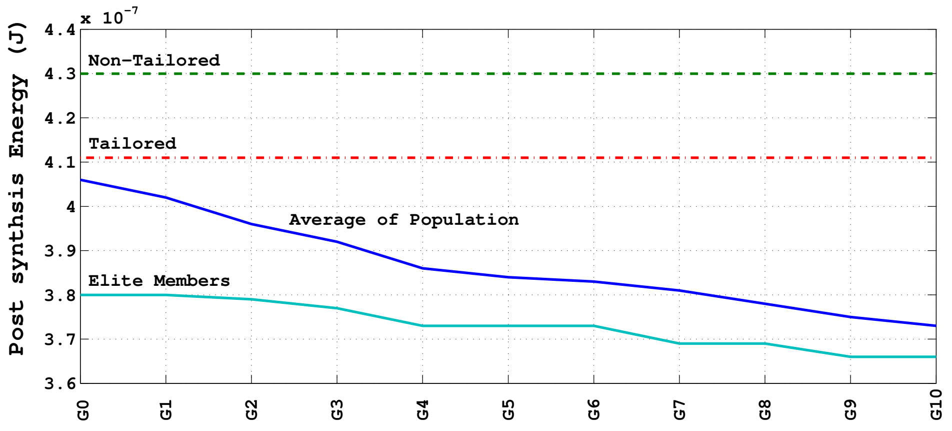

Figure 3.4: *Average and elite members' energy dissipation of subsequent generations of the optimization process.*

In Table 3.3 some major interconnect configurations of the datapath template obtained by the OE tool are sorted based on their post-synthesis energy, to investigate the effect of using a datapath interconnect configuration that has been optimized for a specific application, on a very different application. As an alternative application the EEMBC FFT benchmark was chosen, and it appears that a good configuration for autocorrelation may result in a good configuration for FFT as well. This is one of the major advantages of using the flexible datapath interconnect compared to general-purpose or application-specific implementations: The design has been tailored to a specific application, but still it can take on other applications thanks to the general-purpose functionality that is preserved.

Table 3.3: *Application versatility.*

	Autocor (μJ)	FFT (μJ)
Optimal autocorrelation configuration	0.377	4.226
Elite member (Generation 1)	0.392	4.256
GPP	0.43	4.239
Worst member (Generation 1)	0.432	4.361

3.4 The Effect of Interconnect Configuration

So far the effect of interconnect configuration on the energy dissipation, while maintaining loyalty to GPP, has been investigated. We have also used the reduction method that eliminates the possibility of generating inefficient interconnect configurations. Fig. 3.5 shows several examples of interconnect configurations: HED is a interconnect configuration associated with very high energy dissipation, GPP is the 33-link GPP-lite interconnect, FT is a 53-link interconnect that was tailored to the autocorrelation benchmark, Res-GPP is the GPP-compliant interconnect obtained from the OE tool, while Res-NGPP is a 51-link interconnect that the OE tool found to specifically execute the autocorrelation application. We notice in Fig. 3.5 that, for example, Res-NGPP is 38% more energy efficient than GPP.

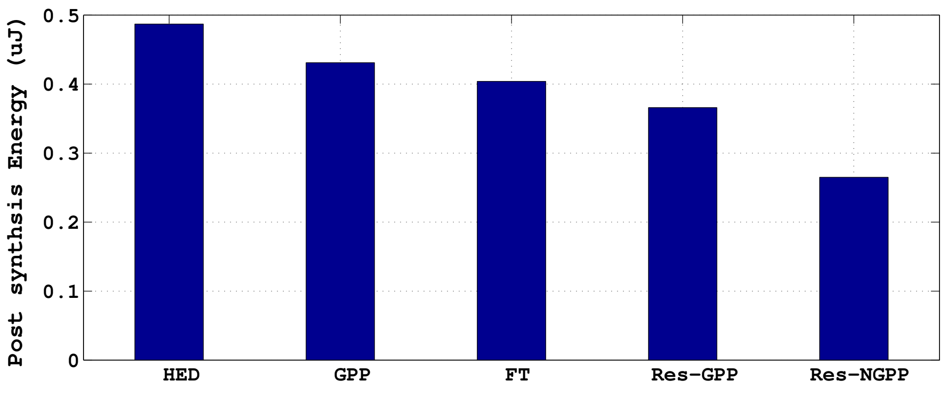

Figure 3.5: *Comparing the energy dissipation of the interconnect configurations.*

Chapter 4

Future Work

The flexibility aspects of FlexCore processor (Interconnect and Datapath flexibility) are explained in previous chapters. Although the ultimate goal of processor optimization is to optimize both interconnect and datapath configuration, the presented results in this project are only based on interconnect optimization and do not include the flexible datapath. It is important to consider that interconnect optimization is prerequisite for a more general optimization scheme. Currently our on-going research deals with an expansion of the optimization engine, to consider a larger set of datapath units for improved performance and to leverage run-time reconfiguration inside the interconnect.

One delimitation chosen in this work is that only the datapath circuitry is considered. The reconfiguration done to the interconnect affects the instruction set too; for example, a larger interconnect requires more control bits, which increases the instruction word length. Simultaneously with this project a flexible scheme for decoding of instructions is being developed.

4.1 DataPath Optimization

Due to the current version of the FlexTools that does not support datapath flexibility, the datapath optimization was not taken into account in the current optimization engine. The current FlexTools version only supports the basic FlexCore processor that is extended by a multiplier.

Our optimization engine is able to perform the datapath optimization as soon as the compiler can support different datapath variations. However we need to conduct some experiments on the engine and evaluate the results to make sure that our scheme is perfectly suitable. The optimization engine's approach towards datapath optimization is presented below:

- Evaluating different datapath variations:

  The number of datapath units is limited and it is possible to extract all different combinations with limited number of the same functional datapath units. In order to find the best configuration among the different combinations of datapath variation, we choose some major interconnect configurations for these combinations and evaluate their energy dissipation. For instance these interconnect configurations can be the full-interconnect configuration, full-tailored configuration and the GPP configuration.

- Choosing best configuration in terms of energy dissipation:

  By having the evaluation's result we can choose the most energy efficient datapath configuration targeting the specified application.

- Performing the interconnect optimization:

  The final step would be applying the interconnect optimization phase to the chosen datapath configuration to fine tune the result of the optimization process.

