Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bojan Antonović

Evolving the Ultimate Processor

Diploma Thesis DA-2003.31
Summer Term 2003
Tutors: Stefan Bleuler, Rolf Enzler, Christian Plessl

Supervisor:
Prof. Dr. Lothar Thiele

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Contents

1 Introduction **3**

2 Processor optimization **5**

2.1 Task description . . . 5

2.2 Overview . . . 9

2.3 Evaluation tools . . . 10

2.3.1 SimpleScalar . . . 10

2.3.2 CACTI . . . 11

2.3.3 SimWattch . . . 13

2.4 Area consumption . . . 14

2.4.1 Caches . . . 14

2.4.2 Branch predictors . . . 15

2.4.3 Rest of the processor . . . 16

3 Evolutionary algorithm components **21**

3.1 Random elements . . . 21

3.1.1 General description . . . 21

3.1.2 Simple parameters . . . 22

3.1.3 Branch predictor . . . 22

3.1.4 Caches . . . 23

3.2 Mutation . . . 25

3.2.1 Simple parameters . . . 25

3.2.2 Complex parameters . . . 26
3.3 Cross-Over . . . 26
3.3.1 Description and simple parameters . . . 26
3.3.2 Branch Predictor . . . 27
3.3.3 Caches . . . 27
3.4 Summary of EA aspects . . . 28

4 Realization and Implementation 29
4.1 Realization criteria . . . 29
4.2 Model . . . 31
4.2.1 Procedural . . . 31
4.2.2 Functional . . . 32
4.3 JavaPISA . . . 32
4.3.1 Introduction into JavaPISA . . . 32
4.3.2 Variators with JavaPISA . . . 35
4.4 Classes and their collaboration and workflow analysis . . . 36
4.5 Miscellaneous . . . 40
4.5.1 Processes and their output . . . 40
4.5.2 Parallelization . . . 40

5 Optimization results 43
5.1 HelloWorld . . . 44
5.2 Rijndael-enc . . . 44
5.3 Adpcm-enc . . . 47
5.4 Frag . . . 47

6 Conclusion 57

A Processor configuration 59
A.1 Processor core . . . 59
A.2 Memory hierarchy . . . 60

A.3 Branch predictor 61

A.3.1 Branch predictor parameters 61

A.3.2 Area consumption 62

A.4 Caches 63

A.4.1 Unified caches with SimpleScalar 64

B Installation **65**

List of Tables

2.1 Some modern processors with the number of floating point and integer ALUs, and the sizes of their level 1 and 2 cache. 17

2.2 Some modern processors with their structure size λ, the size of their core, and the calculated size of the core in λ^2. 17

List of Figures

2.1 Core of the Athlon processor without level 2 caches. Shape of the units is very detailed. 18

2.2 Core of the Athlon processor with 128 KB of level 2 caches. . . . 19

2.3 Comparison between two Athlon processors with 128 KB and 256 KB of level 2 caches (last two pictures). The first two pictures compare the processors at whole. 20

3.1 All by SimpleScalar allowed cache combinations. The one with DL1 and UL2 makes it freezing. 24

4.1 Basic JavaPISA classes with final variators LotzVariator and ProcessorVariator. All final variators have to implement HashPisaVariator. 34

4.2 The package JavaPISA with its classes. 34

4.3 The inheritances and usages in JavaPISA. 35

4.4 Basic states from the variator view in JavaPISA. 37

4.5 Basic packages and classes used in the thesis software (JavaPISA excluded). 38

4.6 Dependencies of the important classes. 39

5.1 HelloWorld at initial population. 44

5.2 HelloWorld at 25th generation. 45

5.3 HelloWorld at 50th generation. 45

5.4 HelloWorld at 75th generation. 46

5.5 HelloWorld at 100th generation. 46

5.6 Rijndael at initial population. 47

5.7 Rijndael at 25th generation. 48

5.8 Rijndael at 50th generation. 48

5.9 Rijndael at 75th generation. 49

5.10 Rijndael at 100th generation. 49

5.11 Adpcm-enc at initial population. 50

5.12 Adpcm-enc at 25th generation. 50

5.13 Adpcm-enc at 50th generation. 51

5.14 Adpcm-enc at 75th generation. 51

5.15 Adpcm-enc at 100th generation. 52

5.16 Frag at initial population. 53

5.17 Frag at 25th generation. 53

5.18 Frag at 50th generation. 54

5.19 Frag at 75th generation. 54

5.20 Frag at 100th generation. 55

A.1 Area consumption for the combined predictor including its bimodal and 2-level part. 63

Abstract

This diploma thesis is about optimizing processors by the criteria performance, area and power consumption. Because this is a multi criterion optimization problem, the created results do not represent a single optimal or best fraction of optimal solutions, they represent a bunch of non-dominated elements, called the Pareto front. To get this Pareto front, a multi criterion optimization algorithm, based on the principle of Evolutionary Algorithms (EA), SPEA2, is used. Tools are used to compute the objectives where possible. The estimation of the area consumption is showed how it's made with tools and complementary information like pictures of processor cores. EA aspects like creation of random elements and variation operators like mutation and cross-over are discussed. A model is shown which support this all. The protocol PISA is used to connect the optimization algorithm with the optimization problem, together with its variators. A PISA implementation in Java, JavaPISA, is presented. At the end optimization results are presented and discussed.

Chapter 1

Introduction

Optimizing processors is an iterative process. One can select a certain processor configuration by hand or by reflection, and then evaluate it considering chosen criteria like performance, area and power consumption. Afterwards, one selects an other processor configuration and evaluates this one again. This selection and evaluation cycle produces and evolutionary path of elements one is interested in. One might have found the ideal product or a collection of interesting ones of them. All in all, this is a manual process, which can be automatized to get at least an other optimization alternative.

To do an automated optimization of an processor, all steps have to be automatized, which means they must be executable by a computer. Both parts of the optimization process, the element selection and the element evaluation, have to be done in software. The smart element selection has to be done by an optimization algorithm, and the element evaluation by simulators.

Working with multiple criteria leads to the consequence, that, in difference to optimizing against a single criteria, where one gets a single optimal solution, with multiple criteria one can not receive a single optimal solution in general, because some elements might be good in one criteria, but for an other one there exists a better element. Of course, if an element is at least as good in *all* criteria as an other one, then the last one is ignored, because it's dominated by the first one. So filtering all dominated elements out of the collection, one receives a bunch of compromises, called the *Pareto front*[1]. It's exactly this Pareto front in which all decision makers are interested in.

Because we work with an *multi objective optimization problem*, MOOP in short, we have to have an *multi objective optimization algorithm* (MOOA). MOOAs are a new and living field. It's relevant to select one or two well tested algorithms for, firstly avoiding to reinvent the wheel, and secondly and much more important, to have a reliable part for your automated optimization. SPEA2 and NSGA2 are two favorites. See [1] for a good introduction into the topic of MOOPs.

The other component of the optimization process, the evaluation part, is done by simulators. Simulators can serve completely for an optimization criteria, or

they can only partially fulfill their job, for which it's necessary to complete the necessary information. An other problem is the precision of the received results. While precise values are not problematic, so are their counterparts, the imprecise ones. One can try to improve their precision to make them useful. But one may have few or no luck with other ones, for whatever reason it may be behind. So one has to live with this problem and being aware of the imprecise results. An other possibility is to learn how to deal with this uncertainty, which makes the needs to have knowledge that doesn't exist in mature form.

Whatever problem you want to optimize, you have to provide the tools for optimizing it. Using a *multi objective evolutionary algorithm* (MOEA) as SPEA2 is, you need also the components for it to serve. This means you have to have to define how random elements are created, to have an initial population, and one or more variation operators like mutation and cross-over. Multiple of them might be possible and necessary if the problem to optimize for offers and needs multiple variation possibilities.

So having your optimization algorithm and your problem variator, you have to connect them together so that they can work. Realizing this collaboration can be done with specific knowledge and by reinventing the wheel, or you can take an existing collaboration protocol to realize your optimization process. PISA[2] is an example.

Finally, the problem you want to optimize has to be well understood a priori. While this is clear and sounds reasonable, understanding it and knowing all internals like value dependencies in before is a key and a must for successfully realizing the steps needed to complete your optimization process. Mostly you have to transform your vague, and for yourself clear, knowledge into a clear mathematical model to work with it.

This diploma theses deals concretely with optimizing processors for performance, area and power consumption, with SPEA2 by PISA, with simulators for all criteria, and with the goal to create Pareto fronts in respect to certain benchmarks. Not treated is the uncertainty problem or a comparison between different MOEAs. The focus is on developing a complete and qualitative model and framework to do processor optimizations. Thorough work and a solution stand in front, oppositely to fast, vague and imprecise steps which make a found solution useless, reuseless and therefore oblige to redo and restart and new solution search.

Chapter 2

Processor optimization

2.1 Task description

Here the exact task description in german:

Aufgabenstellung für die Diplomarbeit
"Mehrkriterielle Optimierung eines Prozessors mittels
Evolutionärer Algorithmen"
Sommersemester 2003

Student:	Bojan Antonovic
Betreuer:	Stefan Bleuler (TIK), Christian Plessl (TIK), Rolf Enzler (IfE)
Professor:	Lothar Thiele

Thematischer Hintergrund

Die Rechenleistung eines Prozessors wird von einer Vielzahl von Faktoren bestimmt. Die komplexen Interaktionen der Architekturparameter machen eine analytische Bestimmung des Einflusses einzelner Parameter auf die Gesamtleistung der CPU schwierig. Deshalb werden vermehrt CPU Simulatoren eingesetzt. In unseren Projekten wurde bisher der *SimpleScalar* Simulator eingesetzt (www.simplescalar.com) [2, 6]. SimpleScalar hat sich für viele Fragestellungen als zweckmässig erwiesen und wird von vielen Forschungsgruppen eingesetzt.

Mit SimpleScalar wird ein CPU-Architekturmodell simuliert, das sich mittels Textdateien konfigurieren lässt. Die Applikationen werden mit einem C Cross-Compiler kompiliert und auf der simulierten CPU ausgeführt. So kann die benötigte Ausführungszeit bestimmt werden. Neben der Ausführungszeit für bestimmte Applikationen gibt es zwei weitere zentrale Kriterien beim Prozessordesign: Erstens, die Chipfläche, die ein guter Indikator für die Kosten eines

Prozessors ist, und zweitens, der Energieverbrauch. Für diese beiden Kriterien existieren Ansätze von Estimations- und Simulationsverfahren.

Da die Qualität eines Prozessors bezüglich der genannten drei Kriterien durch Simulation oder Estimation bestimmt werden kann, bietet sich eine automatisierte Optimierung der Prozessorarchitektur an. Dazu sollen in dieser Arbeit Evolutionäre Algorithmen eingesetzt werden. In dieser Richtung existieren schon vereinzelte Ansätze [14, 9, 1].

Evolutionäre Algorithmen (EA) sind stochastische, iterative Optimierungsverfahren [10, 3]. Sie imitieren die Prinzipien der natürlichen Evolution (Variation und Selektion), um eine Population von Entscheidungsalternativen (Lösungvorschlägen) sukzessive zu verbessern. Beim vorliegenden Optimierungsproblem stehen die verschiedenen Optimierungsziele in Konflikt. Es soll deshalb nicht nach *einer* optimalen Lösung gesucht werden, sondern nach der Menge optimaler Entscheidungsalternativen, die den "trade-off" darstellt, den man beim Design eines Prozessors eingehen kann. Für solche Mehrzieloptimierungsprobleme eignen sich EAs sehr gut [18, 7].

Ausgangslage

Ein Pool von Applikationen aus dem embedded and wearable Bereich wurde auf SimpleScalar portiert und ausgemessen [8, 15]. In einer vorgängigen Semsterarbeit wurde ein Framework geschaffen zur Ankopplung von Optimierungsalgorithmen an den SimpleScalar Simulator [12].

Mit CACTI existiert ein Tool zur Bestimmung der Chipfläche und des Energieverbrauchs von Caches [13]. Zur Abschätzung des Energieverbrauchs eines simulierten Prozessors existieren mehrere SimpleScalar-Erweiterung wie Power-Analyzer [11], Wattch [5] oder SimplePower [16, 17].

Auf der Seite der EAs existiert eine Schnittstelle (PISA) [4] die es erlaubt die problemabhängigen Teile einer Optimierungsmethode (z.B. Mutation und Rekombination) von den problemunabhängigen Teilen (z.B. Selektion) zu trennen. Diese Teile können dann als unabhängige Programme implementiert werden. Es bestehen bereits Module für mehrere evolutionäre Mehrzieloptimierungsverfahren. Siehe auch `www.tik.ee.ethz.ch/pisa`.

Problemstellung

Basierend auf den bestehenden Komponenten ist ein Programm zur evolutionären Mehrzieloptimierung von Prozessoren zu entwickeln. Das Ziel ist, Prozessoren für bestimmte Anwendungsgebiete optimieren zu können. Wie erwähnt sollen dabei dem Benutzer alternative Designvarianten präsentiert werden.

Für die Arbeit sind im wesentlichen folgende Teilaufgaben zu bearbeiten:

- Einarbeitung in SimpleScalar und das bestehende Framework zur Anbindung an Optimierungsalgorithmen.

- Abklären der Varianten für die Bestimmung der Chipfläche und des Energieverbrauchs.
- Einarbeitung in evolutionäre Mehrzieloptimierung und PISA.
- Definition einer geeigneten Repräsentation des Optimierungsproblems und Definition entsprechender Variationsoperatoren.
- Auswahl eines geeigneten Sets von Applikationen die als Benchmarks dienen.
- Untersuchen des Einflusses verschiedener Architekturparameter.
- Implementierung des Optimierungsproblems als Varitor Modul für PISA.
- Vergleich verschiedener Selektionsmethoden mittels PISA.
- Optimierung von Prozessoren für verschiedene Applikationsgruppen.
- Vergleich der resultierenden Architekturen mit real existierenden Prozessoren.

Neben den genannten Kernaufgaben bestehen weitere Fragestellungen, die in Absprache mit den Betreuern untersucht werden können. Darunter fallen:

- Während die Laufzeitsimualtion sehr genau Resultate liefert, basiert z.B. die Bestimmung der Chipfläche auf einer Abschätzung. Die unterschiedlichen Unsicherheiten in den Zielfunktionswerten könnten im Optimierungsalgorithmus berücksichtigt werden.
- Vergleich mit anderen nicht evolutionären Optimierungsverfahren.
- Weiterführende Abschätzungen von Chipfläche und Energieverbrauch.

Organisation

Dauer der Arbeit: 4 Monate.

Zeitplan: Erstellen Sie am Anfang der Arbeit einen Zeitplan. Halten Sie Ihren Arbeitsfortschritt laufend fest.

Wöchentliche Besprechung: Um den Stand der Arbeit zu verfolgen und um Schwierigkeiten oder das weitere Vorgehen zu besprechen ist ein wöchentliches Treffen mit den Betreuern geplant.

Anfangsvortrag: Ca. zwei bis drei Wochen nach Beginn der Arbeit soll ein kurzer Vortrag über die Aufgabenstellung und die geplanten Schritte gehalten werden. Dauer: maximal fünf Minuten.

Schlussvortrag: Gegen Ende der Arbeit sollen die Resultate in einem ca. 20 minütigen Vortrag präsentiert werden.

