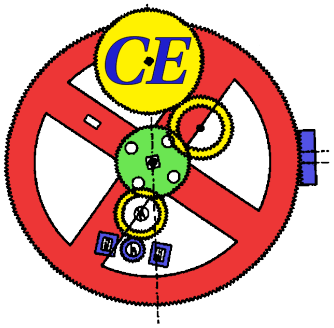


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2007

MSc THESIS

A Quantitative model for Hardware/Software Partitioning

Roel Meeuws

Abstract

Heterogeneous System Development needs Hardware/Software Partitioning performed early on in the development process. In order to do this early on predictions of hardware resource usage and delay are necessary. In this thesis a Quantitative Model is presented that can make early predictions to support the partitioning process. The model is based on Software Complexity Metrics, which capture important aspects of functions like control intensity, data intensity, code size, etc. In order to remedy the interdependence of the software metrics a Principal Component Analysis performed. The hardware characteristics were determined by automatically generating VHDL from C using the DWARV C-to-VHDL compiler. Using the results from the principal component analysis, the quantitative model was generated using linear regression. The error of the model differs per hardware characteristic. We show that for flip-flops the mean error for the predictions is 69%. In conclusion, our quantitative model can make fast and sufficiently accurate area predictions to support Hardware/Software Partitioning. In the future, the model can be extended by introducing extra software metrics, using more advanced modeling techniques, and using a larger collection of functions and algorithms.

CE-MS-2007-02

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

A Quantitative model for Hardware/Software Partitioning

A decision model for partioning candidate functions using software metrics

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Roel Meeuws
born in Rotterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A Quantitative model for Hardware/Software Partitioning

by Roel Meeuws

Abstract

Heterogeneous System Development needs Hardware/Software Partitioning performed early on in the development process. In order to do this early on predictions of hardware resource usage and delay are necessary. In this thesis a Quantitative Model is presented that can make early predictions to support the partitioning process. The model is based on Software Complexity Metrics, which capture important aspects of functions like control intensity, data intensity, code size, etc. In order to remedy the interdependence of the software metrics a Principal Component Analysis performed. The hardware characteristics were determined by automatically generating VHDL from C using the DWARV C-to-VHDL compiler. Using the results from the principal component analysis, the quantitative model was generated using linear regression. The error of the model differs per hardware characteristic. We show that for flip-flops the mean error for the predictions is 69%. In conclusion, our quantitative model can make fast and sufficiently accurate area predictions to support Hardware/Software Partitioning. In the future, the model can be extended by introducing extra software metrics, using more advanced modeling techniques, and using a larger collection of functions and algorithms.

Laboratory : Computer Engineering
Codenumber : CE-MS-2007-02

Committee Members :

Advisor:	Koen Bertels, CE, TU Delft
Chairperson:	Stamatis Vassiliadis, CE, TU Delft
Member:	Georgi Kuzmanov, CE, TU Delft
Member:	Erik Dirkx, ETRO, VU Brussel

To the Lord God Almighty.

Contents

List of Figures

List of Tables

List of Algorithms

List of Terms

AICC	Average Information Content Classification
API	Application Programming Interface
ALAP	As Late As Possible Scheduling
ANOVA	ANalysis Of VAriance
ANSI-C	American National Standards Institute standard for the C programming language
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
AST	Abstract Syntax Tree
BCS	Binary Constraint Search
CCU	Custom Computing Unit
CDFG	Control- and Data Flow Graph
CFG	Control Flow Graph
CLB	Configurable Logic Block
COCOMO	COnstructive COst MOdel
COTS	Custom Off-The-Shelf
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
DSP	Digital Signal Processing or Digital Signal Processor
DWARV	Delft Workbench Automated Reconfigurable VHDL generator
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GC	Global Criticality

GCLP	Global Criticality/Local Phase driven algorithm
GLM	Generalized Linear Model
GPP	General Purpose Processor
GPRS	General Packet Radio Service
GSM	Global System for Mobile Communications previously Groupe Spéciale Mobile
H-CDFG	Hierarchical Control- and Data Flow Graph
ILP	Instruction-Level Parallelism
KDSI	1000 Delivered Source Instructions
LOC	Lines Of Code
LUT	Look-Up Table
MIBS	Mapping and Implementation Bin Selection
MSE	Mean Squared Error
PC	Principal Component
PCA	Principal Component Analysis
PRESS	Predicted Residual Sum of Squares
Q-Q Plot	Quantile-Quantile Plot
R^2	Coefficient of Determination
RC	Reconfigurable Computing
$\rho\mu$**-code**	Reconfigurable Micro-code
RMSE%	Rooted Mean Squared Error Percentage of the mean of the independent variable
RTL	Register Transfer Level
STG	State Transition Graph
TEW	Total Edge Weight
VHDL	VHSIC Hardware Description Language (VHSIC stands for Very-High-Speed Integrated Circuit)
UMTS	Universal Mobile Telecommunications System