4.2 Macro-Modeling

Power macro-modeling is a high-level power estimation method that allows the designers to effectively explore and analyze the design in early design stages [20]. However it is worth mentioning that each method of macro-modeling has a different level of accuracy. Since the major amount of execution time of the engine is consumed by power estimation of different configurations, macro-modeling seems to be a promising approach to reduce the overall execution time of the engine.

The energy dissipations of different configurations especially in the final generations of the engine are getting closer to each other. As no macro-modeling technique is 100% accurate, it is possible to incorrectly place different configurations in the sorted list of energy dissipation.

One approach to solve this problem could be evaluating the power dissipation based on macro-modeling in the primary generations of the genetic algorithm where the power dissipations vary significantly. A major advantage of using macro-modeling at early stage is that we can evaluate populations with bigger size and discard the configurations that are associated with high energy dissipation. In the final generations where accuracy is important, we can shrink the populations size and perform accurate circuit-level power estimation to find the optimal solution.

In the macro-modeling methods the power dissipation of each component can be stored either as an equation or in a look-up table, based on the probabilities of the input and/or output signals. There are different equation and table based approaches towards power macro-modeling as mentioned below:

- **3DTab:** The method is based on a look-up 3D table indexed with the signal properties of P_{in}, D_{in}, D_{out}, which are input signal probability, input transition density and output signal probability respectively. In an RTL simulation the signal properties are obtained and then the corresponding power value is extracted from the table. The method is accurate assuming independent input signals [21] [22].

- **EqTab:** This approach is a combination of using equations and tables; the look-up table consists of two elements of P_{in} and D_{in} and then in each entry of the table, coefficients of an equation are stored [21].

- **e-HD**: This method is based on expressing the power as an equation of two inputs of signal properties, the Hamming distance and the number of stable one bits [21].

- **4DTab:** This method is an extension to the 3DTab adding a new dimension of SC_{in}, which is the average spatial correlation coefficient. This parameter adds sufficient accuracy to macro-modeling even for the inputs that are highly correlated. The method has shown to have an average error of about 6% [22].

We have decided to choose the 4DTab macro-modeling approach for our optimization engine. As the interconnect varies for each configuration it is not possible to perform interconnect macro-modeling and our intention is to perform the macro-modeling for the datapath units only, while the interconnect power evaluation will be preformed based on circuit level power estimation techniques.

Chapter 5

Conclusion

A flexible GPP datapath interconnect provides designers with an ability to fine tune the processor targeting specific applications. This work presents an optimization engine that automates the process of finding an energy-optimal interconnect configuration, saving precious design time. The evaluation of exploration of datapath variations is postponed to future projects since the current FlexTools compiler version does not support many of the possible datapath variations.

Since the current engine targets the interconnect configuration, evaluation of the results widely shows the effect of efficient data routing and hardware complexity on power dissipation for any specific application. On the other hand evaluation of large number of possible interconnect configuration demonstrate the fluctuation of energy dissipation versus power dissipation and cycle count. This means that besides the optimization goal of the engine, it is a very strong and reliable tool that provides a comprehensive environment to evaluate and compare different possible configuration of the FlexCore processor and other similar structures.

The flexible interconnect offers parallelism that can be utilized by a compiler, and consequently significant application energy reductions are demonstrated by only rerouting data inside a simple datapath. Although the engine focuses on power evaluation and performs the optimization in terms of energy, it is very straightforward to customize the engine to optimize the processor structure based on other hardware features such as area or performance.

Bibliography

[1] “www.flexsoc.org,” 2010.

[2] John Hughes, Kjell Jeppson, P. Larsson-Edefors, Mary Sheeran, P. Stenström, and LJ Svensson, “FlexSoC: Combining flexibility and efficiency in SoC designs,” in *Proceedings of the IEEE NorChip conference*. 2003, pp. 52–55, Citeseer.

[3] Tung Thanh Hoang, *Customization for an Energy-Efficient Embedded Processor with Flexible Datapath*, Chalmers University of Technology, 2010.

[4] Tung T. Hoang, Ulf Jälmbrant, Erik der Hagopian, Kasyab P. Subramaniyan, Magnus Själander, and Per Larsson-Edefors, “Design space exploration for an embedded processor with flexible datapath interconnect,” in *21st IEEE Int. Conf. on Application-specific Systems Architectures and Processors (ASAP)*, 2010, pp. 55–62.

[5] M. Wahde, *Biologically Inspired Optimization Methods*, WIT Press, Billerica, MA 01821, USA, 2008.

[6] J. H. Holland, “Adaptation in Natural and Artificial Systems, journal = University of Michigan Press, year = 1975,,” .

[7] J E Baker, “Reducing bias and inefficiency in the selection algorithm,” in *Proceedings of the Second International Conference on Genetic Algorithms and their Application ICGA2*, John J Grefenstette, Ed. Cambridge, Massachusetts, United States, 1987, pp. 14–21, Lawrence Erlbaum Associates.

[8] A Lambrechts, P. Raghavan, M. Jayapala, Bingfeng Mei, F. Catthoor, and D. Verkest, “Interconnect exploration for energy versus performance tradeoffs for coarse grained reconfigurable architectures,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 151–155, 2009.

[9] Praveen Raghavan, Murali Jayapala, Andy Lambrechts, Javed Absar, and Francky Catthoor, “Playing the trade-off game: Architecture exploration using COFFEE,” *ACM Trans. on Design Automation of Electronic Systems*, vol. 14, no. 3, pp. 1–37, 2009.

[10] Gayatri Mehta, Justin Stander, Mustafa Baz, Brady Hunsaker, and Alex K. Jones, “Interconnect customization for a hardware fabric,” *ACM Trans. on Design Automation of Electronic Systems*, vol. 14, pp. 11:1–11:32, 2009.

[11] Grant Martin, “Recent developments in configurable and extensible processors,” in *Int. Conf. on Application-specific Systems, Architectures and Processors*, 2006, pp. 39–44.

[12] A. Tumeo, C. Pilato, F. Ferrandi, D. Sciuto, and P. Lanzi, “Ant colony optimization for mapping and scheduling in heterogeneous multiprocessor systems,” in *Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2008, pp. 142–149.

[13] G. Palermo, C. Silvano, and V. Zaccaria, “Discrete particle swarm optimization for multiobjective design space exploration,” in *Euromicro Conf. on Digital System Design*, 2008, pp. 641–644.