Dokumentation: Am Ende der Arbeit ist ein schriftlicher Bericht abzugeben. Dokumentieren Sie Ihre Arbeit sorgfältig. Beschreiben Sie darin nicht nur die Resultate, sondern auch die Überlegungen und Designentscheide. Im weiteren sollen auch die von Ihnen geschriebenen Programme sorgfältig kommentiert und dokumentiert werden.

Literatur

[1] S. Agarwal, E. Chan, B. Liblit, and C. J. Lin. Processor characteristic selection for embedded applications via genetic algorithms. Semester project report, EECS, Univ. of California Berkeley, Dec. 1998.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[3] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing and Oxford Univertity Press, Bristol, United Kingdom, 1997. www.iop.org/Books/CIL/HEC/index.htm.

[4] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA — a platform and programming language independent interface for search algorithms. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494 – 508, Berlin, 2003. Springer.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. 27th Int. Symp. on Computer Architecture (ISCA)*, pages 83–94, 2000.

[6] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report #1342, Computer Sciences Dept., Univ. of Wisconsin–Madison, June 1997.

[7] C. C. Coello. List of references on evolutionary multiobjective optimization. www.lania.mx/ ccoello/EMOO/EMOObib.html.

[8] R. Enzler, M. Platzner, C. Plessl, L. Thiele, and G. Tröster. Reconfigurable processors for handhelds and wearables: Application analysis. In *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications III*, volume 4525 of *Proceedings of SPIE*, pages 135–146, 2001.

[9] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *IEEE Computer*, 35(9):39–47, Sept. 2002.

[10] V. Nissen. *Einführung in Evolutionäre Algorithmen*. Vieweg, 1997.

[11] PowerAnalyzer homepage. http://www.eecs.umich.edu/∼jringenb/power/.

[12] M. Rufer and S. Wyss. Processor Wizardry. Semesterarbeit, Institut für Technische Informatik und Kommunikationsnetze, ETH Zürich, Juli 2002.

[13] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. WRL Research Report 2001/2, Western Research Laboratory, Compaq Computer Corp., Dec. 2001.

[14] T. J. Stanley and T. Mudge. Systematic objective-driven computer architecture optimization. In *Proc. Conf. on Advanced Research in VLSI (ARVLSI'95)*, pages 286–300, 1995.

[15] G. Thomassin. Applications for Handhelds: Profiling and Analysis of MediaBench. Student thesis, Computer Engineering and Networks Laboratory, ETH Zurich, Oct. 2000.

[16] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. 27th Int. Symp. on Computer Architecture (ISCA)*, pages 95–106, 2000.

[17] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Proc. 37th Design Automation Conf. (DAC)*, pages 340–345, 2000.

[18] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, 1999. TIK-Schriftenreihe Nr. 30, Diss ETH No. 13398, Shaker Verlag, Aachen, Germany.

2.2 Overview

As explained in the introduction, this diploma thesis deals with the optimization criteria performance, area and power consumption. For using them, the tools SimpleScalar[4][5] (performance, result in cycles, evaluation by a benchmark),

CACTI[8][9] (area consumption and access time of caches) and SimWattch[6][7] (power consumption, result in Watt) are presented, together with their needs. Further, all parameters of a processor are presented and, the most important, where importance is defined for analytical importance instead for the object itself, are discussed. Then the ideas of random population creation and the variation operators mutation and cross-over are discussed with some variants of them. Then an analysis is presented how the area consumption is made by using multiple information sources. Afterwards some implementation aspects and products are presented. Beyond them is JavaPISA, a realization of the PISA protocol in Java. At the end, optimization runs of multiple benchmarks are presented and analyzed.

The differences between this diploma thesis and a similar semester thesis[10] are:

- a complete cache architecture model is used, which allows also unified caches
- a better area consumption model is used
- a better MOOA and the interface protocol PISA are used
- reutilization by structured, object-oriented implementation in Java is made (instead with Perl)

This points shouldn't be understood as a critique. It's so that preworkers are concentrating on building up something, for which they selected their best possible realization method. Also SPEA2 and PISA where not available at this time period. Nevertheless, in the implementation part it's explained what realization decisions are chosen for this thesis to provide a good framework.

2.3 Evaluation tools

In this section the used tools for determining the criteria performance, area and power consumption are presented and their basics are shown up. Informations about processor parameters will be introduced drop-wise and where necessary. It's not necessary for you to understand all parameters in order to understand the explained problem. But feel free to look intro appendix A for an excact explanation about all processor parameters.

2.3.1 SimpleScalar

SimpleScalar 3.0 [4][5] is a tool to determine, how many cycles a processor, by a given processor configuration, needs to proceed through a given benchmark. Your processor configuration must be in a SimpleScalar readable format (see appendix) and the benchmark compiled for the processor model SimpleScalar

uses. Compilation of a in C given benchmark can be done by a modified GCC version. For details about the SimpleScalar suite see appendix.

SimpleScalar can emulate an out-of-order processor. For this, the command `sim-outorder` has to be used. If your configuration has the name `myconfig.config` and the compiled benchmark the name `mybenchmark.eio`, then the corresponding call of sim-outorder is

```
sim-outorder -config myconfig.config mybenchmark.eio
```

If you omit the configuration, then the default configuration is used. If you omit a single or more parameters, then their default values are taken.

Very useful is the feature to automately send the output to a file, which can be read and parsed later:

```
sim-outorder -redir:sim result.txt mybenchmark.eio
```

which, in this example, sends the output of the evaluated default configuration to the file `result.txt`. The value of interest at SimpleScalar is the number of cycles:

```
sim_cycle              11491 # total simulation time in cycles
```

which can be easily parsed by before cutting off the text left and right aside from the cycle number or how one prefers.

SimpleScalar is a very stable tool. Two ugly bugs are when you choose the block size of a cache in the second level smaller then it's corresponding part in the first level cache, like the example

```
-cache:dl1      dl1:8:32:8:f
-cache:dl2      dl2:8:16:8:r
```

shows, where the first block size (32) is greater then the second (16) , and when you use for the level 1 cache only a data cache with the combination of a unified second level cache. While this case is not forbidden or caught by the SimpleScalar input check, it makes SimpleScalar freezing. A really ugly, and for beginners a hardly discoverable one!

2.3.2 CACTI

CACTI [8][9] is a tool for computing the access time, the area and power consumption of caches and everything you see as caches like memories. CACTI takes a cache configuration and returns the demanded values with an internally

optimized cache. Optimized means that beyond multiple possible caches the best is chosen. This is an internal CACTI feature, as described in [9]. While the CACTI parameters are described in [9], appendix A, here is a more helpful description of them with an useful example.

To call CACTI, you have to pass the following parameters as scalar values, if nothing else stands:

- cache size
- number of blocks
- associativity
- technology (lambda)
- number of subbanks

An extended parameters list comprehends the number of read-and-write ports, and the number of read-only and write only ports. So the call syntax looks like:

```
CACTI <size> <blocks> <assoc> <lambda> <subbanks>
```
or
```
CACTI <size> <blocks> <assoc> <lambda> <RWP> <RP> <WP> <subbanks>
```

The values can have the following parameters:

- cache size ≥ 64 bytes
- number of blocks ≥ 8
- associativity $\in [1, 32]$ or FA
- technology (lambda) value in μm
- number of subbanks ≥ 1
- read-and-write ports $\in [1, 2]$, default 1
- read-only ports ≥ 0, default 0
- write only ports ≥ 0, default 0

Example: Let's say we have a cache with 64 sets, 16 blocks, associativity of 4, lambda of 0.13 μm and 1 subbank. Then we have a cache size of $64 \cdot 16 \cdot 4 = 4096$ bytes. The CACTI call would be:

```
CACTI 4096 16 4 0.13 1
```

CACTI delivers many parameters. We are only interested in the access time, the area and power consumption.

```
Access Time (ns): 0.889773
Total Power all Banks (nJ): 0.544613
Total area subbanked (cm^2): 0.004362
```

The CACTI manual [9] gives an other example with an old technology size, and its written for CACTI 3.0. In CACTI 3.2 the naming of the last two parameters has changed slightly.

The access time, which is here in nano seconds, is used later for computing the access time of the cache in processor cycles. For this, a 2 GHz processor is assumed, and the number of cycles is rounded up, because even a 10th of cycle can't be ignored, so the access will take a full additional cycle. This means that our access time of 0.889773 ns will have equivalent of 1.779546 cycles, which means 2 clock cycles in the practice.

The consumed power is directly irrelevant for this thesis. However, because the following tool, SimWattch, and future extensions of this thesis can use it, it's presented here and parsed in the software.

2.3.3 SimWattch

SimWattch[6][7], like SimPower and PowerAnalyzer, is an extension of SimpleScalar, especially of version 3.0, which, in opposite to the other two, came at closest for our purposes, because the others two either didn't fulfill our demands or where excluded by the tutor of this thesis and the last one was not regarded after SimWattch was found as useful.

SimWattch takes a processor configuration and returns, beside all values SimpleScalar does, also the total power consumption of the processor, which is expressed in the line

```
Total Power Consumption: 78.5882
```

The returned value is in Watt. SimWattch uses CACTI to determine the power consumption of caches, and internal models for the rest. After many investigations to make SimWattch runnable on our working platform(s), and by patching it do support missing caches, and parsing the power value in software, it was discovered that SimWattch uses and old version of CACTI, likely to be version 2.0, which has a less detailed area model and no subbanking, and therefore also an incompatible power consumption model with CACTI 3.2. Also the returned values made no sense. Even for a processor in 0.18 μm technology, with a clock frequency and voltage we also couldn't extract, results in more than three magnitude of order like 7 W and 21 KW (!) disqualified SimWattch for further usage. Therefore, and because power consumption was made usable in the last part of the thesis time and before no one had a reliable power consumption tool, power consumption was dropped as an optimization criterion. At least it served as a proof of concept for the thesis software and model.

2.4 Area consumption

An estimation of the area consumption can be made for caches, as in the CACTI presenting subsection was explained. What about the rest? Branch predictors need tables to look up. While this tables have often a width of some few bits, it should be possible to view them as memories, and then to compute their area consumption by CACTI. But there's still a rest of the processor. The saving idea came up to estimate the area consumption of certain units from pictures of processor cores.

First, a remark about the area consumption of normal caches is made. Second, the area consumption for branch predictors is presented. Together with the value of all caches, only the rest has to be computed.

The computations are made to express the results in the structure size λ, because this gives the possibility to choose a demanded λ and to get the die size of an actual technology like 0.13 μm, which is also used in the implementation.

2.4.1 Caches

You can read this subsection without understanding completely how caches are organized. Exact knowledge about caches is not necessary.

Caches, like the instruction and data cache of the first or second level, or a unified part of them in a level, can, together with the instruction and data TLB, be straightforwardly sent to CACTI to get their area consumption. On the fly, the latency times for the first and second level cache can be determined to have the most realistical values. For instruction and data TLB this possibility doesn't exist. Therefore this is an open gap in the SimpleScalar architecture.

Back to the area consumption eastimation. Because CACTI 3.2 introduced a banking model, and offers the possibility to specify the number of read-and-write, read-only and write-only ports, a reasonable model has to be found to use appropriate values. [9] says that a multi-banked cache can in best case be as good as a multi-ported. While one may understand it that a 1-ported and n-banked cache is in ideal case like a n-ported 1-bank cache, the question comes up for the behavior of a n-ported m-bank cache. A mail to one author of CACTI stayed unanswered[1]. So no experiments where made and the assumption is selected that all caches consist of only one bank while only the number of ports is varied.

SimpleScalar has a parameter `-res:memport` which specifies the total number of L1 cache ports. Neither in the document nor in the code can be seen exactly of which type this ports are. So, generally, a cache of the first level, might it be a IL1, DL1 or a UL1, has the number of read-and-write ports that is equally to the memport parameter, and no read-only and write-only parameter.

[1] Premishkore Shivakumar is meant. But he gave many replies before. It could be that he's absent at the moment.

Because the IL1 cache never writes data back to higher cache hierarchies (IL2, UL2 or RAM), it shouldn't need a real write port. It should only need a fast updating logic which could be implemented efficiently in reaction time and area consumption, and so only a read-only port would be necessary. Again, also this question of the same CACTI author stayed unreplied.

It was observed by the author and described in [9], that a split of a read-and-write port into a read-only and write-only port doubles the size of the cache like the number of read-and-write ports is doubled. Experiments have shown that a write-only increases the area consumption more then a read-only port, and a read-and-write port, like mentioned above, more then both together.

The reason why this port analysis was made is that CACTI doesn't support more then two read-and-write ports. So the question came up, how more then two memports should be represented. A possibility is that up to a value of two ports a value of two for the read-and-write ports is used, zero for the read-only and write-only, and for all values above two the rest one read-only *and* write-only port is taken. This thoughts were made to provide a work-around for the CACTI limit to still offer more than two points. A contra point is that the area consumption of the cache is dramatically increasing as described in [9]. The final decision, chosen with the tutors, was that the memport parameter can be only in the interval [1,2], and that our optimization trials have to live with this limit.

For the caches of the second hierarchy and the TLB caches a single read-and-write and no read-only or write-only port configuration was assumed, because SimpleScalar doesn't support multiple cache ports for this caches.

2.4.2 Branch predictors

Branch predictors need tables for look ups. And tables are memories, and those can be regarded as caches under certain circumstances. But let's first take a look at all possible branch predictors and then analyze their area consumption step by step.

- perfect
- taken
- not taken
- bimodal [BTC, RSS, BTB]
- 2-level [2LEV, RSS, BTB]
- combined [BTC, 2LEV, RSS, BTB, COMB]

We see that the first three contain no parameters. For this simple branch predictors a negligible area consumption, which means 0, is assumed. The rest has some parameters of common type. So, if, for example, the parameter BTC exists, then its area consumption is computed and then added up to the result of results of other parameters, if they exist.

It should be noted in advance, that all this parameters use tables with n entries and a width of m bits, but the minimal block size, the depth of an entry, that CACTI needs for an area consumption service to provide, is 8 bytes, 64 bits at the end. So doing it trivially would lead to a massive area waste.

The transformation model that was used is the following. Assume that 64 is a multiple of m, the width. Then $\frac{64}{m}$ entries can be packed into a block, which is an entry of a higher order cache. So we would have a $\frac{mn}{64}$-entry cache of size mn bits (equal to $\frac{mn}{8}$ bytes). Assuming that this higher order cache is of associativity 1, which means direct mapped, we have a higher order cache with $\frac{nm}{64}$ sets, block size of 8 bytes, associativity 1 and a cache size of $\frac{mn}{8}$ bytes. This are all parameters you need to feed CACTI! And for the structure size λ the same value can be used as for ordinary caches.

It's assumed that the packing and depacking logic is minimal and efficient.

Because CACTI can't proceed smaller cache sizes then 64 bytes, we have to use a minimal amount of space for each branch predictor entry if it falls under a certain value. If some parameter has a widht of 2 bits, which would correspond to $\frac{64}{\frac{2}{8}} = \frac{64 \cdot 8}{2} = 256$ entries, then every part of the branch predictor with less then 256 entries would have a bigger estimation of the area consumption than in reality[2]. However, and more important, an area consumption model for branch predictors is received, which is reasoned and correct in the trend of larger entry sizes.

2.4.3 Rest of the processor

In the previous subsection we have seen how the area consumption for usual caches and as cache viewn branch predictors is computed. However, for the rest of the processor we don't have the luck for a CACTI, or any tool, supported area consumption estimation. A different way has to be found.