Acknowledgements

I am grateful for all the help and support I received during my thesis project. Therefore, my thanks go out to Koen Bertels, for all his advice, support, and encouragement, to Stamatis Vassiliadis for his inspiration - may he rest in peace -, to Yana Yankova for her endless work on DWARV that was essential for my work, to Arcilio for his close cooperation and friendship, to Eric Cator for his important advice on Modeling and Statistics, and Gerard Aalbers for proof reading my work and his friendship.

Also, I would like to take the opportunity to thank the people that have played an important role in my life. First of all my parents, thank you for your nurturing, support, love, and confidence over the years! Also, I would like to thank all my friends, who have been there for me and thought of me in their prayers.

And most importantly all my gratitude go to my Lord God, who has guided me every step along the way.

Roel Meeuws
Delft, The Netherlands
May 1, 2007

1 Introduction

For many years, computers have been based on the Von-Neumann Machine (or Stored-program machine), which is a machine divided into a processing unit, a combined data and program memory for data, and a sequential flow of data and control elements between the memory and the processing unit [99]. The idea of a program of instructions that are executed sequentially made the implementation of algorithms much simpler, hence the rapid advancement of software development in the following decennia became possible.

However, as Backus [9] pointed out, the concept showed an inherent bottleneck, which he called the "Von-Neumann bottleneck". Because the processing unit and the memory in a Von-Neumann machine are separate, instructions and data have to be moved continually. Furthermore, the sequential nature of this process limits the speed one can achieve by exploiting more parallelism. Still, the Von-Neumann computer has been successful due in no small part to the many tools supporting the paradigm at each level. Moreover, the miniaturization of electronics have provided regular speed improvements (Moore's Law), diminishing the need for a non-Von-Neumann architecture.

Despite the dominance of Von-Neumann machines, other architectures have been used in specific areas. Application Specific Integrated Circuits (ASICs), for example are able to use the parallelism inherent to the problem at hand and combine processing and storage into their data-path. In contrast to more general applications, application specific systems did not need the programmability and flexibility of the Stored-Program machine. Special languages, tools, and design methodologies have been developed to make the implementation of ASICs possible.

In recent years, the continuing applicability of Moore's Law has come into question. For one, wire delays become an increasing problem at higher speeds, and second, the manufacture of transistors smaller than a few atoms seems unlikely. Furthermore, a growing demand for mobile technology and other systems with limited power supplies have made the use of fast Von-Neumann processors in such systems difficult if not impossible. To cope with this problem, designers increasingly use ASICs to speed up expensive algorithms like media encoding and signal processing. Such systems, where both programmable and application specific systems are combined, are called heterogeneous systems.

The problem that remains, however, is the inflexibility of such custom hardware, i.e. every different task needs a different circuit. This results in a combinatorial explosion of ASICs, driving up the cost considerably. In order to remedy this problem the research community introduced the concept of Reconfigurable Computing (RC). Reconfigurable Computing complements programmable software components with programmable hardware components [67], like Field Programmable Gate Arrays (FPGAs). Hence, Reconfigurable Computing advances the idea of heterogeneous systems by intro-

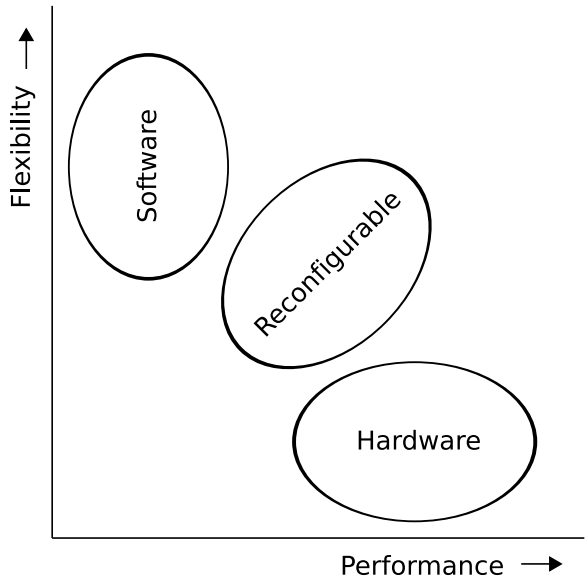


Figure 1.1: The design space between performance and flexibility of programming. Reconfigurable Computing positions itself between Hardware and Software.