[14] M. Palesi and T. Givargis, “Multi-objective design space exploration using genetic algorithms,” in *Int. Symp. on Hardware/Software Codesign*, 2002, pp. 67–72.

[15] S. Kang and R. Kumar, “Magellan: A search and machine learning-based framework for fast multi-core design space exploration and optimization,” in *Design, Automation and Test in Europe*, 2008, pp. 1432–1437.

[16] David A. Patterson and John L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, Morgan Kaufman Publishers Inc., 2nd edition, 1998.

[17] Embedded Microprocessor Benchmark Consortium, ," `http://www.eembc.org`.

[18] M. Wahde, *Biologically Inspired Optimization Methods*, WIT Press, Billerica, MA 01821, USA, 2008.

[19] Tung T. Hoang, Ulf Jälmbrant, Erik der Hagopian, Kasyab P. Subramaniyan, Magnus Själander, and Per Larsson-Edefors, "Design space exploration for an embedded processor with flexible datapath interconnect," in *21st IEEE Int. Conf. on Application-specific Systems Architectures and Processors*, 2010, pp. 55–62.

[20] R Zafalon, M Rossello, E Macii, and M Poncino, "Power macromodeling for a high quality RT-level power estimation," *Proceedings IEEE 2000 First International Symposium on Quality Electronic Design Cat No PR00525*, pp. 59–63.

[21] Felipe Klein, G Araujo, Rodolfo Azevedo, Roberto Leao, and L.C.V. dos Santos, "A multi-model power estimation engine for accuracy optimization," in *Proceedings of the 2007 international symposium on Low power electronics and design*. 2007, pp. 280–285, ACM New York, NY, USA.

[22] S. Gupta and F.N. Najm, "Power modeling for high-level power estimation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 1, pp. 18–29, 2000.

Part II

APPENDICES

Chapter 6

Appendix I

6.1 Optimization Engine Main Body

```
================================================================
Name : Optimization Engine
Author : Babak Hidaji, Salar Alipour
Version : Beta
Copyright :
Description : This is the main body of the optimization Engine program that uses genetic algorithm and some
solution space reduction methods to find an energy optimal interconnect configuration for a FlexCore proces-
sor targeting a specific application.
================================================================
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
using namespace std;
#include "FlexToolsFunctions/runFlexToolsThread.hpp"
#include "DataExtractFunctions/DataExtract.hpp"
#include "DataTypes.hpp"
#include "GeneticAlgorithmFunctions/RandomChromosomeGenerator.hpp"
#include "ChromosomeSizeReduction/ChromComparision.hpp"
#include "ProgramFunctions/CompactToFullsize.hpp"
#include "ChromosomeSizeReduction/FrameChromGenerator.hpp"
#include "GeneticAlgorithmFunctions/Crossover.hpp"
#include "GeneticAlgorithmFunctions/Mutation.hpp"
```

```
#include "GeneticAlgorithmFunctions/Selection.hpp"
#include "ProgramFunctions/PrintChrom.hpp"
#include "ProgramFunctions/ChromExistanceCheck.hpp"
#include "ProgramFunctions/StringToChrom.hpp"
#include "ProgramFunctions/LinkConnectionFromCycleFile.hpp"
#include "ProgramFunctions/SetAndObtainParameters.hpp"
#include "ProgramFunctions/AvgEliteWorst.hpp"

int main() {
```

Seeding the random number generator

```
srand((unsigned)time(0));
```

Defining the Pool of Chromosomes

```
vector <chrom_rectype> ChromPool, NextGeneration;
pthread_t thread[MAXTHREAD];
int iret;
```

Setup thread attributes

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

Defining temporary chromosomes needed for selection crossover and mutation steps

```
chrom_rectype Father, Mother, Son, Daughter;
chrom_rectype FrameChrom,BaseChrom, gppChrom, fullChrom;
```

Defining the engine parameters

```
int numberOfThreads, Base, Population, Generation, NoOfEliteMembers, MutaionProbability, NumberOf-
Mutations, ChromPoolForFrameSize;
crossoverType CsOM;
FrameGenerateType FgenM;
```

This function receives some parameters required for the engine from the users and also predefines some of them, and put them into the base chromosome which is used later on to generate all the chromosome

```
SetAndObtainParameters(BaseChrom, numberOfThreads, Base, CsOM, FgenM, Population, Generation, NoOfE-
liteMembers, MutaionProbability, NumberOfMutations, ChromPoolForFrameSize);

char ChromName[10] ;
```

Initializing the Frame GPP and Fully connected chromosomes

FrameChrom=BaseChrom;
gppChrom=BaseChrom;
fullChrom=BaseChrom;

The following functions produces the Frame chromosome based on either comparison or the statistic methods. It also generates a body for the Compact chromosome
FrameChromGenerator(FrameChrom, gppChrom, fullChrom, FgenM, Base, numberOfThreads, ChromPoolForFrameSize);

First Phase (Initialization)

For the first step we put boundary chromosomes on test
gppChrom.mode=BaseChrom.mode;
fullChrom.mode=BaseChrom.mode;
ChromPool.push_back(gppChrom);
ChromPool.push_back(fullChrom);

The ChromExistanceCheck Function produces a list of chromsomes that have already been on evaluation in order to avoid generating the same chromosomes in later stages of the engine
ChromExistanceCheck(fullChrom);
ChromExistanceCheck(gppChrom);

for (int i=2 ; i <Population ;i ++) {

Initializing the chromosome's body
ChromPool.push_back(BaseChrom);

Random generation
produce a random compact-size chromosome, while avoiding production of the same chromosomes
while (true) {
RandomChromosomeGenerator(ChromPool[i].CompactChromosome,
FrameChrom.CompactChromosome);
if (ChromExistanceCheck(ChromPool[i])) break; }

Expansion
Expand the compact chromosome to fit the desired chromosome type using the Frame Chromosome
CompactToFullsize(FrameChrom.FullChromosome ,
ChromPool[i].CompactChromosome, ChromPool[i].FullChromosome);

Naming the chromosome based on specific order of creation and the number of its generation

```
sprintf(ChromName,"%d", Base + i);
ChromPool.at(i).Name= "I-" + string(ChromName);

}
```

Evaluation and DataExtraction

The runflextoolsThread is the evaluation function of the engine and takes relatively a very long time so it is designed run parallel and evaluating a number of chromosomes based on the number of the CPU cores, which is defined by the user.

```
for (int i=0 ; i <Population ;i += numberOfThreads) {
```