A good possibility is to collect information from real processors like the number of units they have, their core sizes, their fabrication λ, and of pictures of their cores where it can bee seen how the single elements are arranged. A simple and compact arrangement shows us that an area packing optimization problem of the units would fall away, which can become a heavy burden under the sheer amount of difficulties that are already here.

The table 2.1 shows a collection unit data of some real processors. In table 2.2 we see area information of some real processors.

Interesting is the observation, that many processor seemed to be constructed that way that their area consumption is a fine multiple of an area expressed by the square of the structure size (λ^2).

More important is that pictures of the Athlon processor core where found without, with 128 KB and 256 KB L2 cache, where it can be seen that the L2 cache was only attached to the rest core and that the processor itself is packed very

[2] Negligible in contrast to a large level two cache.

processor	FP ALUs	INT ALUs	L1 cache	L2 cache
Power4	2	2	128K / 64K	1.41 M
Power3-II	2	2	32K / 64K	4 or 8 M
Pulsar	2	2	64K / 64K ?	8 M
PowerPC 970	2	2	64K / 32 K	512 K
G4 (latest)	1	1	32K / 32K	256 K
Opteron	n/a	n/a	64K / 64K	1 M
Athlon XP (Barton)	3	3	64K / 64K	512K
Athlon (Thor.,M8)	3	3	64K / 64K	256K
Athlon (K7)	3	3	64K / 64K	-
P4 (Northwood)	n/a	n/a	n/a	512K
P4 (Willamette)	n/a	n/a	n/a	256K

Table 2.1: Some modern processors with the number of floating point and integer ALUs, and the sizes of their level 1 and 2 cache.

processor	λ	die (in mm^2)	die in $G\lambda^2$
Power4	180	-	- ≈ 7
Power3-II	220	163	≈ 7
Pulsar	220	140	≈ 7
PowerPC 970	130	118	$6.98 \approx 7$
G4 (latest)	180	106	3.27
G4 (MPC 7410)	180	-	-
P4 (Northwood)	130	145	$8.58 \approx 9$
P4 (Willamette)	180	217	6.67
Athlon XP (Barton)	130	101	$5.98 \approx 6$
Athlon XP (Thoroughbred)	130	84	$4.98 \approx 5$
Athlon XP (2000+)	180	128	$3.95 \approx 4$

Table 2.2: Some modern processors with their structure size λ, the size of their core, and the calculated size of the core in λ^2.

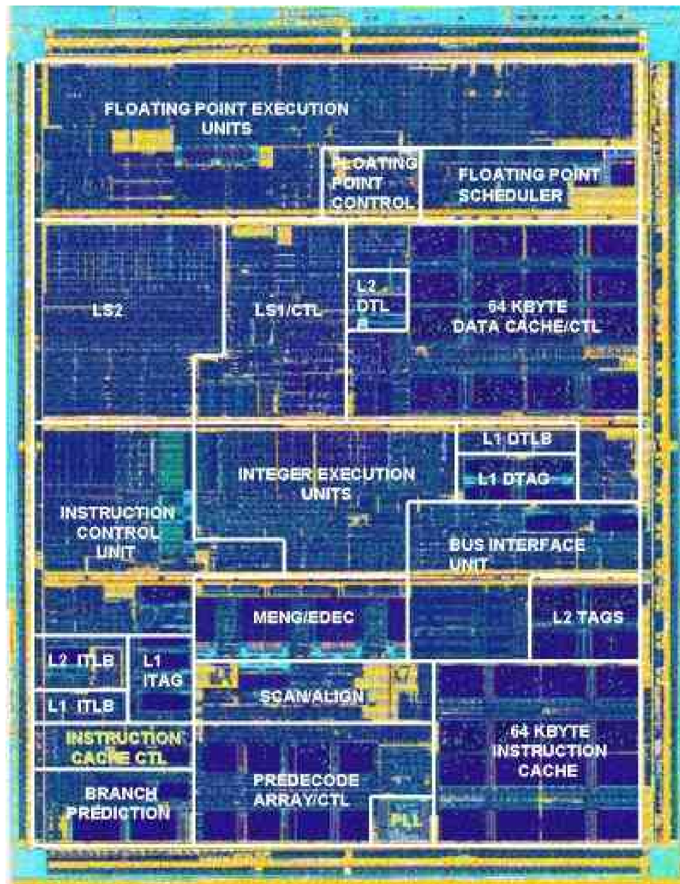

Figure 2.1: Core of the Athlon processor without level 2 caches. Shape of the units is very detailed.

efficiently, which removes us the trouble of an unit packing problem, as it can be seen by the pictures 2.1, 2.2 and 2.3.

What one can do and what was made is to measure out manually the sizes of single units, look at their proportionality corresponding to the whole picture size, and then multiplying the size of the processor in λ^2 to receive the size of the unit in λ^2.

From this core pictures we are not interested in information we have already like the caches or the branch predictors. The most useful information is the size of the integer and floating point units. Because we know the area size of a unit aggregation, and because we know the number of subunits this aggregation holds (3 in the case of the Athlon), we simply divide the area by the number of units to obtain the area for a single unit.

The story isn't over know. SimpleScalar has parameters for a INT ALU and a INT MulDiv, a multiplication and division unit, and corresponding parts for the FPU, the FPU ALU and the FPU MulDiv. As far no further information is

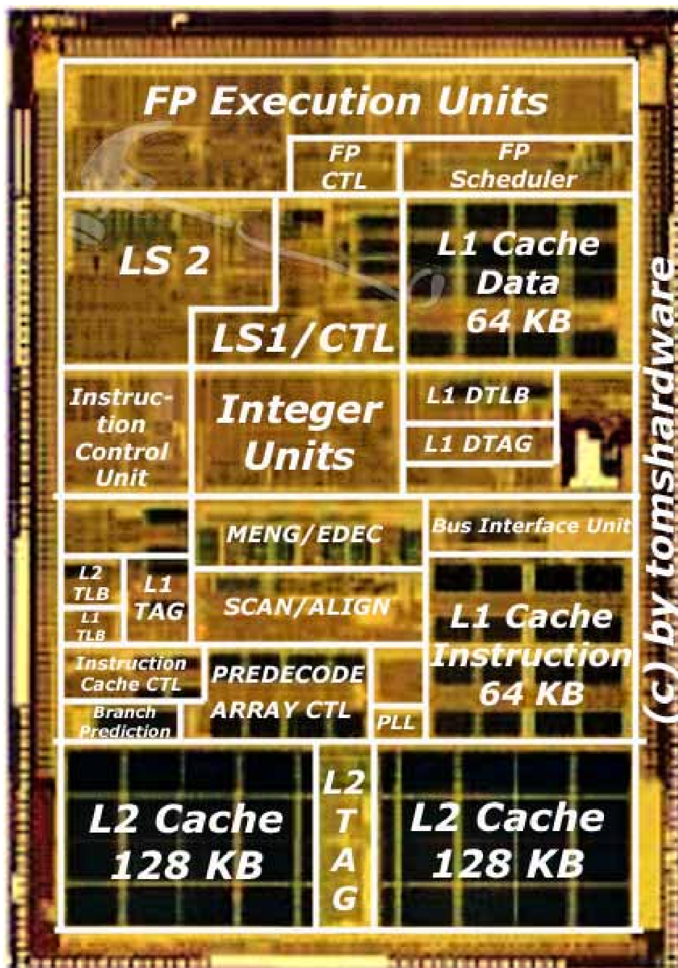

Figure 2.2: Core of the Athlon processor with 128 KB of level 2 caches.

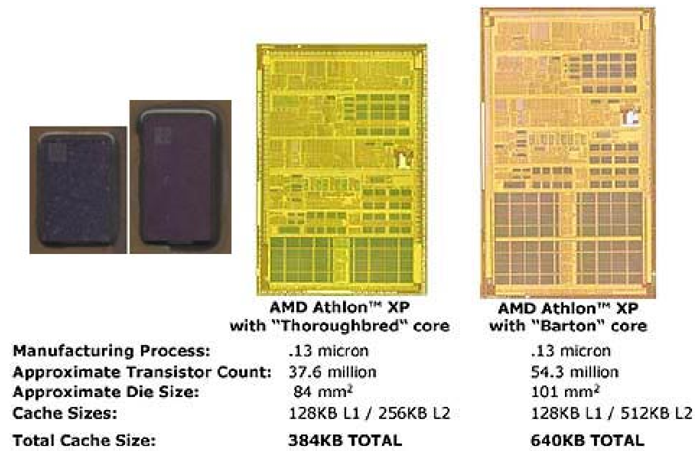

Figure 2.3: Comparison between two Athlon processors with 128 KB and 256 KB of level 2 caches (last two pictures). The first two pictures compare the processors at whole.

available about a partitioning of the corresponding area or size proportionalities between an ALU and a MulDiv unit, it's assumed that they have the same size, which is the computed value above.

The rest of the processor, from which no information can be gained, is assumed to be a basic area and unit demand for a workable processor.

The overall model is surely discussable. But it should be said that the Athlon was the only processor where all informations for completing the calculation were found, and that at most 5 values in the area estimation have to be changed when better informations are obtained.

Chapter 3

Evolutionary algorithm components

In this chapter the topic of components for evolutionary algorithms is regarded. This means, ideas are presented how a processor configuration can be randomly created and variated, specially how mutation and cross-over can be done over it. The ideas are not regarded to be complete, and are discussable.

The discussion will be separated for simple values like the number of integer units or certain powers of 2, and for complex parameters like branch predictors and cache configurations.

3.1 Random elements

3.1.1 General description

Understanding how to create random elements is a prerequisite for further complex operations like variation. While one would think this is not a great deal, he/she must realize the the opposite is true. A big problem is the one of the feasibility of the created element, because most complex problems have a sort of feasibility. If you have a feasibility checker, a simple one or an aggregated one with multiple subcheckers, the you could create elements, check whether they are feasible, and then kick the unfeasible away. While this is surely a simple and good method to begin with, it produces the problem of the unfeasibility rate. If the unfeasibility rate is of an enough high fraction, let's say x, then we have to expect to make simply $\frac{1}{x}$ more trials to collect our amount of feasible elements. But if x is a very small number, then we will need a huge, maybe even exponential high, number of trials to have our initial population. A possibility is to find a model that represents your problem that reduces your unfeasibility rate drastically. But this would open an other problem front which has to be closed with eventually a large amount of work. An other, maybe smarter, pos-

sibility is to create feasible elements by default. While perfect feasibility in one step can be hard to reach, they idea is to reach it in two or more steps. First an as feasible element as possible is created, and then a feasible one is created by transforming the given one into a close but feasible one. This multi-step production may produce certain elements more often then others, because their creation probability is not uniform as good random would allow, but this is only the problem of the initial population. Because we deal with algorithms based on evolution, their overrepresentation can disappear.[1]

Now we want to take a view on the parameters, to make a categorization of them and to see the difficulties clearer. Exact informations about processor parameters, like presented in the appendix, are not necessary for the first two subsections. And for the caches they will be introduced to understand the topic.

3.1.2 Simple parameters

Most parameters are simple and independent. This means they are usual numbers. Some of them are powers of two. All have to lay in a certain range. So their creation is simple. One takes a random number from this range and then returns this value. For powers of two an exponent in the range for exponents is taken, and then a power of two with this exponent is returned. So the returned element consist of independent coordinates. Seeing the powers of two only by their exponent, the returned object is originated in an object with a geometric form of a multidimensional cuboid.

3.1.3 Branch predictor

The branch predictor is a complex parameter but still independent from others. Instead of being filled by numbers of any form, this value consists of an object, chosen from a collection of disjoint objects. So this objects are a generalization of numbers. Having a view at the possible values for a branch predictor

- perfect
- taken
- not taken
- bimodal [BTC, RSS, BTB]
- 2-level [2LEV, RSS, BTB]
- combined [BTC, 2LEV, RSS, BTB, COMB]

we see that only the first three possible values are simple objects like a generalization of numbers. The other three, if chosen, contain subparameters. So for

[1]Non-uniform probability of created elements is only a problem for topics like Monte-Carlo tests, where some data has to be collected, but we have our correction factor, the EA.

an external viewer having 5 integers units is in the same range like having a bimodal branch predictor, but for using a bimodal branch predictor firstly some more arguments are needed, and secondly not all of them occur in the other branch predictor values. So a branch predictor has to have additional values that others don't need. Creating a branch predictor can be done by first randomly choose a type, and then further a random value for each of its subtypes. Furthermore, if this subtypes again contain subtypes, then the random selection has to continue until the last level of this parameter tree.

3.1.4 Caches

Cache configurations themselves don't have this amount of parameter levels like branch predictors. They can be of certain type, and each cache type uses then same parameters like every other cache. But attention, there are different sorts of caches. On the one side, we have caches in the first and second level of the cache hierarchy, and we have instruction and data TLBs. The last two are independent of the others and from each other. So creating them randomly means creating their parameters randomly. Then for them the creation process in finished.

Caches of both levels are independent from other parameters but they have multiple intra-dependencies. First, a cache of a certain level can have one of this possibilities:

- none
- instruction and data cache
- instruction cache only
- data cache only
- unified cache

Caches with at least an instruction or data cache can be viewn as splitted caches, oppositely to a unified cache. Furthermore, not all possibilities between cache types between the first and the second level are allowed, as can be seen by figure 3.1.

So there is an inter-dependence of cache types. Also there's a further inter-dependence between a certain type of the cache parameters, the block size, and this only between the same cache type of two levels. This means that the block size of the instruction cache of the second level must be at least as great, the opposite of not smaller, as the block size of the instruction cache of the first level. The same counts if there's a data cache on both level. Note that, because a unified cache on one level implicitly has also the corresponding cache type, this counts also. For example, if you have an instruction cache on the first level, and an unified cache on the second, then, because a unified cache also counts as an instruction cache, the block size of the UL2 is a least as great as its part

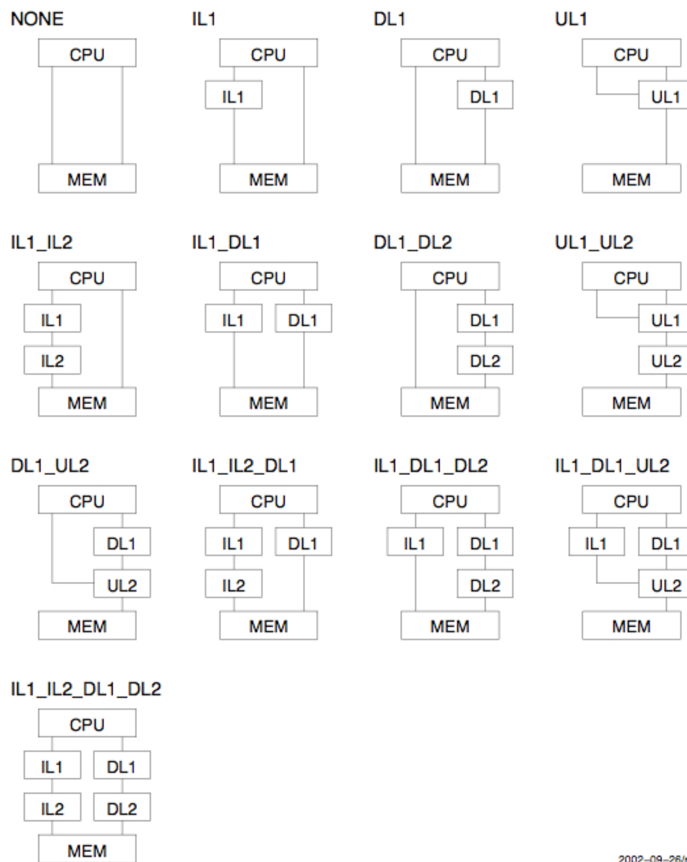

Figure 3.1: All by SimpleScalar allowed cache combinations. The one with DL1 and UL2 makes it freezing.

of the IL1. Continuing with this logic, having two unified cache levels unified, which means having a UL1 and UL2, then the block size dependency counts also here. The block size of the UL2 is greater or equal then the block size of the UL1.