ducing programmability to the hardware components. These programmable hardware components make it possible to dynamically load different ASIC designs or configurations, making flexible non-Von-Neumann machines a possibility. This way Reconfigurable Computing positions itself in the design gap between hardware and software, as illustrated in Figure 1.1. To clarify the concept of Reconfigurable Computing , let us look at [28], where three levels of programmability are identified for both control-flow models (software) and data-flow models (hardware) (Table 1.1). The programmability in RC comprises all those instances of programmability.

Control-flow	**Data-flow**
Different programs can be executed	Different circuits can be executed
Executing programs can be modified	Executing circuits can be modified
Dynamic behavior through choice in control flows	Dynamic behavior through choice in data flows

Table 1.1: Three different levels of programmability in control-flow and data-flow systems, as presented in [28]

1.1 A case study: Software Radio

As an illustration of how Reconfigurable Computing can help alleviate processing requirements, while remaining flexible, we now look into the Software Radio[20, 102, 27, 87, 88]. In mobile communications many different frequency bands are used for different

applications, like Global System for Mobile Communications (GSM) for voice, General Packet Radio Service (GPRS) for Internet, and Universal Mobile Telecommunications System (UMTS) for video. To make things more complicated the exact frequencies differ per region, for example GSM at 1800MHz in Europe and Asia and GSM at 1900MHz in North America. Because of the different networks, frequencies, and bandwidths, a programmable radio that can service different networks on demand would be very helpful.

However, the computing power required for a software radio on a conventional processor is quite high. As an illustration, look at the example in [102], that mentions that processing a 500MHz carrier frequency using a 1 GHz sampling rate (as dictated by the Nyquist theorem) on a 32-bit 4-issue 4GHz system, leaves 32 operations per sample, which is not enough to filter and (de)modulate the signal, apply error correction, and so forth. [102], argues for the use of a heterogeneous multiprocessor architecture, i.e. an architecture that comprises different ASIC and General Purpose Processor s (GPPs), to tackle this problem. Nevertheless, such an approach may be expensive. [27, 87] suggests RC (especially FPGA technology) as a means to make radio signal processing possible, while remaining flexible enough to service different networks at different times. In [87] we even find an example design: the Layered Radio Architecture.

1.2 Reconfigurable Computing Requirements

Although the advantages of Reconfigurable Computing are clear, it has not pervaded industry as traditional computing has. In an attempt to explain this, [67] argues that while the Von-Neumann Machine is supported by an extensive and mature base of tools, Application Programming Interface s (APIs), and design methodologies, no such extensive support is available for the Reconfigurable Computing paradigm. In other words, for Reconfigurable Computing to be commercially viable, it should have a comparable support base. In recent years some tools have been developed to attack this problem.

The problem does not end there, however. Because Reconfigurable Computing moves away from the Von-Neumann model, the extensive base of support should be adapted accordingly. In [6], for example, current hybrid programming models are pointed out to be immature, because FPGAs and Central Processing Unit s (CPUs) are treated completely separate. In order to make the design of Reconfigurable Computing systems feasible the paper proposes a more transparent model, i.e. the multithreading model. This model provides a way to describe concurrency without specifying where the thread will be implemented. A separate partitioner can then partition the threads over the hybrid processing elements. In [100] this model is elaborated in more detail. The paper describes how to implement software and hardware threads by using a common abstraction layer in the operating system, providing a common interface between hardware and software threads. Another possible programming model for RC is presented in [90], where a functional programming model, called V, is introduced. That model uses implicit parallelism and aims to be similar to both traditional embedded (compositional) functional models, as well as more component based models used in hardware design. Computational models need to be redefined with respect to Reconfigurable Computing as well. In [23] a redefinition of the term algorithm, as used in computability theory, is presented tailored to Reconfigurable Computing .