Evaluating the chromosome through flexTools steps

```
for (int j=0 ; j <numberOfThreads ;j++) {
ChromPool.at(i+j).LoyalityChromosome=gppChrom.FullChromosome;
iret=pthread_create(&thread[j], &attr, runflextoolsThread,
(void *)&ChromPool.at(i+j));
}
```

Waiting for all threads to arrive

```
for (int j=0 ; j <numberOfThreads ;j++) {
pthread_join(thread[j], NULL);
}
```

Obtaining results

The DataExtarct function would extract all the data associated to the specific chromosome that were generated during the evaluation phase

```
for (int j=0 ; j <numberOfThreads ;j++) {
DataExtract(ChromPool.at(i+j),true , BaseChrom.mode);
}

}
```

Removing incomplete evaluations, in case there were problems with the evaluating a specific chromosome that may happen due to several reasons.

```
int count=0;
while (true) {
if (ChromPool.at(count).EnergySynVCD==0) {
ChromPool.erase(ChromPool.begin()+count);
count- -;
}
count++;
if (count==ChromPool.size()) { break;}
}
```

Storing the results of the initial phase in a file This functions stores and sorts all the results of a generation in file and also obtains average, the best member, and the worst member of the generation.

```
AvgEliteWorst("results/Initial",ChromPool);
```

End of the first phase (Initialitialization)

Second phase (Next Generations)

```
for (int g=0 ; g <Generation ;g++ ) {

NextGeneration.clear();

char GN[2];
sprintf(GN,"%d", g);
string GenerationNumber(GN);
```

Initializing the chromosome pool of the next generation

```
for (int i=0 ; i< Population+2; i++) {
```

Initializing the element in the vector

```
NextGeneration.push_back(BaseChrom);
}
```

Choosing the elite members and putting the on the next generation

The selection function consists of several functionalities based on the function's mode. In this mode the function chooses the best chromosomes of the previous pool based on the energy consumptions

```
Selection (ChromPool, NextGeneration[0], NextGeneration[1], Elitism , LinearRanking, 2);

for (int i=2 ; i<Population +2 ; i +=2) {

Father.CompactChromosome.clear();
Mother.CompactChromosome.clear();
Son.CompactChromosome.clear();
Daughter.CompactChromosome.clear();
```

Selecting two individual chromosomes and storing them in Father and Mother. To give the chance for better chromosomes to be chosen more often sp (i.e. the last input of the function),should be bigger than 1 and equal or smaller than 2

```
Selection (ChromPool, Father ,Mother , RouletteWheel, LinearRanking,2);
```

The printChrome function prints the Chromosomes in the output console

```
PrintChrom(Father.CompactChromosome);
PrintChrom(Mother.CompactChromosome);
```

The CrossOver produces two new chromosomes Son and Daughter from the mother and Father, by One-Point, Two-Point or Uniform methods. The Crossover method (CsOm) is selected by user.

```
Crossover(Father,Mother,Son,Daughter,CsOM);
```

Mutation is changing the state of bit in the chromosomes string.
The number of the mutating bits and also the probability of the mutations are predefined in the engine. The "while" loop makes sure that such chromosome was not created before and mutates the chromosome until a new chromosome is generated

```
while(true) {
Mutation(Son, NextGeneration[i], MutaionProbability, NumberOfMutations);
if(ChromExistanceCheck(NextGeneration.at(i))) break;
}

while(true) {
Mutation(Daughter, NextGeneration[i+1], MutaionProbability, NumberOfMutations);
if(ChromExistanceCheck(NextGeneration.at(i+1))) break;
}
```

Filling the newly created chromosomes with the necessary information from the base chromosome

```
for (int j=0; j<2 ; j++) {
sprintf(ChromName,"%d", Base + i +j);
NextGeneration[i+j].Name= "G" + GenerationNumber + "-" + string(ChromName);
NextGeneration[i+j].MainDirectory=BaseChrom.MainDirectory;
NextGeneration[i+j].bm = BaseChrom.bm;
NextGeneration[i+j].stm = BaseChrom.stm;
NextGeneration[i+j].clockperiod=BaseChrom.clockperiod;
NextGeneration[i+j].MainDirectory=BaseChrom.MainDirectory;
NextGeneration[i+j].mode=BaseChrom.mode;
NextGeneration[i+j].Tailored=BaseChrom.Tailored;
NextGeneration[i+j].Elite=BaseChrom.Elite;
NextGeneration[i+j].LoyalityChromosome=gppChrom.FullChromosome;
NextGeneration[i+j].GppLoyality=BaseChrom.GppLoyality;
}
```

Expansion

Expand the compact chromosome to fit the desired chromosome type using the Frame Chromosome

```
CompactToFullsize(FrameChrom.FullChromosome ,
NextGeneration.at(i).CompactChromosome, NextGeneration.at(i).FullChromosome);
CompactToFullsize(FrameChrom.FullChromosome ,
NextGeneration.at(i+1).CompactChromosome,
NextGeneration.at(i+1).FullChromosome);

}
```

Discarding the previous population and replacing it with the new generation

```
ChromPool.clear();
ChromPool=NextGeneration;
NextGeneration.clear();
```

Extracting elite members' data

```
ChromPool.at(0).Elite=true;
ChromPool.at(1).Elite=true;
DataExtract(ChromPool.at(0), false , BaseChrom.mode);
DataExtract(ChromPool.at(1), false , BaseChrom.mode);
```

Starts from 2 cause elite members have already been evaluated

```
for (int i=2 ; i <Population+2 ;i += numberOfThreads) {
```

Evaluating the chromosome through flexTools steps

```
for (int j=0 ; j <numberOfThreads ;j++) {
iret=pthread_create(&thread[j], &attr, runflextoolsThread, (void *)&ChromPool.at(i+j));
}
```

Waiting for all threads to arrive

```
for (int j=0 ; j <numberOfThreads ;j++) {
pthread_join(thread[j], NULL);
}
```

Obtaining results

```
for (int j=0 ; j <numberOfThreads ;j++) {
DataExtract(ChromPool.at(i+j), true , BaseChrom.mode);
}
}
```

Removing incomplete evaluations

```
count=0;
while (true)
if (ChromPool.at(count).EnergySynVCD==0) {
```

```
ChromPool.erase(ChromPool.begin()+count);
count- -;
}
count++;
if (count==ChromPool.size()) { break;}

}
```