Having seen all difficulties that are expecting us, we have to deal with the problem step by step. First, we have five cache type on each level, including none. This means, we can create an initial cache combination by randomly choosing a cache type for each level with also random values for the parameters of each cache type. This product is likely to be unfeasible. Secondly, we transform the cache type pair into a feasible one by removing single caches. So if there's no cache in the first level, then no cache in the second level is allowed. If we have a unified cache in the first level then the second can't be splitted. So here the second cache is removed.

While this method clearly produces cache pairs with a higher amount of absent partial caches, it's a multi-step method that produces - by cache types - feasible caches, as proposed above.

The third step is to deal with the block size dependency. Here a simple method is to manually adjust the block sizes of all caches in the second level. Depending on the internal cache configuration representation, this can increase the overall cache size, leading to a unfeasibly too large cache size, dependent on how feasibility is defined here. The focus in this section was to present a three-step method to produce a feasible cache by its criteria. It's an completely different topic what problems this else might produce elsewhere.

3.2 Mutation

After having seen how random elements are created, it should be easier understandable, how certain parameters of them are mutated. Again, we have to deal with the separation of simple and complex objects, between independent and depended ones.

3.2.1 Simple parameters

Simple objects like numbers can be mutated by moving them one position up or down, or in the case of powers of two, by shifting them up or down, where shifting is understood as the binary multiplication or division. Or both of them can receive a completely new random value. The grain coarsed changing of a single parameter is called *macro level mutation*, while the fine coarsed one *micro level mutation*. So we have two different mutation operators[2].

[2]Random element creation can be understood as doing a macro level mutation on all parameters of an given element.

3.2.2 Complex parameters

Now let's see how it stands with some complex parameters, like the branch predictor. Because this is a multi-level parameter, changes can be made on different levels. A change on the first level will change its type. It's hard to see in advance if a single position change will lead to a better or worse branch predictor, meaning that a more or less, like in the case of numbers, doesn't exist. So changing a branch predictor can be only made by choosing a completely new one, like in the case of a macro level mutation. Continuing this logic, you can perform grain coarsed or fine coarsed changes on lower level parameters.

Having understood the mutation of the previous two parameter categories, and then creation of random cache elements, it's easy to see that you can do mutation on caches also on multiple subparameter levels. On the first level, you have to change the entire cache type. On the second level, you can already change one or two cache configuration. Two then and only then when you have a splitted cache with both instruction and data cache available. Later, after having mutated your both levels of your main cache configurations, you have to look to get again to a feasible cache composition, and this is meant by type with type transforming, by the block size dependency[3].

Summarized, you have a huge possibility of mutation operators. They can be grain or fine coarsed on each level. Analogue to the simple parameter case, a *macro level mutation* should be understood for a grain coarsed mutation on the first level, and the *micro level mutation* for fine coarsed mutations on the first level and for all mutations on lower levels. This names where chosen to indicate that mutations on the macro level are generally more influent and those on a micro level.

3.3 Cross-Over

3.3.1 Description and simple parameters

The cross-over variation operator is an operator that takes some parameters from one parent and the other parameters from the other parent[4], and then producing a child on this way, like the biological cross-over of genes.

Cross-Over for simple and independent parameters is not a problem, as the rules are clear and side-effects like damaged dependencies don't occur. For complex parameters this is different. We start again with the branch predictor and work then to the caches.

[3] It's recommended to leave this mutation possibility away in a implementation. Type changing mutation of caches will bring you enough fresh information into your population.

[4] A parameter is *either* taken from one parent *or* from the other, but not from both. Also every parameter must have a parental origin.

3.3.2 Branch Predictor

Let's recall again the parameter possibilities for the branch predictor:

- perfect
- taken
- not taken
- bimodal [BTC, RSS, BTB]
- 2-level [2LEV, RSS, BTB]
- combined [BTC, 2LEV, RSS, BTB, COMB]

Now comes the question up how to define a cross-over. If both parents have the same simple parameter, then the child will have it. However, this lucky case will occur seldomly in practice. When both parents have a simple branch predictor, and by the analogy from above these are generalizations of numbers, so the child will have one of this two possibilities. More complex is the case when the parameter loaded possibilities are brought to game. Now the child can have an entire copy of a parental branch predictor for all levels. Having only this cross-over, you won't have individuals with different parameters on lower levels then those which were created with the initial population. You have to live with the limitations your "big bang" has brought to you.

But both parents can have a multilevel parameter, where also the possibility exists, that this subparameters are of common type[5], or those that common in every branch predictor[6]. Here comes the possibility up to cross-over those common parameters, and take the noncommon from a parent. This can be understood that the child is a clone of one parent but takes some common information also from the other one[7].

3.3.3 Caches

For caches there are similar possibilities, but slightly different. First, the child can take over the cache type for each level. Secondly, if two parents have an instruction cache on a common level, which is the case when an instruction cache is available with or without the data cache, then the configurations of this instruction caches can be cross-overed. Similarly the same can be done when having common data caches on the same level. Of course, having a cache level unified gives you also the possibility to cross-over this caches.

As you have seen, there's a strong analogy in the cross-over of caches like with branch predictors, except that, you guess it, the caches of both levels have to be brought in a feasible state.

[5]like BTC and 2LEV

[6]like RSS and BTB

[7]It's easier for implementations to determine a parental origin in advance like the first parent and the to look for common parts in the other parent.

3.4 Summary of EA aspects

In the last three sections a huge number of possibilities was presented how to create and, especially, variate processor configurations. This possibilities can be extended as one likes. They where presented here independently of a chosen representation of a processor configuration to separate the *what* is done from *how* it is done.

Chapter 4

Realization and Implementation

In this chapter all aspects regarding a realization of the previously presented subjects is discussed. This considers various paradigms, guide lines, and also the presentation of JavaPISA, an implementation of the PISA protocol in Java.

4.1 Realization criteria

Every work made has to serve certain critieria. Here, not *what* is realized is described, it's *how* it's realized. A list of keywords is shown with the description what is understood of them:

Completeness Completness means that no, or nearly no, compromises are made. Every compromise you make offers others, and often also yourself, a critique point and a weakness for reusing your work. Many achieved problem solutions had to be thrown away because they showed massive leaks due to strong approximations.

Syntax A strong syntax of the programming language you choose reduces the complexity and enables the compiler to exclude many potential errors at compile time. A good example of preventable bugs was found in SimWattch, file power.c. There, multiple global variables were defined. The author reused them often. A refactoring of the code with introducing local variables has shown that certain function values are computed but never used. Was this a result of too many copy-and-paste operations or avoidable mistakes? It's the last one. Therefore, one should use all possibilities for avoiding mistakes, included a strong programming language.

Common platforms Under this is not a common OS understood[1]. Here is meant that people should prefer using commonly accepted things like a

[1] The convergency of many UNIX OS could lead to this but it's not a subject here.

common programming language, concepts of whatever kind and so on. It's no longer acceptable that every one comes with his own wishes and excuses himself for that the others are too lazy or stupid to work into his field. Instead of putting everybody's energy into horizontally (managing the diversity), people should focus of verticallity (quality and progresses). Many progresses where made in the last years, which are often unused and no excuse stayes of not using them.

Reusability Reusability is also the result of multiple efforts and sometimes hard to define. But reusability is meant that projects are made with the idea in background of really being reusable; this by the idea, by the created code and so on. Completeness, common platforms and simplicity can lead to this.

Simplicity Simplicity is not a present for less gifted people. Simplification is meant as the information keeping simplification, and not the destructive one where generality is lost. Unsimplified results like software consume much worthful energy, which can be used better. Simplicity, together with testing, will enable you to remove many potential errors, but not all.

Factorization Here is not the mathematical factorization meant. It's more meant the common parts of a code should be unified. A good factorization technique is also object oriented programming (OOP). Here, superclasses keep common code, and subclasses define the small amount of necessary functionality. A good effect of realizing something in the OOP way can be seen in the difference between the implementations PisaLib and JavaPISA (see below). The first one is in C, the second one in Java. Instead of pasting your code into an other one, it's much better to focus on the missing parts in a simple file. Also a change in the common parts won't have great impacts into the specific parts when implemented in the object oriented way. Instead, if a bug is found in PisaLib, all users of it have manually to adapt their code. With centralizing the code this won't happen[2].

Testing Testing and simplicity are the two key components for an error poor software. Testing is a necessary part of software or any product development. It can't be replaced by a good design, as inexperienced people thing and how reality shows, because also the designer self is a error source.

For a successful realization of all this criteria, the programming language Java in its latest version, 1.4, was elected. Version 1.4 supports assertions, a key feature to check pre- and postconditions.

Remark: Many criteria are hard to reach and also the author knows that he is judged by the same laws like others. Nevertheless, they had to be shown and defined. In the conclusion you will see that most of the enumerated criteria where indeed reached.

[2]The original PISA inventors had first the problem to create something like PISA and the PisaLib. This comparison shows only the gain you can make by using appropriate techniques.

4.2 Model

Here two realization paradigms are presented and discussed. The first one was used earlier in the thesis, the second one later. To be precise, a switch of the first one to the second one was made.

4.2.1 Procedural

The **procedural** implementation has the focus on *how* things are done. This means an realization model is thought out. Then a data structure is found, and the whole realization is oriented toward this data structure. This approach was first made, as it appeared naturally for our problem.

We assume that every parameter of the processor configuration has to be filled with an object, its value. An object can be a number, a power of two, a branch predictor, a cache type or cache configuration or whatever you want. We further assume that an array of objects is also an object, a parameter object if you want. Of course, single objects should be distinguishable from object arrays, and those between themself by their number of elements. Such an object can be seen as the Java class `Object`, but we won't focus on a specific programming language or a concept. Most important precondition is, that if a parameter contains one object, it doesn't contain an other one. So this temporary exclusion, doesn't matter if it's seen as trivial, must hold.

This above defined object is used to store different sorts of parameter values. Like mentioned this can be a number for the integer unit count. Or the cache configurations of the instruction and data BTB can be stored so. Branch predictors can be represented so. For those, when having one of the simpler types, we have to store their type, let's say a unique number per predictor. For the complex ones, we can store, additionally to their type, their parameters, which are arrays of objects.

When storing the two cache levels, we can use no object for no cache[3]. The cache levels can be stored into a single variable or into two separate variables, where no cache at all means void entries. Further, if we have a unified cache type in one level, we store directly its configuration, which is an array. If we have a splitted cache, we represented it by an array with two elements, and each element has either no entry or a configuration of the cache it represents (instruction or data cache), but not two empty entries if there's no cache at all. So we can perfectly distinguish whether we have no, a unified or a splitted cache.

This data structure has some great advances. First, it's natural for the problem we deal with, and secondly it allows a unique storage, and therefore identification of a certain parameter value, because storing a value automatically overwrites and replaces the previous value. However, dealing with this data structure is a pain. Each time you have to read a value or identify it's type you have to make an extensive case analysis. And this every time you use is, whether its for the I/O, the mutation, cross-over or for any reason. You see, that this case

[3]Assume the existence or absence of an object is also allowed.

analysis enlarges your realization complexity, and this hits directly to a high bug number, where you risk that your project never finishes due to this handicap.

Therefore is the term procedural selected. We are focusing on how we deal, here considering toward a data structure.

This procedural realization model was taken at the beginning but was dropped exactly due to this extensive case analysis. We now will look at a better method.

4.2.2 Functional

The **functional** implementation has the focus on *what* is done, oppositely than *how* it's done. This means it's more important that we get the information of a certain cache type, that we can use a certain branch predictor, that we can generally use a service, rather than to deal each time how we have to manage it, like it stands in the focus of the procedural way. The functional implementation is a paradigm that keeps the jobs done on a higher, more abstract level. Basically this high abstraction level is never left. A lower abstraction level work would deal more with data structures. An data structure API belongs clearly to this lower abstraction level, because the data structure is in the focus. Functional realizations simple don't care about data structures, because data structure dteal with *how* something is stored, rather than *what* is stored.

Functional viewing is a paradigm as object oriented is. And like the last one, also programming languages where developed for this paradigm. They're not called C++ or Java, they're ML, PROLOG[4] and so on. See for more information in [11]. This functional implementation paradigm has help drastically to simplify the realization. The transformation was done quickly. The result is encapsulated from, and therefore, independent of any data structure.

4.3 JavaPISA

4.3.1 Introduction into JavaPISA

Because the programming language Java was chosen to implement the thesis software, and PISA had to be used, a realization of the PISA protocol, mainly the variator supporting part, had to be implemented in Java. Furthermore, some additional goals where chosen for a Java-based realization of PISA, called JavaPISA:

- OOP (to remove all common parts from the final variator)
- usage of data structure knowledge like hash based versions (HashMap in Java)

[4] Deals mainly with logic programming.

- life long identification numbers for individuals (to better understand their evolution career)
- use all strengths of Java (like libraries, exception handling)
- snapshotting (storing each created generation individually for evolutionary research)
- keeping the PISA consistence when the variator crashes
- simplicity in the realization
- small usage effort (quick understanding what and how to implement)
- maintainability

An excellent introduction and description about many sorts of heaps, and especially the hash based one, can be found in [12]. The Java class `java.util.HashMap` is described in [13], also all Java features. Life long identification numbers for individuals can be reached by a ticket system. This means every ever created element receives a unique number. Snapshotting is good when you want to see how the evolution process progressed. You can use the created files to see each generation in a presentation program like Gnuplot[14]. There you can get a movie which shows the dynamic of the evolution.

PISA can become inconsistent if a participator, the selector or variator, terminates in an unexpected behavior like a crash or assert. A possible problem is that the other part keeps running forever (until you stop it manually, of course). An other problem is that the recorded PISA state has a value, which, when you want to restart the PISA partners again, will lead you again in an illegal state. For all this reasons, any many more, JavaPISA catches, with the help of Java's exception handling system, all internal exceptions and terminates into a consistent state[5]. Of course, the caught exception can be stored for debugging, if you want.

Figure 4.1 shows the main structure of JavaPISA. PisaVariator is the main class. It represents a PISA variator. HashPisaVariator is a HashMap based realization of PisaVariator. Their helper class in PisaCommon. Those three classes contain about 90% of the functionality for a successful realization of a PISA variator, like LotzVariator realizes the LOTZ example found in [2]. ProcessorVariator is the variator that represents a processor variator, that, what this thesis is about.

The last figure was very rough. The are more classes in the context of JavaPISA, like the next two figures (4.2 and 4.3) show.

In this two figures we see also a class PisaSelector, which indicates how a completed PISA selector would fit into JavaPISA. It's easier to see so that PisaCommon holds all common parts of a selector and variator. RandomComparator helps sorting a group of values like a table in lexicographical order.

[5] At this time only one variator supports this feature, but on a very global place.