Domain	Kernels	Bit-Based	Streaming	Account-Keeping[a]	Control-Intensive
Compression	2	x		x	x
Cryptography	56	x	x		x[b]
DSP	5	x	x		x[b]
ECC	6	x	x		x
Mathematics	19				
Multimedia	32	x[b]	x		x
General	15			x[b]	x
Total	135				

[a]Non-constant space complexity.
[b]Only some instances in that domain express this characteristic.

Table 1.2: The number of functions in each domain together with the main algorithmic characteristic present in the software implementations.

1.3 Problem definition

We have already established the need for extensive tool support when considering the acceptance of RC in industry. Tool support and integration for RC is the main focus of the Delft Workbench project (see Chapter 2.3.1). Its research covers the entire design process from code profiling to compilation.

Within the Delft Workbench project research is being done into hardware estimation and hardware/software partitioning. One possible strategy that has been suggested in order to tackle these two problems was to use a model based on software metrics. However, the question remains, *can hardware characteristics be predicted using a model based on software metrics?* In this thesis, I try to answer this question by building a preliminary decision model based on software metrics for the Delft Workbench.

For this purpose, I have collected a dataset of 135 functions in the C-language from a broad spectrum of application domains, as can be observed in Table 1.2. Using such a broad set of functions ensures that any results would be applicable to real world applications. As a consequence of the broad scope of the dataset, the candidate functions have varying types of algorithms and memory access patterns as can be observed in Table 1.2. This implies the need for different kinds of software metrics that can characterize the different aspects of these application domains. In my thesis I will use these C-kernels to build a statistical model based on linear regression. This model will predict several hardware characterics. Although these predictions are made in the very first design stages where many aspects of the final design are not yet concrete and thus a relatively large error margin can be expected, these predictions can be invaluable for guiding the hardware/software partitioning process.

My thesis is organized as follows. In Chapter 2 I investigate the problem further and review previous research and literature. In Chapter 3 I investigate software metrics in general as well as select a set of metrics for use in the decision model. Then, in Chapter 4, I present the relevant theory of building a statistical model using software metrics. Next, I present the methodology and experimental results in Chapter 5. The

results comprise the model specification and validation. Finally, in Chapter 6 I briefly discuss the results presented in this work and indicate possible future directions for research and implementation.

Literature Review and Related Research

2

The two main functions of the decision model envisioned by Delft Workbench are a) deciding how to partition a software application over hardware and software components and b) estimating the resource requirements and performance of the resulting system. These problems have been researched extensively and in this chapter I review previous solutions to hardware/software partitioning (Section 2.1) and estimation (Section 2.2).

Apart from discussing these two functionalities of the model, this chapter also describes the direct context of the mode, i.e. the MOLEN and Delft Workbench projects Section 2.3.1. Other projects in the field of Reconfigurable Computing are also briefly discussed.

2.1 Hardware/Software Partitioning

Later on, we discuss several hardware estimation measures that can help discover and select candidates for hardware implementation. While estimation is an important part of Hardware/Software partitioning, the actual process of partitioning is at least as important and will be discussed first. In the past partitioning was often performed by hand, but in more recent years, automatic partitioners have begun to appear. Many algorithms have been proposed over the years focusing on different aspects of a design. All partitioning algorithms try to optimize some measure under certain resource constraints, e.g. minimizing power consumption under speed and area constraints. A model for candidate selection for hardware implementation is in fact a partitioning model and therefore further investigation into partitioning algorithms can be useful in determining such a model. In this section, we discuss several partitioning strategies and how they connect with estimation and profiling.

2.1.1 Partitioning Algorithms

The Hardware/Software partitioning problem shows many similarities with graph partitioning. And indeed there are papers on hardware/software partitioning that apply algorithms from graph theory. Other algorithms come from evolutionary programming or statistical analysis. In the following, we discuss some of these algorithms.

2.1.1.1 Greedy

An early approach to the partitioning problem, presented in [42], uses a greedy algorithm. Greedy algorithms use locally optimal decisions to approximate the globally optimal partitioning. In case of [42], the algorithm starts with an all hardware partitioning and moves nodes that yield the largest decrease in communication first, until the constraints

can no longer be satisfied. While this approach is relatively fast and may give acceptable solutions, it leaves quite some room for further improvement. This scheme of sorting all nodes according to some measure and then moving the topmost portion that still fulfills the constraints to hardware is used in many other partitioning schemes in literature. In [91], for example, the most frequently executed loops are migrated to hardware. The CRUSADE algorithm [29] allocates task clusters with the highest priority in hardware first.