Storing and sorting the results of this generation in a file

```
AvgEliteWorst("results/Generation-" + GenerationNumber ,ChromPool);

}

return 0;
}
=================================================================
```

6.2 Chromosome Comparison

Description : This function receives two aggressively tailored chromosomes a fully connected and a gpp-lite and then compares these chromosomes and produces a frame chromosome.

```
    #include "../DataTypes.hpp"
#include "../ProgramFunctions/LinkConnectionFromCycleFile.hpp"
#include "../FlexToolsFunctions/runFlexTools.hpp"
using namespace std;
void ChromComparision (vector<gene>& gppTailoredChrom,vector<gene>& fullTailoredChrom, chrom_rectype&
frameChrom ){
```

providing the frame chromosome

```
for (int i = 0; i<gppTailoredChrom.size(); i++) {
```

If they are the same both C or D make frame the same

```
if (fullTailoredChrom.at(i)==gppTailoredChrom.at(i))
frameChrom.FullChromosome.push_back(fullTailoredChrom.at(i));
else {
```

else if gpp is D that means F is connected make frame as F

```
if (gppTailoredChrom.at(i)==D){
frameChrom.FullChromosome.push_back(F);
frameChrom.CompactChromosome.push_back(F);}
```

else if gpp is C that means F is disconnected make frame as G or (C for now)

```
else frameChrom.FullChromosome.push_back(C);
}}}
=============================================================
```

6.3 Frame Chromosome Generator

Description : This Program generates gpp and full connected chromosomes, and then based on either statistic or the comparison methods, it produces a frame and a compact chromosome.

```
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include "../DataTypes.hpp"
#include "csvFileToChrom.hpp"
#include "../FlexToolsFunctions/runFlexTools.hpp"
#include "../FlexToolsFunctions/runFlexToolsThread.hpp"
#include "../ProgramFunctions/LinkConnectionFromCycleFile.hpp"
#include "ChromComparision.hpp"
#include "../ProgramFunctions/PrintChrom.hpp"
#include "../GeneticAlgorithmFunctions/RandomChromosomeGenerator.hpp"
#include "../ProgramFunctions/ChromXor.hpp"
#include "../DataExtractFunctions/DataExtract.hpp"
#include "../ProgramFunctions/StringToChrom.hpp"

void FrameChromGenerator ( chrom_rectype& FrameChrom, chrom_rectype& gppChrom,chrom_rectype&
fullChrom , FrameGenerateType FGT, int& Base, int& numberOfThreads, int& ChromPoolForFrameSize){
vector <chrom_rectype> ChromPoolForFrame;
```

Names and toolSteps modes are different for these chromosomes

```
gppChrom.Name= "GPP";
gppChrom.mode=compileSimulation;
fullChrom.Name= "FullyInterconnect";
fullChrom.mode=compileSimulation;
FrameChrom.Name= "Frame";
FrameChrom.mode=compileSimulation;
```

Producing the gpp chromosome:
The csvFileChrom function transforms csv files t(interconnect configuration files in FlexTools) to chromosomes.

```
csvFileToChrom (FrameChrom.MainDirectory+"gpp_interconnect.csv",
gppChrom.FullChromosome );
```

Making the full connected chromosome, the size of full.chrom and frame chromosome are determined by the size of gppChrom, so when using the various explorations of the data path. We need to take the expanded version of gpp interconnect into account.

```
for (int i=0; i<gppChrom.FullChromosome.size() ; i++) {
fullChrom.FullChromosome.push_back(C);
}
```

Running gpp and full interconnect chromosome to obtain the usage statistics, it will be only through compile and simulation phase.

```
runflextools(gppChrom);
runflextools(fullChrom);
```

This function extracts usage statistics of the chromosomes after they have been evaluated from the cylce file. If a specific link was never used 0 is put in its place otherwise 1.

```
LinkConnectionFromCycleFile (gppChrom.MainDirectory + gppChrom.Name + "/65nm-sim/soft/autcor.cycle",
gppChrom.FullChromosomeTailored);
LinkConnectionFromCycleFile (fullChrom.MainDirectory + fullChrom.Name + "/65nm-sim/soft/autcor.cycle"
,fullChrom.FullChromosomeTailored);
if (FGT==Comparison) {
```

This function takes the gpp and full tailored chromosomes and based on the comparison method produces the frame chromosome.

```
ChromComparision( gppChrom.FullChromosomeTailored,
fullChrom.FullChromosomeTailored, FrameChrom);
}
else if (FGT==Statistics){
ChromPoolForFrame.push_back(gppChrom);
ChromPoolForFrame.push_back(fullChrom);
vector <gene> temp;
vector <gene> tempGPPed;

pthread_t thread[MAXTHREAD];
```

```
int iret;

    Setup thread attributes
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
for (int i=2 ; i < ChromPoolForFrameSize ;i++) {
temp.clear();

    To initialize a chromosome in the pool
ChromPoolForFrame.push_back(FrameChrom);

    Produce a temp chromosome with gpp full-chrom size

    RandomChromosomeGenerator( temp, gppChrom.FullChromosome); if (gppChrom.GppLoyality) {

    Guaranteeing the compilation
tempGPPed=ChromXor( temp ,gppChrom.FullChromosome);}
else {

    Guaranteeing the compilation

    tempGPPed=ChromXor( temp ,gppChrom.FullChromosomeTailored ); }

ChromPoolForFrame[i].FullChromosome=tempGPPed;
}

    Naming The chromosomes
char ChromName[10] ;
for (int i=0 ; i < ChromPoolForFrameSize ;i++) {
sprintf(ChromName,"%d", Base + i );
ChromPoolForFrame.at(i).Name= "F-" + string(ChromName);
}

for (int i=0 ; i <ChromPoolForFrame.size() ;i += numberOfThreads) {
```

Evaluating the chromosome through flexTools

```
for (int j=0 ; j <numberOfThreads ;j++) {
ChromPoolForFrame[j+i].LoyalityChromosome=gppChrom.FullChromosome;
iret=pthread_create(&thread[j], &attr, runflextoolsThread, (void *)&ChromPoolForFrame.at(i+j));
}
```

Waiting for all threads to arrive

```
for (int j=0 ; j <numberOfThreads ;j++) {
pthread_join(thread[j], NULL);
}
```