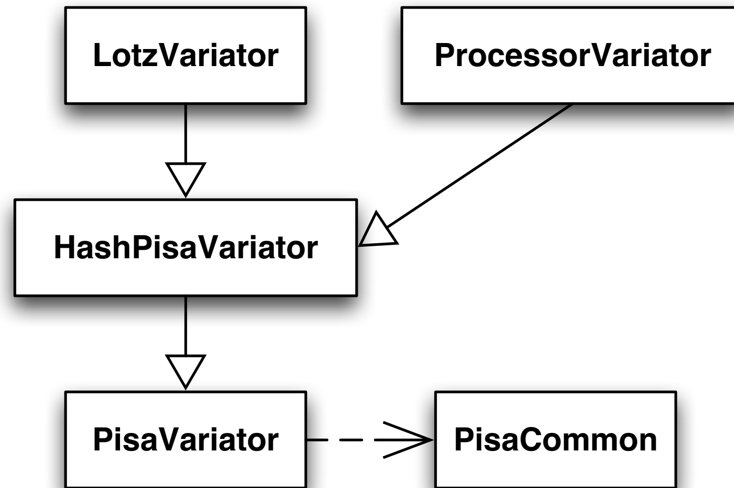

Figure 4.1: Basic JavaPISA classes with final variators LotzVariator and ProcessorVariator. All final variators have to implement HashPisaVariator.

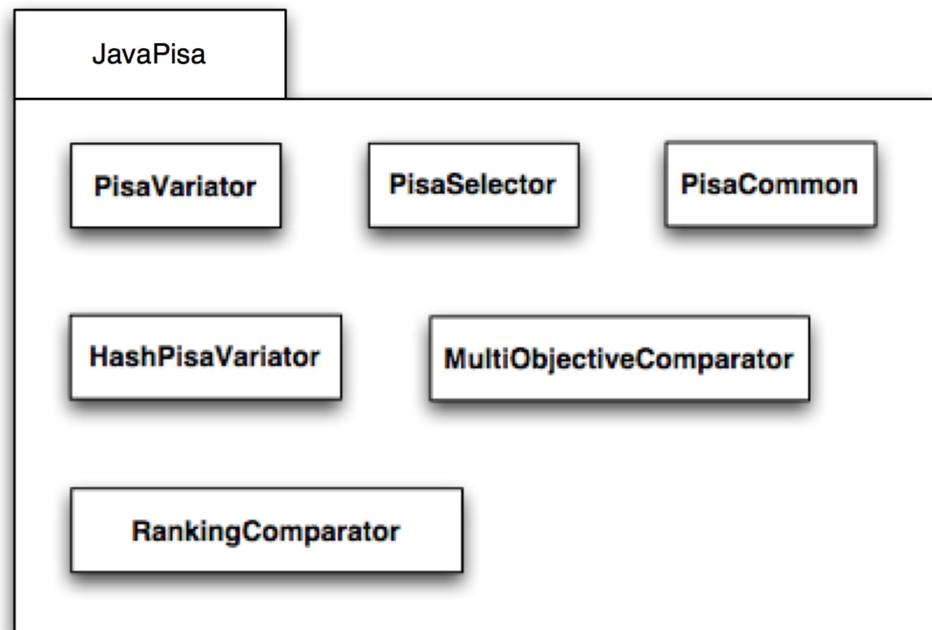

Figure 4.2: The package JavaPISA with its classes.

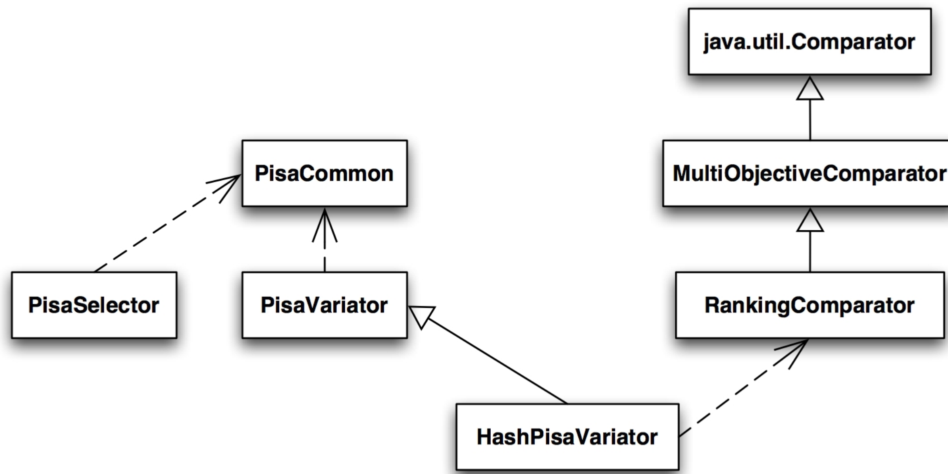

Figure 4.3: The inheritances and usages in JavaPISA.

4.3.2 Variators with JavaPISA

While the structure of JavaPISA is nice to see, it's more important how you can use it. Therefore, a quick introduction into the implementation of a variator is presented. More can be seen when taking a look into the files.

The following methods *must* be completed to have a minimal variator:

computeHashCode Returns the a numerical identifier of a single element in the pool.

newRandomElement Creates a new random element.

evaluateElement Evaluates an element by all its objectives.

applyVariatorOperator Variates a given population.

stopCriteriaReached Defines a stopping criteria for the optimization process.

The constructor must call the method `actionLoop()` so that the optimization process can begin. Therefore, your variator's constructor can look so:

```
public DummyVariator(String parameterFile, String fileNameBase,
double pollIntervall) throws Exception
{
super(parameterFile,fileNameBase,pollIntervall);

// enables snapshotting
buildSnapshot=true;
```

```
// starts the optimization process
actionLoop();
}
```

The optimization process starts automatically when a new object of your variator is created. The parameters `parameterFile` , `fileNameBase`, `pollInterval` are the same as for each variator presented on [2]. Anyway here a short presentation: `parameterFile` is the file name of the file containing the parameters for your variator[6], `fileNameBase` like `PISA_` is the prefix of all files a PISA selector like spea2 needs to work with, and `pollInterval` is the time interval for the variator to poll when awaiting the selector to have finished, so that he can continue again.

Remark: Note that a thorough introduction into JavaPISA (with or without [2] and [3]) was *not* a core task of this thesis and thesis report. So you have to be familiar with PISA and some implementations of PISA to use JavaPISA. Also the code should be enough documented to work into an example variator like LotzVariator to understand what is meant, if you're also a little bit familiar with Java.

4.4 Classes and their collaboration and workflow analysis

Figure 4.5 shows the basic packages and classes used in the thesis software, without JavaPISA. Figure 4.6 shows the major dependencies between the classes of the thesis software.

There's a JavaDoc-based description of each class, and also the code is self-speaking. Here some short description anyway:

AbstractProcessorConfiguration Processor configuration with simple parameters and an API for complex ones to realize the functional implementation paradigm. All classes work with this class when they have to work with processor configurations.

ProcessorConfiguration A concrete processor configuration. Realized by the procedural paradigm.

EvaluateConfiguration Evaluates a processor configuration with SimpleScalar or SimWattch.

CactiBasedAreaComputation Estimated the area consumption of a processor with CACTI and die size informations.

ProcessorConfigurationWriter Abstract class for writing processor configurations to console and files.

[6]It's up to your responsibility to use this option. It's not included in JavaPISA to support it.

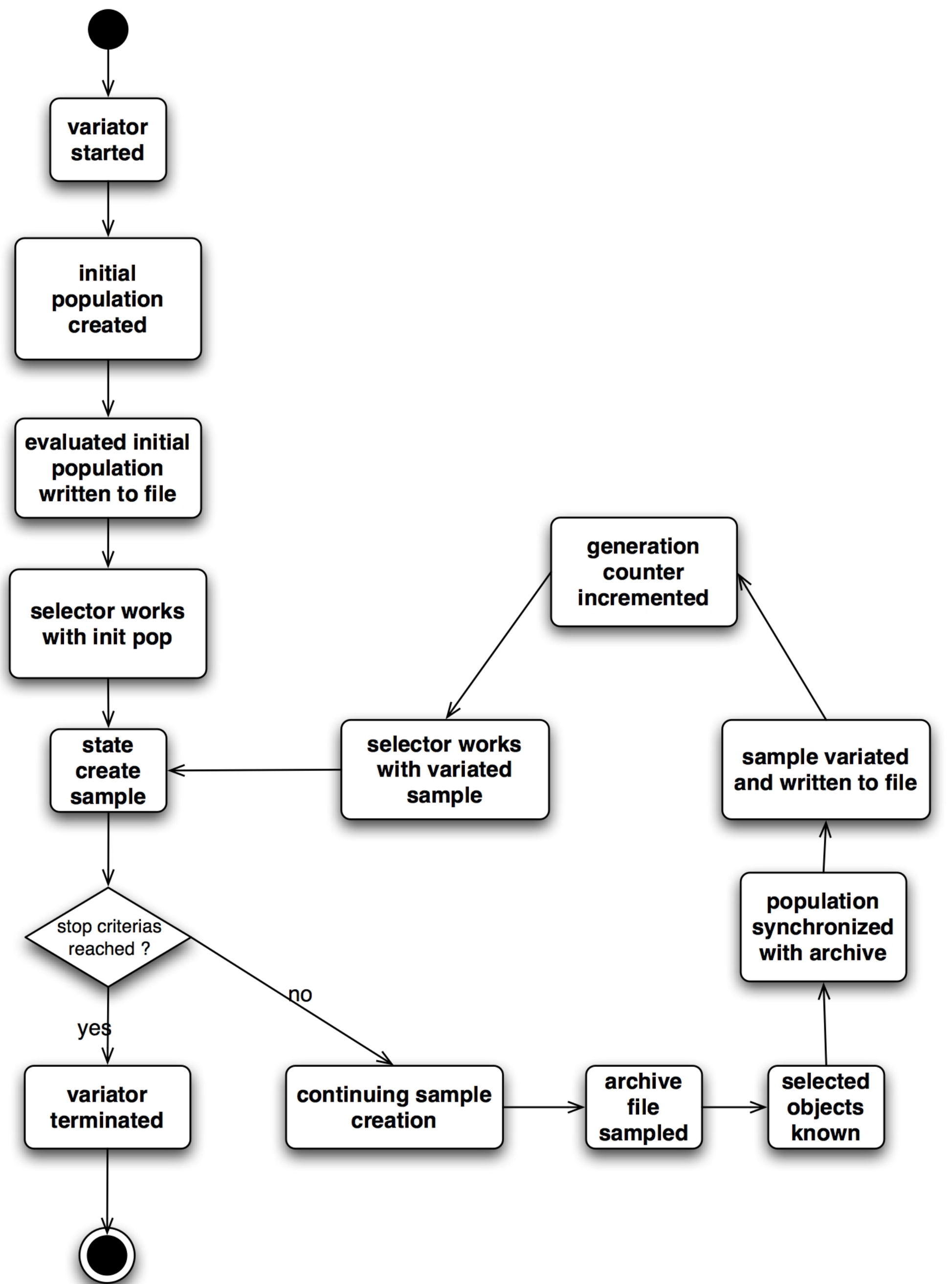

Figure 4.4: Basic states from the variator view in JavaPISA.

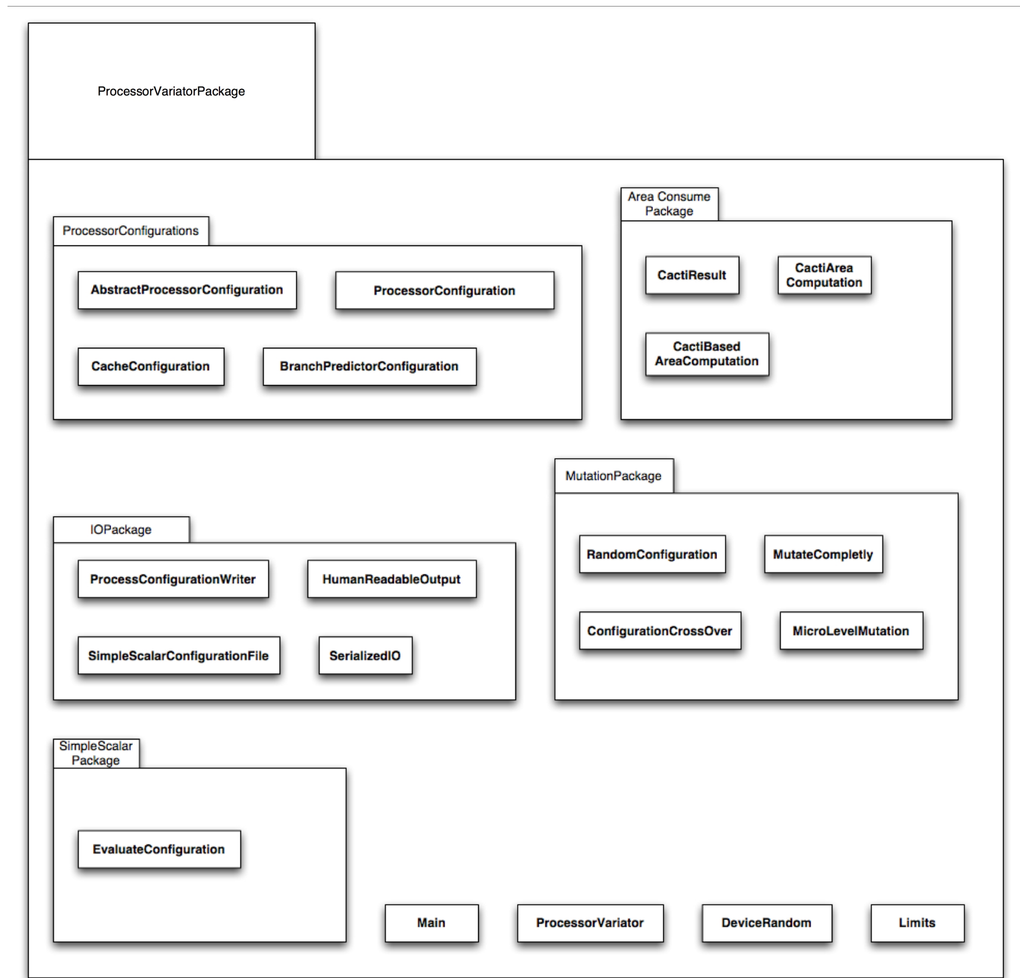

Figure 4.5: Basic packages and classes used in the thesis software (JavaPISA excluded).