2.1.1.2 Simulated Annealing

In order to remedy the flaw of greedy algorithms to get stuck in local optima, several hill-climbing algorithms have been proposed for graph partitioning. These algorithms can temporarily accept less optimal solutions, in order to find a (more) global optimum. One such hill-climbing scheme, Simulated Annealing [58], that is often used in hardware

Paper	Dynamic/ Static	Strategy	Criteria	Model/ Data Structure	Granularity of Partitioning	Time Complexity
[58]	Static	Simulated Annealing	n/a	n/a	n/a	n/a
[42]	Static	Greedy	Minimal area, data-rate constraints	System Graph Model (like H-CDFG)	operations	linear
[41]	Static	Greedy (see [42])	Minimal area, data-rate constraints	Hierarchical Sequence Graph	operations	n/a
[77]	Static	Simulated Annealing	Minimal communication cost	Petri-nets, (annotated) CDFG	operations	$O(tn)$ t=temperature steps
[34]	Static	Simulated Annealing	Hardware suitability (compare local phase [54])	(extended) $C^{\times}$ syntax graph	basic blocks	n/a
[54]	Static	GCLP	GC objective function (e.g. Area combined with speed)	n/a	Tasks (instruction level subgraphs)	$O(ne)$, e=edges
[96]	Static	Binary Constraint Search	Constraints of encapsulated partitioning algorithm	n/a	n/a	$O(part(S))$ $part(S)$ = encaps. part. alg.
[50]	Static	Dynamic Programming	Temporal size of loops / leaf functions	n/a	loops, leaf functions	n/a
[53]	Static	GCLP (MIBS)	See [54]	CDFG	Tasks	$O(n^3 + n^2B)$, B=bins
[82]	Static	Evolutionary (Genetic)	Minimal area, timing and concurrency constraints	CDFG	functional elements	$O(gp)$, g=generations, p=population
[30]	Static	Clustering	Minimal cost, minimal power, timing and power constraints	Task Graph	task clusters	n/a
[29]	Dynamic	Greedy, Clustering	Minimize area, timing constraints	Task Graph	task clusters	n/a
[65]	Dynamic	Clustering	Area constraints	CDFG	loop clusters	linear
[89]	Static	Evolutionary (Genetic)	maximize fitness (minimize area and interconnect)	DFG	fine:operations coarse:DFGs	n/a
[84]	Dynamic	Evolutionary	Maximum rank (Pareto ranking in power and price)	Task Graph	Tasks	n/a
[91]	Static	Greedy	Temporal size of loops / leaf functions	n/a	loops	n/a
[14]	Static	Dynamic Programming	Minimum latency, resource constraints	DFG	Tasks	polynomial
[12]	Static	Simulated Annealing, Kernighan-Lin	Minimize latency, area constraints	Call graph	functions	n/a

Table 2.1: Inventarization of several papers on hardware software partitioning with corresponding partitioning schemes, criteria, and data structures

```
1  function GreedyPart(system: graph): graph
2    // Initialize partition and cost
3    partition := InitPartition(system);
4    cost := Cost(partition);
5
6    // partition each node
7    for each node in system do
8      partition' := partition;
9      // move node in partition' from/to HW
10     swap(node, partition');
11     if feasible(partition') then
12       if Cost(partition') < cost then
13         partition = partition';
14         cost = Cost(partition');
15       end if
16     end if
17   end for
18   return partition;
19 end function
```

Algorithm 2.1: Pseudo code of an example greedy algorithm for solving the partitioning problem.

software partitioning [34, 77], is based on techniques in statistical mechanics. The idea is to introduce a "temperature" to the optimization process, which determines how often counter-productive moves from one part of the graph to another are allowed. This "temperature" is lowered during the partitioning process, until the temperature is zero at which point a local optimum can be found. This partition is not only locally optimal, but also the optimal partition of several locally optimal partitions and is therefore closer to the global optimum. While simulated annealing generally finds better solutions than a simple greedy approach, it is also slower because the temperature has to drop to zero over several iterations first.