Obtaining results

```
for (int j=0 ; j <numberOfThreads ;j++) {
DataExtract(ChromPoolForFrame.at(i+j),true , compileSimulation); //*
ChromPoolForFrame.at(i+j).FullChromosomeTailored.clear();
LinkConnectionFromCycleFile (ChromPoolForFrame.at(i+j).MainDirectory + "/results/"
+ ChromPoolForFrame.at(i+j).Name + "/"+ ChromPoolForFrame.at(i+j).bm + ".cycle", ChromPoolForFrame.at(i+j).FullChromosomeTailo
if (gppChrom.GppLoyality) {
ChromPoolForFrame.at(i+j).FullChromosomeTailored=ChromXor( ChromPoolForFrame.at(i+j).FullChromosomeTailored
,gppChrom.FullChromosome); // Guaranteeing the compilation
} } }
for (int i=0 ; i < ChromPoolForFrameSize ;i++) {
PrintChrom(ChromPoolForFrame.at(i).FullChromosomeTailored);
}
```

```
vector<int> sum;
```

initializing the sum vector

```
for (int i=0 ; i < gppChrom.FullChromosome.size() ;i++) {
sum.push_back(0);
} for (int i=0 ; i < ChromPoolForFrameSize ;i++) {
for (int H=0 ; H < gppChrom.FullChromosome.size() ;H++) {
sum[H]=sum[H]+ ChromPoolForFrame[i].FullChromosomeTailored.at(H);
}
}
for (int H=0 ; H < gppChrom.FullChromosome.size() ;H++) {
cout << sum[H] << " ";
}
cout <<gppChrom.FullChromosome.size()<<endl;
```

```
for (int i = 0; i<gppChrom.FullChromosome.size(); i++) {
if (gppChrom.FullChromosome.at(i)==C and gppChrom.GppLoyality==true){
FrameChrom.FullChromosome.push_back(C);}
else if (gppChrom.FullChromosomeTailored.at(i)==C and
gppChrom.GppLoyality==false){
FrameChrom.FullChromosome.push_back(C);}
else if (sum.at(i)==0) {// if it is equal to the maximum
FrameChrom.FullChromosome.push_back(D);
}
else {
FrameChrom.FullChromosome.push_back(F);
FrameChrom.CompactChromosome.push_back(F);}
}
PrintChrom(FrameChrom.CompactChromosome);
PrintChrom(FrameChrom.FullChromosome);

} else if (FGT==Test) {
```

This option is to test the function. The frame chromosome here was generated for FFT benchmark on statistics method after evaluation 25000 random chromosomes.

```
FrameChrom.FullChromosome=StringToChrom("30033001110000000111100313
31313310331111333101333330133331111331333313311131333310013131113");
FrameChrom.CompactChromosome=StringToChrom("3333333333333333333333
33333333333333333");

    }
```

Making The compact Chromosomes for GPP and Full interconnect

```
gppChrom.CompactChromosome.clear();
fullChrom.CompactChromosome.clear();
for (int i=0; i< FrameChrom.FullChromosome.size(); i++) {
if (FrameChrom.FullChromosome.at(i)==F) {
gppChrom.CompactChromosome.push_back(gppChrom.FullChromosome.at(i));
fullChrom.CompactChromosome.push_back(fullChrom.FullChromosome.at(i));
}}
```

Showing the gpp chromosome and its size

```
cout << "GPP Chromosome Size: " << gppChrom.FullChromosome.size()<<endl;
cout << "GPP Chromosome : ";
PrintChrom(gppChrom.FullChromosome);
```

Showing the full chromosome:

```
cout << "Full Chromosome : ";
PrintChrom(fullChrom.FullChromosome);
```

Showing the gpp tailored chromosome:

```
cout << "GPP Tailored : ";
PrintChrom(gppChrom.FullChromosomeTailored);
```

Showing the full tailored chromosome:

```
cout << "Full Tailored : ";
PrintChrom(fullChrom.FullChromosomeTailored);
```

Showing the full tailored chromosome and its size:

```
cout << "compact Chromosome Size: " ;
cout << FrameChrom.CompactChromosome.size()<<endl;
cout << "Frame Chromosome :";
PrintChrom(FrameChrom.FullChromosome);
```

Saving the results in a file:

```
ofstream FrameBankf;
string FrameBank(gppChrom.MainDirectory + "results/FrameBank");
FrameBankf.open (FrameBank.c_str(), ofstream::app );
FrameBankf << "GPP Chromosome Size: " ;
FrameBankf << gppChrom.FullChromosome.size()<<endl;
FrameBankf << "GPP Chromosome : ";

for (int i=0; i< gppChrom.FullChromosome.size() ; i++) {
FrameBankf<<gppChrom.FullChromosome.at(i);
}
FrameBankf<<endl;
```

Showing the full chromosome:

```
FrameBankf << "Full Chromosome : ";
for (int i=0; i< fullChrom.FullChromosome.size() ; i++) {
FrameBankf<<fullChrom.FullChromosome.at(i);
}
FrameBankf<<endl;
```

Showing the gpp tailored chromosome

```
FrameBankf << "GPP Tailored : ";
for (int i=0; i< gppChrom.FullChromosomeTailored.size() ; i++) {
FrameBankf<<gppChrom.FullChromosomeTailored.at(i);
}
FrameBankf<<endl;
```

Showing the full tailored chromosome

```
FrameBankf << "Full Tailored : ";
for (int i=0; i< fullChrom.FullChromosomeTailored.size() ; i++) {
FrameBankf<<fullChrom.FullChromosomeTailored.at(i);
}
FrameBankf<<endl;
```

Showing the full tailored chromosome and its size

```
FrameBankf << "compact Chromosome Size: ";
FrameBankf << FrameChrom.CompactChromosome.size()<<endl;
FrameBankf << "Frame Chromosome :";
for (int i=0; i< FrameChrom.FullChromosome.size() ; i++) {
FrameBankf<<FrameChrom.FullChromosome.at(i);
}
FrameBankf<<endl;

}
==========================================================
```

6.4 Bool Operator

This function changes the < operator. So that now if we use the sort function the chromosomes are sorted based on their Post Synthesis Energy dissipation

```
#include "../DataTypes.hpp"
bool operator<(const chrom_rectype& a, const chrom_rectype& b) {
return a.EnergySynVCD < b.EnergySynVCD;
}
==========================================================
```