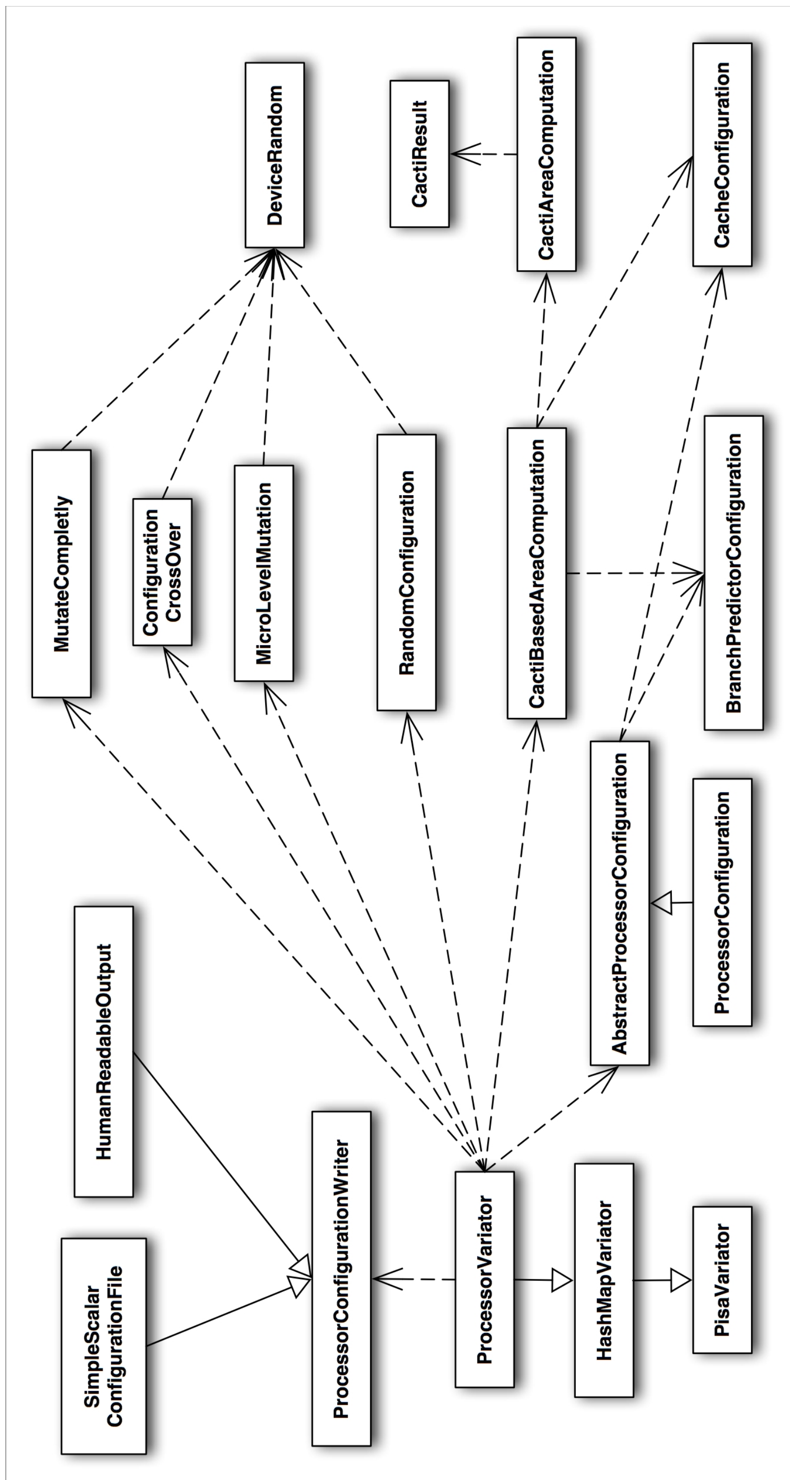

Figure 4.6: Dependencies of the important classes.

DeviceRandom A random number generator which can be initialized by /dev/random to be non-repetitive.

RandomConfiguration Creates a random processor configuration.

MutateCompletly Does a macro level mutation on a processor configuration.

MicroLevelMutation Does a micro level mutation on a processor configuration.

ConfigurationCrossOver Cross-over two processor configurations.

4.5 Miscellaneous

In this section some things were described which are worth being mentioned but don't fit into other part of the documentation.

4.5.1 Processes and their output

To read the results of the tools like CACTI, SimpleScalar or SimWattch, you must somehow obtain them. While the SimpleScalar-family gives the possibility to parse them from a result file, so doesn't CACTI. Here is shown, with CACTI as an example, how processes are created, their inputs parsed, and then terminated. Assumed is that the parameters for the program to execute are in the string `cactiParams`.

```
// create a new process
Process cactiProcess=Runtime.getRuntime().exec(cactiParams);

// await its termination
int returnValue=cactiProcess.waitFor();

// read the its output
InputStream inStream=cactiProcess.getInputStream();

// parse it
CactiResult result=new CactiResult(inStream);

// terminate the process explicitely
cactiProcess.destroy();
```

4.5.2 Parallelization

Evaluations of time consuming benchmarks are a good candidate for a parallelization. This means that the SimpleScalar-family can become the most time consuming part of the evaluation process, so it's worth doing it on multiple

machines. The other parts, especially the area consumption with CACTI, take few time per element to proceed. However, it can be a future task to evaluate multiple bunches of them at multiple machines instead of doing it serially in an for-loop and letting it becoming the bottleneck in the execution process.

Parallelization can be made on two ways. The first is to divide up all jobs to all machines, then letting the machines doing their job, and then collecting the results. This means if you have 105 jobs and 15 machines, then each machine gets $\frac{105}{15} = 7$ jobs to do. The minus is that a single machine has to reserve many resources at once. The second method avoids this resource consumption. It distributes only one job to each machine and then awaits the termination of all. So the jobs are evaluated block wise. The minus of this method is a friction in the partitioning into blocks. This means the synchronization is responsible for a non-optimal usage of the available resources. The first method uses more resource at once, but as soon one process on a single machine is ready, then this machine can concentrate itself to the remaining ones and doesn't have to await for being feed with new jobs. The first method is also the preferred one.

A minus in the current implementation is that is a single machine needs too much time for their tasks, it becomes the bottleneck. However, the implementation was simple and a derivation of the previously described process activation method.

Chapter 5

Optimization results

Optimization results are made on the following benchmarks:

rijndael-enc Encodes a given input by the Rijndael (AES) algorithm.

adpcm-enc Encodes a given piece of sound.

frag An internet based benchmark.

helloworld A simple program printing `HelloWorld!` on the console.

Optimizations where made with all this fore benchmarks. The size of the population was 100 elements, and per turn where 45 new elements created. The optimization was made over 100 turns. The variation operators where the crossover and the mutation. Mutations where made on the macro and micro level. All three variators appeared in this order. Note that the created results should *not* be looked as being perfect. The doubts described in the area consumption parts of this thesis show that the created results should be taken with care. However, most important is to see if the tendency of the optimization leads to plausible results. So should the optimal processor for helloworld be quite minimal in the number of units and in the cache sizes. On the same way should rijndael produce not a FPU loaded processor as not FPU unit is needed there. Similarly frag and adpcm shouldn't prefer cacheless processors.

Because power consumption isn't available as a third objective like explained in the appropriate part of the report, optimizations where only made for the two objectives performance and area consumption. So we have to look for elements that are very good in either or even both optimization criteria and to see how they look. Also interesting is the speed of convergence of the non-dominated elements toward the Pareto front. The results are presented benchmark-wise with some example contents. All are saved into apropriate files.

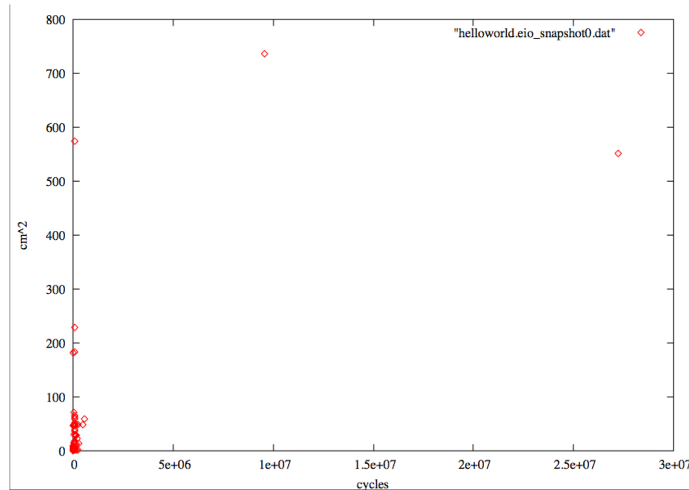

Figure 5.1: HelloWorld at initial population.

5.1 HelloWorld

For HelloWorld, three elements of the Pareto front where picked out and compared:

4169 good in both criteria, 5 INT ALU, 2 L1 cache ports, cacheless, 8 KB ITLB, 16KB DTLB

4552 minimal cycle count, like 4169 except 128 KB ITLB

3779 minimal area consumption, 2 L1 cache ports, 8 KB ITLB, 1 K DTLB

Comment: HelloWorld needs no L2 cache, which seems reasonable. At least 8 KB ITLB seem to be a must. The 2 L1 cache ports have, of course, no influence on a cacheless configuration, and stayed because they where unpunished.

5.2 Rijndael-enc

For rijndael-enc, three elements of the Pareto front where picked out and compared:

4133 minimal cycle number, 6 INT ALUs, 2 L1 cache ports, no instruction cache, 4 KB DL1, 64 KB DL2, 32 KB ITLB, 16 KB DTLB

3633 minimal area consumption, minimal unit count, 4KB UL1, no L2 cache, 1KB ITLB, 512 B DTLB

4546 good in both criteria, 2 INT ALUs, no instruction cache, 4 KB DL1, 64 KB DL2, 1KB ITLB, 512 B DTLB

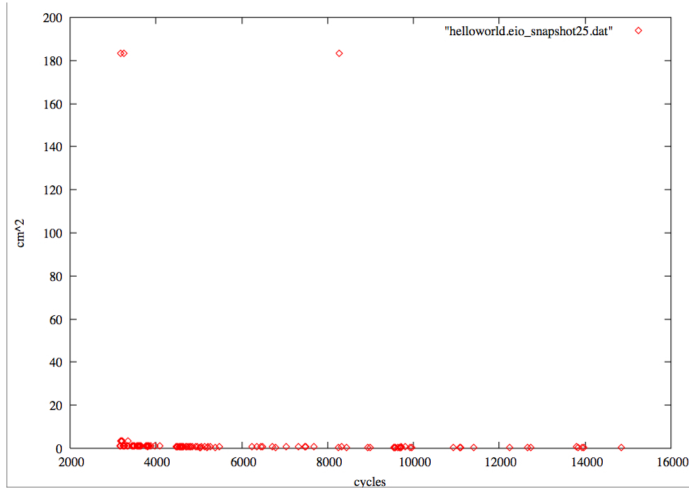

Figure 5.2: HelloWorld at 25th generation.

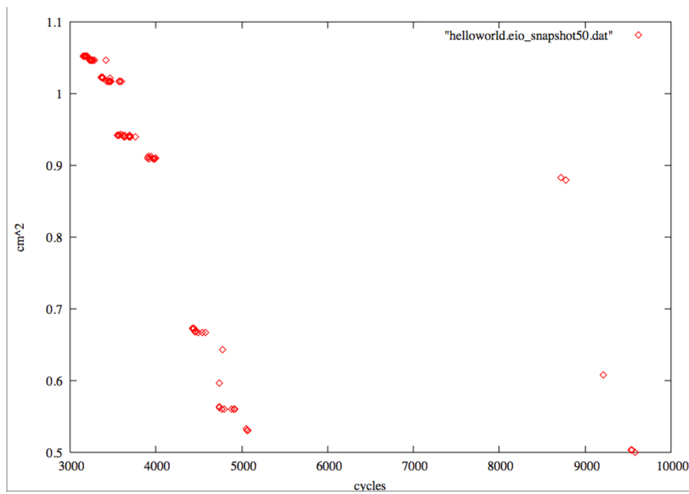

Figure 5.3: HelloWorld at 50th generation.

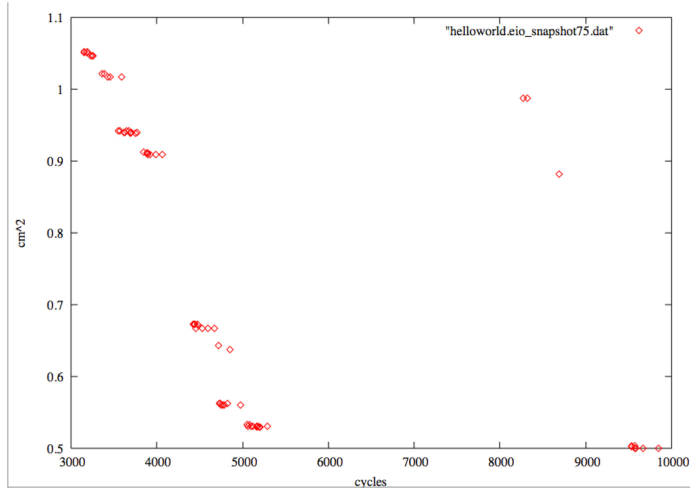

Figure 5.4: HelloWorld at 75th generation.

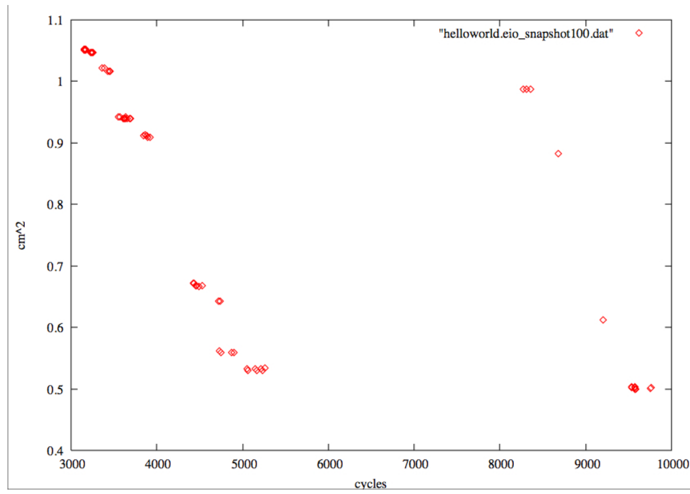

Figure 5.5: HelloWorld at 100th generation.

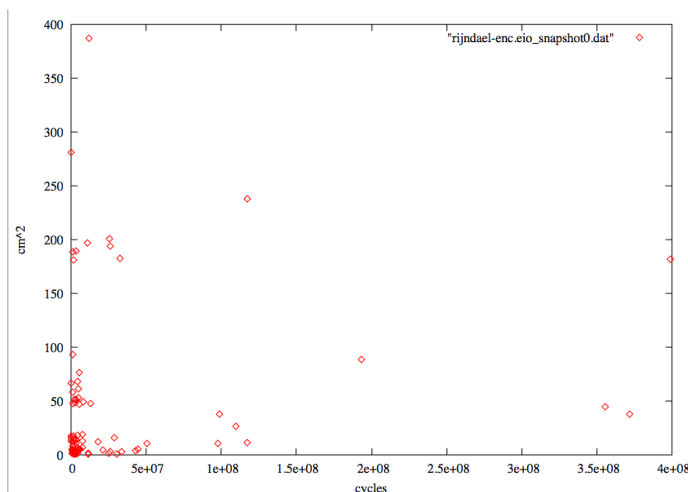

Figure 5.6: Rijndael at initial population.

Comment: It's reasonable that nearly no instruction caches are required, because rijndael is loop intensive. The presence of data caches is a must. Astonishing is that 4546, the on which is good in both criteria, is a mixture of both other extremes.

5.3 Adpcm-enc

For Adpcm-enc, three elements of the Pareto front where picked out and compared:

4338 good in both criteria, 3 INT ALU, 2 L1 cache ports, no IL1, 8KB DL1, no L2, 8 KB ITLB, 8 KB DTLB

3630 minimal cycle number, like 4338 but with 16 KB ITLB

4107 minimal area consumption, minimal unit count, 1 L1 cache port, caches like others except 128 B DL1

Comment: adpcm seems to be very locality oriented as it dislikes a L2 cache and prefers instead of that a multi-ported DL1, because no IL1 is available. 8 KB ITLB are common.