2.1.1.3 Kernighan-Lin/Fiduccia-Mattheyses

Classical graph bi-partitioning algorithms like Kernighan-Lin and Fiduccia-Mattheyses are also used in Hardware/Software partitioning. They consist of a sequence of passes which yield trial partitions, which are used as input to subsequent passes. A trial partition is obtained by repetitively moving nodes between partitions and locking them during the remainder of the pass. Each move is performed according to some cost metric. No paper discussed here presents hardware/software partitioning or synthesis using one of these algorithms. However, [12] shows that Simulated Annealing can deliver better quality partitions in less time.

```
function SAPart(system: graph): graph
  Temp := StartTemperature;
  Conf := InitConfiguration(system);
  while (cost changes) OR
        (Temp > FinalTemp)
  do
    for a number of times do
      generate a new configuration Conf';
        if accept(Cost(Conf'), Cost(Conf),
                  Temp)
        then Conf := Conf';
    end for
    Temp := Temp * ReductionFactor;
  end while
end function

function accept(NewCost, OldCost, Temp)
  CostChange := NewCost - OldCost;
  if CostChange < 0
  then // accept based on cost improvement
    accept := TRUE;
  else // else accept randomly
    Y := exp(-CostChange/Temp);
    R := random(0, 1);
    accept := (R < Y)
end function
```

Algorithm 2.2: Pseudo code of a generic simulated annealing algorithm for solving the partitioning problem as found in [77].

2.1.1.4 Evolutionary or Genetic Algorithms

Another approach to hardware/software partitioning is based on principles from evolutionary biology. Evolutionary programming can be traced back to the fifties [8] and since then several different types of evolutionary strategies have been developed, each with their own niche. For combinatorial problems like graph partitioning, the so-called *genetic algorithm* is most suitable. In hardware/software partitioning genetic algorithms have been used in [82, 89, 84] among others.

In a genetic algorithm possible solutions to a problem are treated as genomes that compete in an evolutionary setting. Intermediate solutions in one iteration (*generation*) of the algorithm compete and the best solutions go to the next generation. When a new generation starts changes (*mutations*) are introduced in the solutions and solutions can exchange parts of their "genome" (*crossover*). In [82], for example, partitioning is modeled as a constraint satisfaction problem, which in turn is mapped to a genetic algorithm. The *fitness* of solutions is determined by the value of the constrained measures. This way fitter solutions are expected to better fulfill the constraints. While genetic algorithms can give good solutions to problems with many constraints and multidimensional cost

```
1  function FMPart(system: graph): graph
2    partition = InitPartition(system);
3    repeat
4      for each node in partition
5        unlock(node);
6      gain = 0;
7      while unlocked nodes remain
8        node = MaxGainUnlockedNode(partition);
9        swap(node, partition);
10       gain = node.gain;
11       lock(node);
12       for each node in partition
13         update node.gain;
14     end while
15   until gain <= 0;
16   return partition;
17 end function
```

Algorithm 2.3: Pseudo code of a Fiduccia-Mattheyses algorithm for solving the partitioning problem.

functions, the strategy provides no test for optimality, requires carefully chosen parameters like mutation rate, crossover rate, etc., and can be very time consuming, i.e. many generations may be needed before a stable state is achieved.

2.1.1.5 Global Criticality/Local Phase driven algorithm (GCLP)

An extension to simple serial greedy partitioning is the Global Criticality/Local Phase driven algorithm (GCLP) [54]. Serial greedy algorithms can only optimize for one cost metric when deciding where to partition a node, GCLP on the other hand facilitates using multiple cost metrics. Which cost metric to use is determined by comparing Global Criticality (GC) , a measure of temporal criticality for each node, to a threshold value. This threshold is augmented by a *local phase* delta. Local phase is a classification measure that indicates the heterogeneity of a node. There are 3 local phase classes: *Extremities*, *Repellers*, and normal nodes. Extremities for an implementation are nodes that are inefficient for that implementation. Repellers of a certain implementation are nodes that are more efficiently implemented in another partition than other nodes with the same costs for the current partition. Normal nodes are nodes that are neither extremities nor repellers. The main advantage of GCLP is the improved accuracy over simple greedy algorithms while maintaining the speed of the partitioning process.