6.5 Mutation

```
#include "../DataTypes.hpp"

void Mutation (chrom_rectype& father, chrom_rectype& son,int MutateProbability, int NumberOfMutations){

son=father;
int Mutate=rand() % 100;

if (Mutate < MutateProbability) {

son.CompactChromosome.clear();
```

This vector keeps the places of mutations

```
vector <int> mutatePlace;

for (int i=0; i< NumberOfMutations ; i++) {
mutatePlace.push_back(rand() % father.CompactChromosome.size());
}

sort(mutatePlace.begin(),mutatePlace.end());

int mutatePlaceCount=0;

for (int i=0; i< father.CompactChromosome.size(); i++) {

if(mutatePlaceCount < NumberOfMutations and i==mutatePlace.at(mutatePlaceCount)) {

if (father.CompactChromosome.at(i)==D) {
son.CompactChromosome.push_back(C);
}
else{
son.CompactChromosome.push_back(D);
}

while(true){

if(mutatePlaceCount>NumberOfMutations-2) break;

if (mutatePlace.at(mutatePlaceCount)== mutatePlace.at(mutatePlaceCount+1)) {
```

```
mutatePlaceCount++;
} else break;
}
mutatePlaceCount++;

}
else {
son.CompactChromosome.push_back(father.CompactChromosome.at(i));
}
}
}
}
==========================================================
```

6.6 CrossOver

```
#include "../DataTypes.hpp"

void Crossover (chrom_rectype& father,chrom_rectype& mother, chrom_rectype& son,chrom_rectype& daughter, crossoverType COT){

int RandNumber1,RandNumber2,RandUniform;
int UniformProbability=50;
```

Creating the first crossover point. This is a point randomly chosen from the first element until the one before the last element

```
RandNumber1=rand() % (father.CompactChromosome.size());
```

Creating the second crossover point. This point is from right after the first crossover point until the last element

```
RandNumber2=rand() % (father.CompactChromosome.size()-RandNumber1) +RandNumber1 +1 ;

son.CompactChromosome.clear();
daughter.CompactChromosome.size();

if (COT==OnePoint or COT==TwoPoint ) {
```

The one-Point and two-point crossover procedures are the same with the difference that in one point the second cross over point is on the last element

```
if(COT==OnePoint) {
```

```
RandNumber2=father.CompactChromosome.size();
}

for (int i=0 ; i< father.CompactChromosome.size() ; i++) {

if (i<RandNumber1 or i>=RandNumber2) {
son.CompactChromosome.push_back(father.CompactChromosome.at(i));
daughter.CompactChromosome.push_back(mother.CompactChromosome.at(i));
}
else
{
son.CompactChromosome.push_back(mother.CompactChromosome.at(i));
daughter.CompactChromosome.push_back(father.CompactChromosome.at(i));
}

}
```

The uniform CrossOver procedure, COT==Uniform

Uniform Probability is set to 50 so the probability of choosing the bits from each chromosome is the same.

```
} else {

for (int i=0 ; i< father.CompactChromosome.size() ; i++) {
RandUniform=rand() % 100;
if (RandUniform < UniformProbability) {
son.CompactChromosome.push_back(father.CompactChromosome.at(i));
daughter.CompactChromosome.push_back(mother.CompactChromosome.at(i));
}else {
son.CompactChromosome.push_back(mother.CompactChromosome.at(i));
daughter.CompactChromosome.push_back(father.CompactChromosome.at(i));
}
}
}
}
==========================================================
```

6.7 Run FlexTools

```
#include <time.h>
#include <iostream>
#include <stdio.h>
#include <fstream>
#include <stdlib.h>
#include <list>
#include "../DataTypes.hpp"
#include "../ChromosomeSizeReduction/ChromToCsv.hpp"
#include "../ProgramFunctions/LinkConnectionFromCycleFile.hpp"
#include "../ProgramFunctions/PrintChrom.hpp"
#include "../ProgramFunctions/ChromXor.hpp"
#include "../ProgramFunctions/StringToChrom.hpp"
using namespace std;

void * runflextoolsThread(void * chrom) {

chrom_rectype *local_chrom = (chrom_rectype *) chrom;

vector <gene> TempChrom;

string path(local_chrom->MainDirectory + local_chrom->Name);
string command;

Making the work Directory and providing the flex.ini file
string mkdirAndflexini( "mkdir -p " + path +
" && cd " + path +
" && source  /run" +
" && generatFlexCore.php"
);

cout << local_chrom->Name << " Is being evaluated" <<endl;

cout << mkdirAndflexini << endl;
system (mkdirAndflexini.c_str());

cout << "FullChromosome: "<<endl;
PrintChrom(local_chrom->FullChromosome);
```

This function takes a sample interconnect.csv file together with a chromosome and then converts the chromosome to proper interconnect file that is readable by compile tool of the FlexTools.

```
ChromToCsv( local_chrom->FullChromosome , local_chrom->MainDirectory + "full/interconnect.csv" ,path + "/interconnect.csv" );
```

in In this stage the new configuration undergoes compile and simulation phases to obtain links' usage statistics of processor running under the specific benchmark.

```
string CompileSimulation(
" cd " + path +
" && source  /run" +
" && cat interconnect.csv " +
" && generatFlexCore.php -intercon=interconnect" +
" && date >> time" +
" && echo compile, CompileSimulation" +
" && make compile bm=" + local_chrom->bm +
" && echo simsoft, CompileSimulation" +
" && make simsoft bm=" + local_chrom->bm +
" && cp interconnect.csv interconnectNotTailored.csv"
);
```

As compileSimulation is the basic step of the function it is always executed, no matter what the options are.

```
system (CompileSimulation.c_str());

if (local_chrom->Tailored==true) {
```

Proceed with the tailoring
Links' usage statistics file is located in /65nm-sim/soft/ , the file name is the name of the benchmark with .cycle as the extension

```
string cyclePath(path + "/65nm-sim/soft/" + local_chrom->bm + ".cycle" );
cout<< endl;
local_chrom->FullChromosomeTailored.clear();
TempChrom.clear();
```

This function takes the link statistic usage file and puts 0 for the links that were never used and 1 for the others into their corresponding place in a temporary chromosome called TempChorm

```
LinkConnectionFromCycleFile (cyclePath , TempChrom);
```