5.4 Frag

For Adpcm-enc, three elements of the Pareto front where picked out and compared:

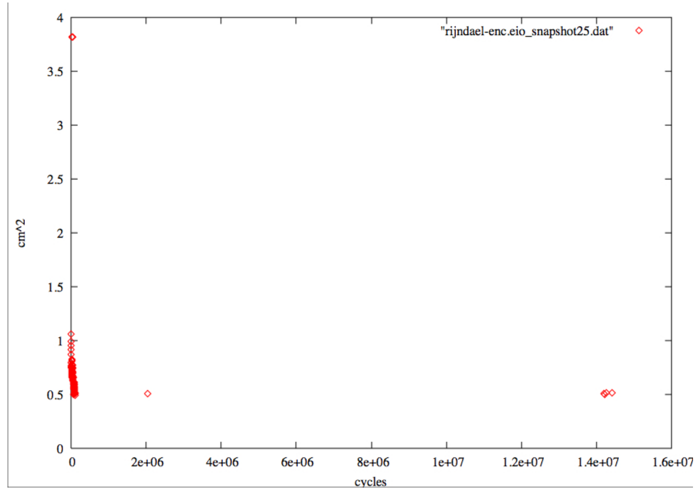

Figure 5.7: Rijndael at 25th generation.

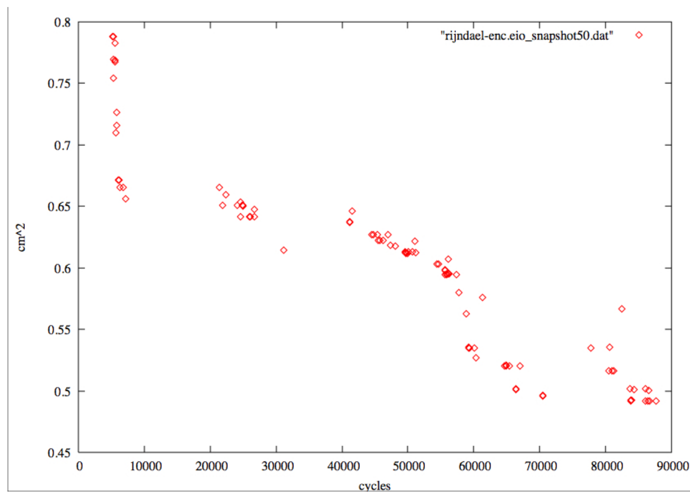

Figure 5.8: Rijndael at 50th generation.

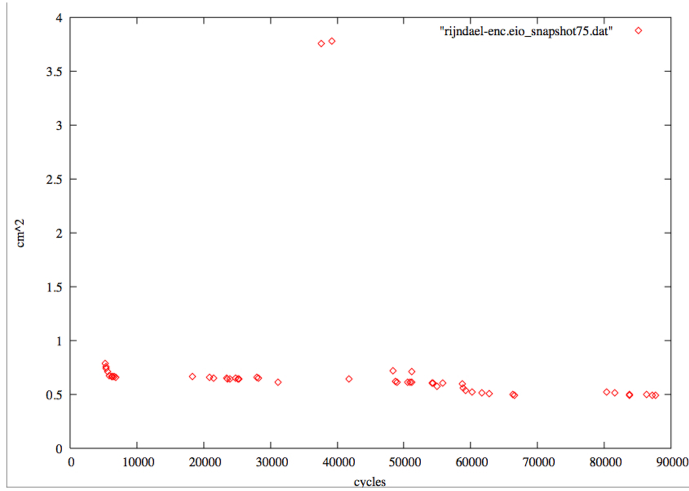

Figure 5.9: Rijndael at 75th generation.

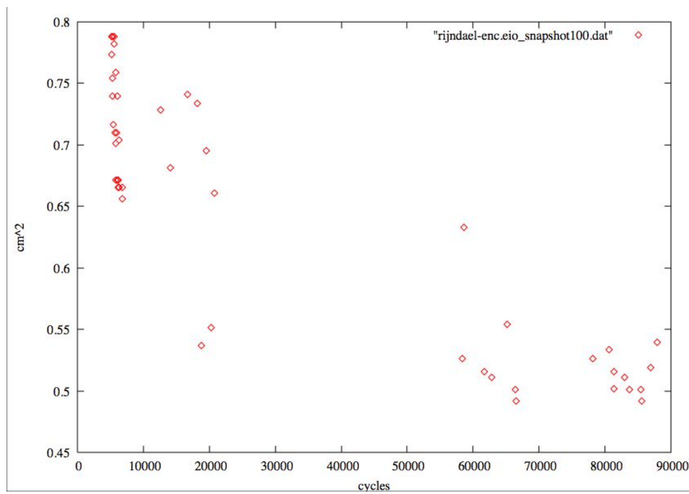

Figure 5.10: Rijndael at 100th generation.

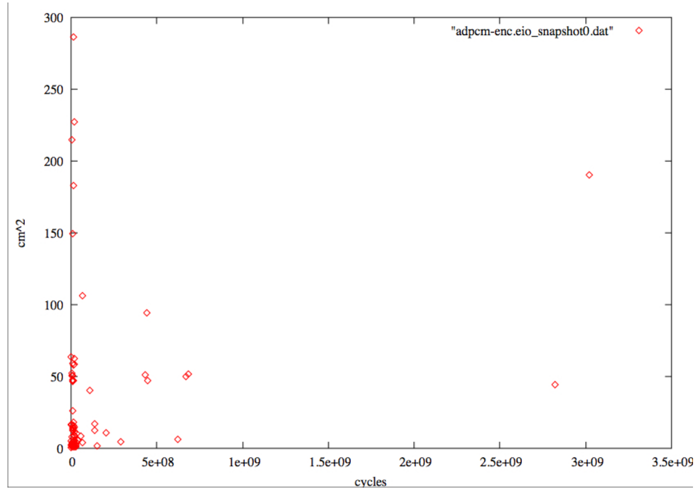

Figure 5.11: Adpcm-enc at initial population.

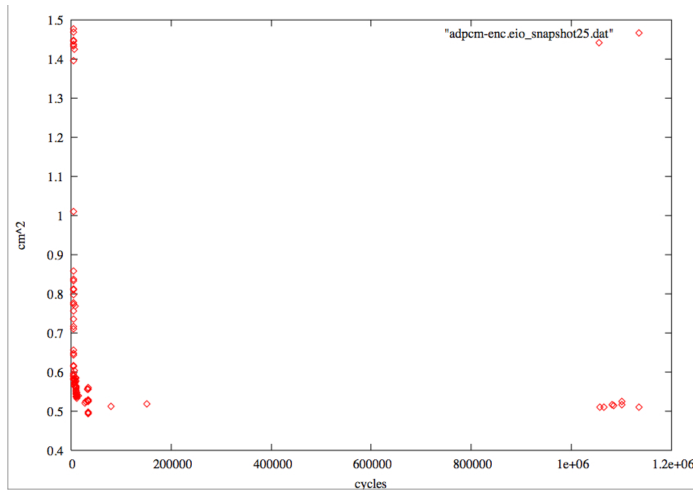

Figure 5.12: Adpcm-enc at 25th generation.

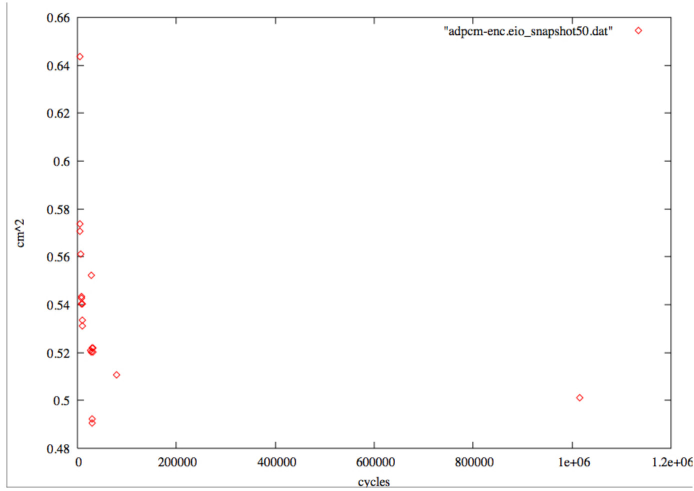

Figure 5.13: Adpcm-enc at 50th generation.

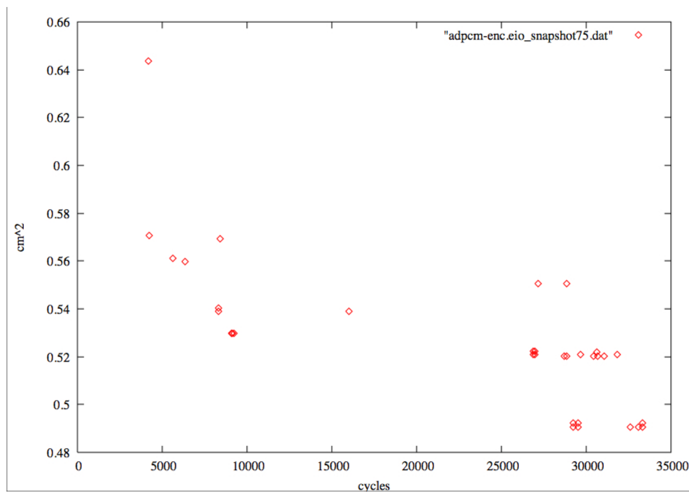

Figure 5.14: Adpcm-enc at 75th generation.

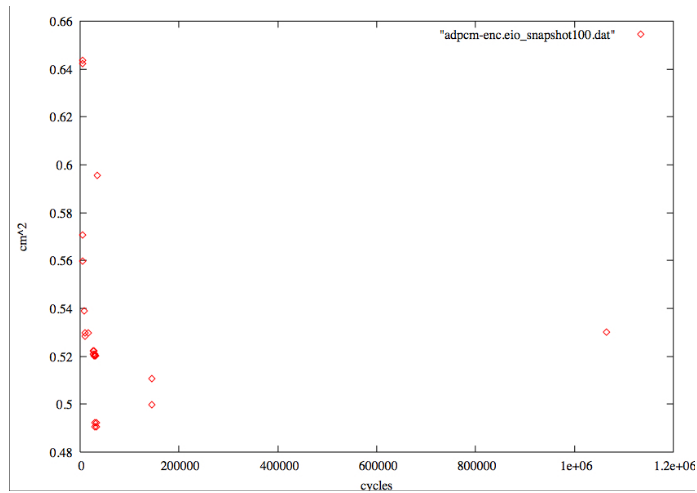

Figure 5.15: Adpcm-enc at 100th generation.

4134 good in both criteria, 2 INT ALU, no IL1, 256K DL2, no L2, 8KB ITLB, 4KB DTLB

4487 minimal cycle count, 3 INT ALU, same cache like 4134 except 512 B DTLB

2960 minimal area consumption, 2 INT ALU, 8 KB IL1, no DL1, no L2, 8 KB ITLB, 512 B DTLB

Comment: Level 2 caches seem to be superfluous. 8 KB ITLB are common. Also a non-minimal amount of INT ALUs is required.

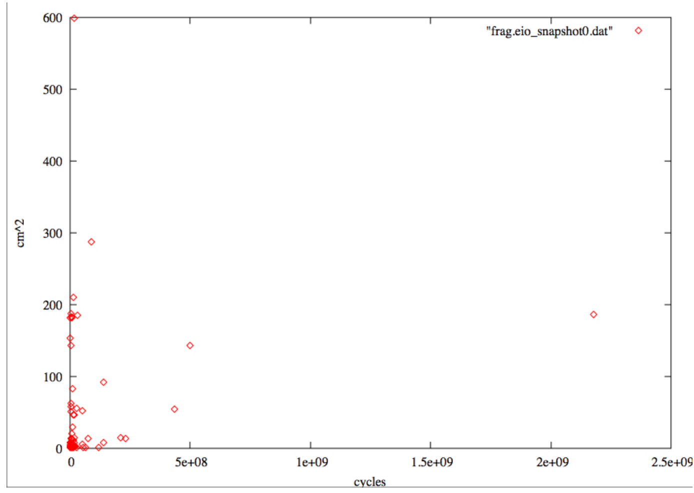

Figure 5.16: Frag at initial population.

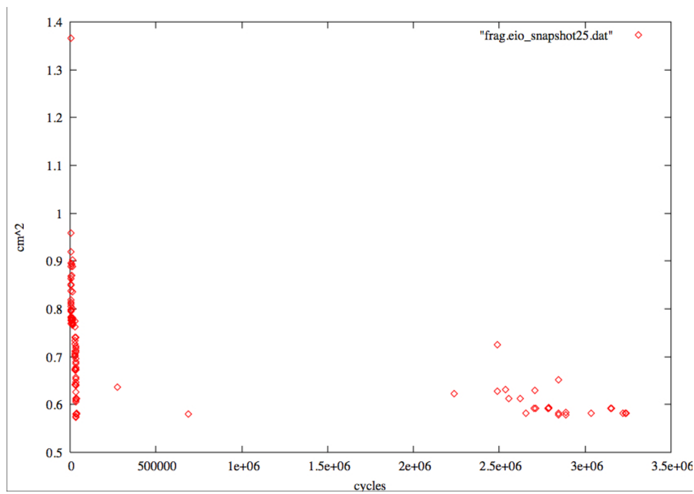

Figure 5.17: Frag at 25th generation.

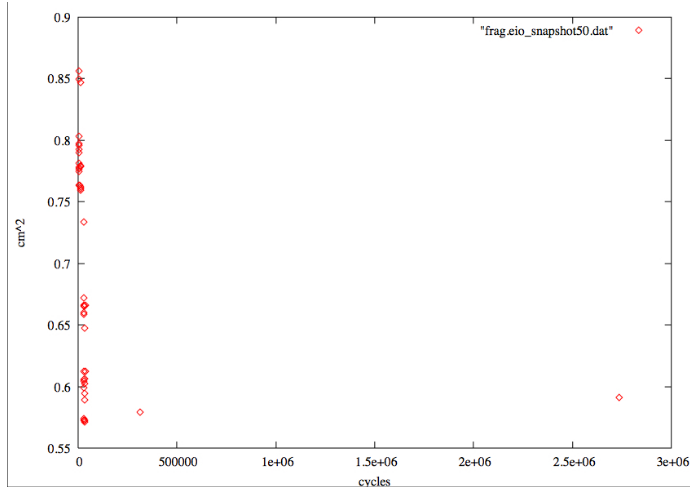

Figure 5.18: Frag at 50th generation.

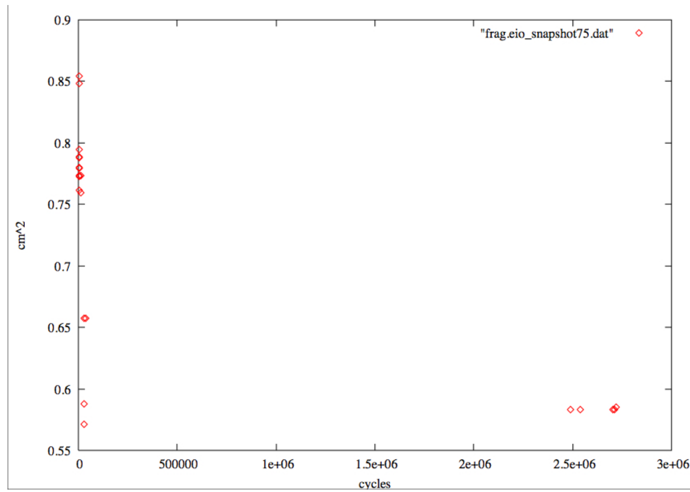

Figure 5.19: Frag at 75th generation.