In [53] a more elaborate scheme based on GCLP is presented. This paper combines GCLP and implementation-bin selection into an iterative algorithm called Mapping and Implementation Bin Selection (MIBS). At the beginning of a MIBS iteration some nodes have been mapped (fixed nodes) and some nodes have not (*free nodes*). GCLP is applied to the free nodes accounting for the fixed nodes as well. From these temporarily partitioned free nodes one node (*tagged node*) is selected for implementation bin selection.

```
(* The chromosomes in the following algorithms are a concatenation of so-called
partial-codes
(x_i), which denote the
implementation(u_il,0 ≤ l ≤ m_i) of a
function (i). The chromosome represents a
solution and it's genes partial-codes. Partial-codes relate to partial-codes as
follows: l = x_i mod m_i*)

const threshold = minimal acceptable gain;
function GeneticPart(system: graph): graph
  var population: chromosome[1..size];
  fill population with random chromosomes;
  generation = 0;
  previous = best = MAXINT;
  repeat
    for each chromosome in population
      calculate its fitness
    newpopulation = [];
    previous = best;
    while newpopulation.size < size do
      parents = chooseParents(population);
      child = crossOver(parents);
      child = mutate(child);
      newpopulation.add(child);
      if child.fitness > best.fitness
      then
        best = child.fitness
        bestchild = child
      end if
    end while
    population = newpopulation;
    generation = generation - 1;
  until (generation = maxgens) OR
        (ABS(best-previous) < threshold)
  return bestchild;
end function

function chooseParents(population)
(*This function is based on the roulette-wheel method, i.e. the fitness of each
chromosome defines the chance it's chosen.*)
end function

const n;
const crossChance;
function crossOver(parents)
  if flipCoin(crossChance)
  then
    choose n crossover sites randomly;
    for each crossover site s
      temp := parent1.partialcode[s];
      parent1.partialcode[s] :=
        parent2.partialcode[s];
      parent2.partialcode[s] :=
        temp;
    end for
end function

const mutateChance = 0.008;
function mutate(chromosome)
  for each bit in chromosome
    if(flipCoin(mutateChance)) then
      invert(bit);
  end for
end function
```

Algorithm 2.4: Pseudo code of the Genetic Algorithm for solving the partitioning problem as found in [82].

```
1  var U: unscheduled nodes;
2  var S: scheduled nodes;
3  (* ready nodes are nodes whose
4     predecessors have all been
5     scheduled *)
6  var R: ready nodes;
7  function GCLPPart(system: graph): graph
8    computeExtremities(system);
9    computeRepellers(system);
10   while not empty(U)
11     ComputeGlobalCriticality(system);
12     Determine R;
13     Compute effective exec. time t_exec(i)
14 (* if i ∈ U:t_exec(i) = GC.t_(HW,i) + (1 − GC)t_(SW,i)
15    else if i ∈ S:t_exec(i) = t_(part,i)*)
16
17    for each node in R
18      compute LongestPath(root, node);
19    pick node from R with Max(longestPath);
20    if isExtremity(node) then
21      delta = node.bias_HWorSW;
22    else if isRepeller(node) then
23      delta = node.repelValue;
24    else
25      delta = 0
26    end if
27    if node.gc >= .5 + delta
28      objective=speedobjective
29    else
30      objective=resourceobjective
31    end if
32    partition node so Min(objective)
33    U = U − node;
34    S = S + node;
35   end while
36   return system;
37 end function
```

Algorithm 2.5: Pseudo code of a GCLP algorithm for solving the partitioning problem, as found in [54]. Some functions have not been listed here for space reasons.

When the implementation is determined the tagged node becomes a fixed node, signaling a new iteration. The implementation-bin selection traverses all implementations for the tagged node from the fastest(*L-bin*) to the slowest (*H-bin*) and for each implementation-bin determines the fraction of free nodes that need to move to their L-bin in order to meet timing constraints. When the difference in the number of free nodes in their L-bin between two successive bins is maximal the second-last visited implementation is selected. The MIBS algorithm presents better results than GCLP, but is also a lot slower. On the

```
1  function DPPart(system: graph): graph
2    if hasMultipleOutputs(system)
3    then
4      add dummy output node;
5      connect each ouput to dummy;
6    end if
7    nodes = reverseTopologicalSort(system);
8    while not empty(nodes)
9      node = pop(nodes);
10     merge(node, system);
11   end while
12   node = output node;
13   bestSolution = minimum latency solution for node;
14   for each node in bestSolution
15     put node in node.partition;
16 end function
17
18 function merge(node, DAG)
19   predecessors = fan-in nodes of node
20   for part in [HW,SW] do
21     node.partition = part;
22     for each pred in predecessors do
23 out:  for each solution in pred.solutions do
24         Delay = solution.delay + delay_{pred->node};
25         for each other predecessor do
26           (* choose solution that has smaller
27              delay than Delay and minimum
28              area.*)
29           if no solution found then
30             continue out;
31           other[i] = chosen solution;
32         end for
33         if sum(other[*].area) > maxarea then
34           continue out;
35         node.solutions.add(combine(other[*], node));
36       end for
37     end for
38   end for
39 end function
```

Algorithm 2.6: Pseudo code of a Dynamic Programming algorithm for solving the partitioning problem, as found in [14].

other hand the paper claims that the quality of the results come close to integer linear programming solutions, while being much faster.