If the design is GPP loyal the GPP link that are not used during the simulation are also added to make the design GPP-tailored, other wise the design is kept aggressively tailored.

```
if (local_chrom->GppLoyality==true){
PrintChrom(TempChrom);
PrintChrom(local_chrom->LoyalityChromosome);
local_chrom->FullChromosomeTailored= ChromXor (TempChrom , local_chrom->LoyalityChromosome);}
else {local_chrom->FullChromosomeTailored=TempChrom;}
```

The chromosome that was just produced is converted to csv file to be readable by the compile tool.

```
cout << "FullChromosomeTailored: "<<endl;
PrintChrom(local_chrom->FullChromosomeTailored);
ChromToCsv( local_chrom->FullChromosomeTailored , local_chrom->MainDirectory + "full/interconnect.csv"
,path + "/interconnect.csv" );

string CompileSimulationTailored(
" cd " + path +
" && source  /run" +
" && cat interconnect.csv " +
" && cp interconnect.csv interconnectTailored.csv " +
" && generatFlexCore.php -intercon=interconnect" +
" && date >> time" +
" && echo compile, CompileSimulationTailored" +
" && make compile bm=" + local_chrom->bm +
" && echo simsoft, CompileSimulationTailored" +
" && make simsoft bm=" + local_chrom->bm
);

system (CompileSimulationTailored.c_str()); */
```

This script takes the tailored interconnect file and perform compile and simulation on it, however since tailoring is prone to some errors as the compile scheduler may not be able to find suitable paths for the datapath the script check whether the compile result was fine or not. and if there string makeit(

```
" echo \"#/bin/bash\"> " + path + "/TailoringScript" +
"&& echo \"cd \" '$' 1\" \" >> " + path + "/TailoringScript" +
"&& echo \"source /chalmers/users/hidaji/run\" >> " + path + "/TailoringScript" +
"&& echo \"pwd\" >> " + path + "/TailoringScript" +
"&& echo \"cat interconnect.csv\" >> " + path + "/TailoringScript" +
"&& echo \"cp interconnect.csv interconnectTailored.csv \" >> " + path + "/TailoringScript" +
"&& echo \"generatFlexCore.php -intercon=interconnect\" >> " + path + "/TailoringScript" +
"&& echo \"date >> time\" >> " + path + "/TailoringScript" +
"&& echo \"echo compile, CompileSimulationTailored\" >> " + path + "/TailoringScript" +
"&& echo \"make compile bm=\"'$'2\" \" >> " + path + "/TailoringScript" +
"&& echo \"if [ \"'$'?\" -ne 0 ]; then \" >> " + path + "/TailoringScript" +
"&& echo \"echo \"Error\"\" >> " + path + "/TailoringScript" +
"&& echo \"touch Error\" >> " + path + "/TailoringScript" +
"&& echo \"return\" >> " + path + "/TailoringScript" +
"&& echo \"fi\" >> " + path + "/TailoringScript" +
"&& echo \"echo simsoft, CompileSimulationTailored\" >> " + path + "/TailoringScript" +
"&& echo \"make simsoft bm=\"'$'2\" \" >> " + path + "/TailoringScript"
);

system (makeit.c_str());
string sourcefile("source "+ path +"/TailoringScript " + path + " " + local_chrom->bm);

system (sourcefile.c_str());
string ErrorFile(path+"/Error");
fstream ChromBankFile (ErrorFile.c_str(),ios::in );
```

Leave the function if there were an error and discard the rest of the process if (ChromBankFile.is_open()) {

```
cout << "There is Error"<<endl;
return 0;
}
else{
cout << "No Error"<<endl;
}

cout << "Tailoring Done" << endl;
}
```

This script performs rtl evaluation and then synthesizes the netlist, and based on a VCD file that was generated during simulation gives accurate post synthesis power estimation.

```
string SynthesizeVcd(
" cd " + path +
" && source /run" +
" && echo simrtl SynthesizeVcd"
" && make simrtl"+ local_chrom->stm + " bm=" + local_chrom->bm +
" && echo clean SynthesizeVcd" +
" && make clean" +
" && date >> time"
" && echo synvcd SynthesizeVcd" +
" && make synvcd" + local_chrom->stm + " bm=" + local_chrom->bm + " CLOCKPERIOD=" + local_chrom->clockperiod +
" && date >> time"

);

cout << SynthesizeVcd;
```

The script performs place and rout phase on the design.

```
string PlaceAndRout(
" && make clean"
" && date >> time"
" && echo socvcd IV"
" && make socvcd" + local_chrom->stm + " bm=" + local_chrom->bm + " CLOCKPERIOD=" + local_chrom-
>clockperiod +
" && date >> time"
);

cout << PlaceAndRout;

if (local_chrom->mode >= synthesizeVcd)
command = SynthesizeVcd;
if (local_chrom->mode == placeAndRout)
command = command + PlaceAndRout;

system (command.c_str());
```

Evaluation time and date are also saved in the chromosome

```
time_t rawtime;
struct tm * timeinfo;
time ( &rawtime );
timeinfo = localtime ( &rawtime );
local_chrom->EvaluationTimeAndDate=string(asctime (timeinfo));

cout << local_chrom->Name<< " exit"<< endl;
}
============================================================
```

6.8 Data Types

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h> #include <vector>
using namespace std;

enum toolSteps  compileSimulation, synthesize, synthesizeVcd, placeAndRout ;
```

```
C: Connected, D: Disconnected, G: GPP only , F: Fully connected only
enum gene  D , C , G , F ;
enum crossoverType  OnePoint , TwoPoint , Uniform ;
enum SelectionType  RouletteWheel , RankSelection , SteadyState , Elitism ;
enum RankingType  LinearRanking , nonLinearRanking ;
enum FrameGenerateType  Comparison , Statistics, Test ;


struct chrom_record{
string Name;
string bm;
string stm;
string clockperiod;
string MainDirectory;
string EvaluationTimeAndDate;

toolSteps mode;

int NoLinks;
int CycleRun;

float GateArea;
float PowerProbabilistic;
float EnergyProbabilistic;
float PowerSynVCD;
float EnergySynVCD;
float PowerSocVCD;
float EnergySocVCD;
float fitnessValue;
float SumOfFitnesses;
float Position;

vector <gene> CompactChromosome;
vector <gene> FullChromosome;
vector <gene> FullChromosomeTailored;
vector <gene> LoyalityChromosome;

bool Elite;
bool Tailored;
bool GppLoyality;
};

typedef struct chrom_record chrom_rectype ;

#define MAXTHREAD 16
=============================================================
```