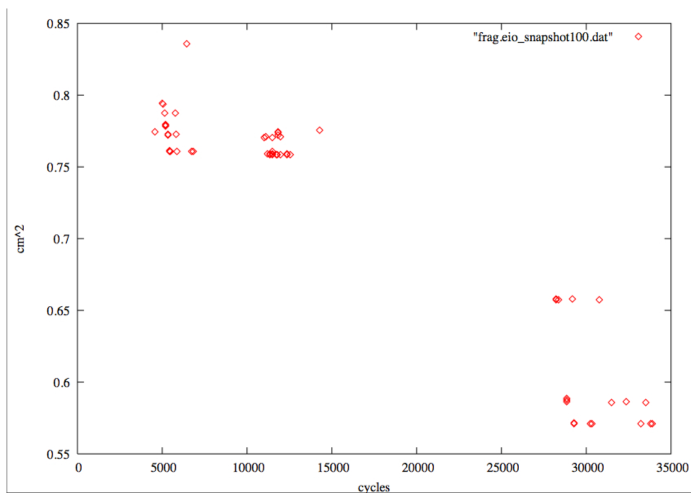

Figure 5.20: Frag at 100th generation.

Chapter 6

Conclusion

The demanded goals where reached, and also in the way the realization guideline describes. Optimizations showed that the software, and the concept it's based on, do it. However, it's unrealistically to await perfection from about 5'600 lines of code[1] made by one person and not reviewn by an other one. Hidden errors, those where your software works but doesn't crash, like also forgotten parameters, can happen. Simplicity and testing effort helped to reduce them, but couldn't do it for all.

Preworks from others like cache limits from the made semester thesis or the PISA protocol were fully integrated. However, the wish on trying to do parallelization on the preferred way like with a favorite programming language (here: Java) can turn into a trap, because the method you choose can bring you problems somebody else has solved already. However, the view to implement and realize most things from scratch has shown as the right way, like JavaPISA demonstrates.

The uncertainty aspect of objectives, meaning that you have to deal with imprecise function values, couldn't be brought into the model because nothing right exists at the moment. Also the deficits of the area consumption, even if they're regarded as small, should, together with those from power consumption originated by SimWattch, be removed in a succeeding work. The author believes, that, together with a good solution of the uncertainty problem, then a very well processor optimizing suite will be created.

The current model is quite well. But as we have on one side existence dependent parts like cache types, and on the other side an integer linear problem[2], then there's a possibility to unify this aspects to a more abstract and simpler level where also the variation operators can be understood and realized simpler.

The author believes to that the work done is enough for a succeeder to make a quite well project.

[1] Earlier there were over 6'000 lines. But the debug code and unnecessary code was removed.
[2] we look the all dependencies and logarithmize the powers of two

Acknowledgment: The author would like to thank his tutors for their cooperation and brought in effort.

Appendix A

Processor configuration

The processor configuration is divided into multiple parts. They are: processor core, memory hierarchy, branch predictor and caches. In this appendix all parameters are presented, together with their equivalent in SimpleScalar and how to be used. SSP stands for *SimpleScalar parameter*, and these are written like SSP `-fetch:ifqsize`. All interval limits are found in the file Limits.java.

A.1 Processor core

FetchWidth Must be in $[2^0, 2^6]$. SSP `-fetch:ifqsize`

FetchSpeedRatio Must be in $[1, 4]$. SSP `-fetch:speed`

FetchBranchMispredictionLatency Must be in $[1, 10]$. SSP `-fetch:mplat`

DecodeWidth Must be in $[2^0, 2^{10}]$. SSP `-decode:width`

MaximumIssueWitdh Must be in $[2^0, 2^{10}]$. SSP `-fetch:ifqsize`

RuuCapacity Must be in $[2^0, 2^{10}]$. SSP `-ruu:size`

LoadStoreQueueCapacity Must be in $[2^0, 2^{10}]$. SSP `-lsq:size`

NumberOfIntegerAlu Must be in $[1, 8]$. SSP `-res:ialu`

NumberOfIntegerMulDiv Must be in $[1, 8]$. SSP `-res:imult`

NumberOfL1CachePorts Must be in $[1, 2]$. SSP `-res:memport`

NumberOfFloatingPointAlu Must be in $[1, 8]$. SSP `-res:fpmult`

NumberOfFloatingPointMulDiv Must be in $[1, 8]$. SSP `-res:fpmult`

Notes:

- NumberOfL1CachePorts has a CACTI limit of 2 and a SimpleScalar limit of 8. 2 was also chosen after a discussion.

- Values with upper limit, 10 have this limit because no reasonable limits was else found.

A.2 Memory hierarchy

MemoryAccessLatencyFirst Set fix to 120. SSP -mem:lat <1st> <next>

MemoryAccessLatencyNext Set fix to 15. SSP see above.

MemoryBusWitdh Set fix to 8. SSP -mem:width

TlbMissLatency Set fix to 30. SSP -tlb:lat

HitLatencyIL1 Determined by CACTI. SSP -cache:il1lat

HitLatencyDL1 Determined by CACTI. SSP -cache:dl1lat

HitLatencyUL1 Determined by CACTI. SSP see notes.

HitLatencyIL2 Determined by CACTI. SSP -cache:il2lat

HitLatencyDL2 Determined by CACTI. SSP -cache:dl2lat

HitLatencyUL2 Determined by CACTI. SSP see notes.

Notes:

- Because SimpleScalar doesn't directly support unified caches, the hit latency of the UL1 is expressed by setting the hit latencies of the IL1 and DL1 with the same value HitLatencyUL1. Else SimpleScalar crashes.

- Similar if a UL2 is available. But SimpleScalar seems to read the value of the DL2 latency and doesn't crash. Anyway, the hit latencies of IL2 and DL2 have also the same value HitLatencyUL2.

- The values for the memory access latency where proposed by the tutor Christan Plessl due to reflection.

- Memory bus width was set to 8 because actual most nowadays processors have this value.

- The TLB miss latency was taken like the SimpleScalar default value.

A.3 Branch predictor

A.3.1 Branch predictor parameters

Branch predictors are used and presented multiple times during the thesis report. Here's anyway a presentation, but with its own focus. First, there are six types of branch predictors. Some of them use the same sort of parameters when they (the branch predictors) are used.

- perfect
- taken
- not taken
- bimodal [BTC, RSS, BTB]
- 2-level [2LEV, RSS, BTB]
- combined [BTC, 2LEV, RSS, BTB, COMB]

To say SimpleScalar which branch predictor type you like to use you have to set the parameter `-bpred` with one of the following values and in this writing form:

- perfect
- taken
- nottaken
- bimod
- 2lev
- comb

The first three branch predictor types are simple ones. So its sufficient to set `-bpred` with their value, and no other branch predictor parameter more. If, however, you want to use a complex branch predictor, you have to set two of their commonly used parameters and some depending on the branch predictor type you use.

The common parameters are:

ReturnStackSize Must be 0 or in $[2^1, 2^6]$. SSP `-bpred: ras`

BtbSets Must be in $[2^3, 2^{12}]$. SSP `-bpred: btb <sets> <assoc>`.

BtbAssociativity Must be in $[2^0, 2^5]$. SSP see above.

If you have a bimodal or combined branch predictor, then you must write the bimodal predictor table size, which must be in $[2^0, 2^{12}]$. In SimpleScalar terminology: `-bpred:bimod`.

For a 2-level or combined branch predictor the necessary parameters are:

- Size of the first level table. Must be in $[2^0, 2^{12}]$.
- Size of the second level table. Must be in $[2^0, 2^{12}]$.
- History width. Must be in $[2^1, 2^{20}]$.
- Xor-ability. For allowing to xor the history and the address in the second level of the predictor. Must be in $[0, 1]$ (for no/yes).

The corresponding SimpleScalar parameter is `-bpred:2lev <l1size> <l2size> <hist_size> <xor>`.

Finally, there's a special parameter if you use a combined branch predictor: The meta-table size of the combined predictor. It must be in $[2^0, 2^{12}]$, and passed to SimpleScalar in the form `-bpred:comb`.

A.3.2 Area consumption

To compute the area consumption of a branch predictor, one must know which assumptions are taken to estimate their area consumption. The assumption are:

- perfect: 0 area at all
- taken: 0 area at all
- not taken: 0 area at all
- ReturnStackSize: 0 area at all
- BTB: normal cache (computed by CACTI)
- Level 2 table width: 2 bit per entry
- bi-modality table: 2 bits per entry
- combined meta-table: 1 bit per entry

The area consumption for the combined predictor, which also contains the bimodal and the 2-level predictor, can be best seen with figure A.1.

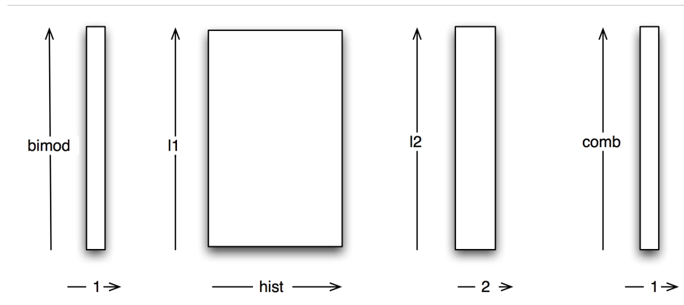

Figure A.1: Area consumption for the combined predictor including its bimodal and 2-level part.

A.4 Caches

Caches were introduced into the subsections about area consumption, and very detailed, in the random elements creation part. To avoid information duplication, you'll see in this section only and mainly a discussion about cache configurations (the lowest parameters).

Each cache configuration like UL1, DL2, instruction TLB, data TLB, consists of four parameters.They are:

number of sets Must be in $[2^0, 2^{13}]$

number of blocks Must be in $[2^3, 2^{12}]$

associativity Must be FA, for full associativity, or in $[2^0, 2^5]$

replacement policy Must be in $[0, 2]$ and stand for LRU, FIFO or random.

The number of sets, of blocks and the associativity must be chosen so that their product, the cache size, lies in $[2^6, 2^{23}]$.

There's a basic difference in the way how SimpleScalar and CACTI see a cache configuration. While SimpleScalar sees it as described above, except that it doesn't support fully associative cache explicitly, CACTI wants to see a total cache size from which it calculates the number of sets, but knows nothing of a replacement policy. Additionally, CACTI supports fully associative and direct mapped caches. The last one means that the associativity is fixed to 1, and CACTI knows no difference between an explicitly set associativity of 1 and a direct mapped one because it detects the single associativity case and interprets it as a direct mapped one.

Fully associative caches have a fix set number of 1, so is there definition. To support fully associative caches in SimpleScalar, the cache configuration creation, which is CACTI like by giving the cache size as a parameter instead the number of sets, calculates the needed associativity from given cache size and block size,

by dividing the cache size through the block size.[1]

The CACTI author Premishkore Shivakumar suggested that the maximal block size should be 2^9, but not larger than 2^{12}, because caches with this block size make no sense. A further problem with CACTI consists when a certain internal product can become larger than 2^{30} bytes (1 GB), then an arithmetical overflow happens and CACTI produces an exception on certain operating systems[2]. To avoid this overflow, for which CACTI can create wrong results, the maximal produced cache size shouldn't exceed the value 2^{30}, therefore a maximal number of sets of 2^{13} is used, because together with maximally 2^{12} blocks and an associativity of 2^5 the cache size limit of 2^{30} is never reached (12+13+5=30). However, all caches are internally at a size of most 2^{23} bytes (8 MB) created, because larger caches are not expected in the next future and would increase the simulation time unnecessary.

A.4.1 Unified caches with SimpleScalar

This is a special topic when using SimpleScalar. Unified caches are defined by first defining the DL2 as of unified type with all parameters of your UL2 configuration, and then "pointing" the IL2 configuration variable to DL2, as the example shows:

```
-cache:dl2 ul2:1024:64:2:1
-cache:il2 dl2
```

This works on the same way with a unified level 1 cache. Of course, the thesis software works on this way.

[1] Creation of illegal set number or so shouldn't stand here in the foreground, because the idea matters.

[2] This happened on Solaris 5.8 and 5.9 with different exception names for each.

Appendix B

Installation

All tools can be compiled and installed without problems. SimpleScalar, and its extension SimWattch, were patched so that they can be compiled under Solaris and Mac OS X. CACTI runs fine everywhere. It's recommended to turn the optimization with the GCC flag `-O3` in the Makefile and to use version 3.2 or higher of GCC because version 2.x of GCC doesn't compile when local variables are declared like in SimWattch.

The made patches consists mostly of adding statements like `#ifdef __GNU__`, `#ifdef __GNUC__`, `#ifdef __APPLE__` or as a parameter for an internal macro called `define` like `define(__APPLE__)`. Sometimes the system header file `termio.h` was demanded, but on Mac OS X you only had `termios.h` so you had to extend the case analysis.

When you've compiled the tools, make them available by adding them to your path searching list.

The complete SimpleScalar suite is compilable and installable on Solaris. With this are all SimpleScalar tools included like a modified GCC to compile your benchmarks in C. But for running the whole optimization process you only need compiled benchmarks from elsewhere and you can run the thesis software as soon as your tools, and all necessary parts, are installed and prepared. For detailed installation instruction see the README file.

If you want to install the SimpleScalar suite anyway, the tutor Christian Plessl wrote a shell script, based on some prework from the author, which downloads the necessary files, if not yet downloaded, and installes them into the directory you want.

```
## installation script for SimpleScalar at TIK
## authors: Bojan Antonovic, Christian Plessl
## version: 0.4.1

## TODO
##
```

```
## Gibc is copied manually, maybe compiling them would be better

SRC_DIR=~/simple_src            # installation for source download
BUILD_DIR=~/simple_build        # build directory
INSTALL_DIR=~/simple_install    # installation directory

DOWNLOAD_SOURCES=1      # set to 1 if download is needed
GTAR=gtar
GMAKE=gmake
```

SRC_DIR, BUILD_DIR and INSTALL_DIR have to be set to your own value. DOWNLOAD_SOURCES must be 1 because you need to download the sources once. And please define appropriate value of GTAR and GMAKE, because the last one set to make can be a bug source on certain OS.

A last word: The thesis software needs Java in version 1.4 to run. Create appropriate links to the Java compiler javac and the Java interpreter "java" if you have multiple versions of it installed.

Bibliography

[1] E. Zitzler. Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. PhD Thesis, Swiss Federal Institute of Technology (ETH) Zurich.

[2] www.tik.ee.ethz.ch/pisa

[3] www.tik.ee.ethz.ch/pisa/pisa_paper.pdf

[4] www.simplescalar.com

[5] The SimpleScalar Tool Set, Version 2.0

[6] www.ee.princeton.edu/ dbrooks/wattch-form.html

[7] David Brooks: Wattch: A Framework for Archivectural-Level Power Analysis and Optimization

[8] research.compaq.com/wrl/people/jouppi/CACTI.html

[9] research.compaq.com/wrl/people/jouppi/cacti3.ps.gz

[10] Rufer, Wyss: Processor Wizardry, semester thesis, 2002

[11] www.inf.ethz.ch/education/courses/catalog/37-003.html

[12] P. Widmayer/T.Ottman: Algorithmen und Datenstrukturen

[13] David Flanagan: Java in a nutshell.

[14] www.gnuplot.info