2.1.1.6 Dynamic Programming

When a problem has recursive characteristics and overlapping subproblems, dynamic programming can help find a solution. Several papers use dynamic programming in their partitioning [50, 14] strategies. For example, [14] describes how possible mappings of a node in a Directed Acyclic Graph (DAG) are determined in a bottom-up fashion. Starting at the root node of the DAG every node builds a solution list with delay and resource information using its fan-in nodes. Infeasible solutions are pruned from the list and if all fanout nodes of a node are processed the list can be pruned entirely. When multiple solutions exist in fan-in nodes, only the one with the minimal resources is selected. When all nodes have their possible solutions assigned, every node is assigned to hardware or software according to the fastest solution listed.

When dynamic programming is applied correctly the algorithm will find exact solutions. Dynamic programming can be time-consuming for arbitrary graphs (NP-hard), but if DAGs are used the paper shows that this specific application of dynamic programming has a time complexity of $O(n)$. Dynamic Programming does have a larger space complexity than a brute force approach.

2.1.1.7 Binary Constraint Search (BCS)

One problem in hardware/software partitioning are the often conflicting goals of minimizing one cost measure while satisfying others. For example, when minimizing area, performance constraints may be violated. The authors of [96] sought to solve this problem by splitting the problem in satisfying constraints and minimizing some cost measure by encapsulating a partitioning algorithm in a binary constraint search algorithm. In [96] the encapsulated algorithm optimizes for performance under a hardware size constraint that is determined by the encapsulating algorithm. The BCS algorithm then uses a cost metric defined as the total number of constraint violations to search for the solution with zero cost that has the lowest hardware size constraint. This encapsulation approach can have various results depending on the encapsulated partitioning algorithm, but the complexity of the resulting partitioning algorithm is $O(C_{part} \log n)$, where C_{part} is the complexity of the encapsulated algorithm.

2.1.1.8 Clustering Algorithms

When dealing with large problem sizes partitioning algorithms can become slow and partitioners may require to switch to less efficient partitioning strategies. Another solution is to reduce the problem size in some way. The COSYN system presented in [30] accomplishes this by clustering nodes (in this case tasks) together. This particular clustering approach is also used by the CRUSADE algorithm [29]. In order to decrease the schedule length [30] uses a clustering algorithm based on decreasing the longest path in a task graph. By clustering tasks on the longest path together, communication between those tasks becomes negligible. Now another path may be critical and clustering starts again until no improvement of the critical path can be made. The acquired clustered task graph can now be partitioned more easily. Of course the obvious speed improvements this scheme has over normal partitioning comes with reduced accuracy. Furthermore,

```
function BCSPart(system: graph,
                 cons: constraints,
                 PartAlg: function)
                : graph
begin
  low = 0;
  high = AllHardwareSize;
  while low < high do
    mid = (low + high +1)/2;

(H',S')=PartAlg(H,
S, Cons, mid, Cost());
    if
Cost(H',S',Cons,
mid)=0 then
      high = mid - 1;

(H_zero,S_zero) =(H',S');
    else
      low = mid;
    end if
  end while
  return
(H_zero,S_zero);
end function

function PartAlg(HW: set, SW: set,
                 Cons: constraints,
                 mid: area constraint,
                 Cost: function)
begin
(* This function can have different
   implementations, like GCLP, Greedy,
   etc. *)
end function
```

Algorithm 2.7: Pseudo code of a Binary Constraint Search algorithm for solving the partitioning problem, as found in [96].

because the granularity has coarsened the final partitioning might contain some unused area where a single task might have been partitioned.

A different clustering method, called hierarchical loop clustering, is employed by [65], where loops are clustered together. Clustering in this method is based on the loop-procedure hierarchy graph, which contains all procedure calls and loops as nodes, while edges indicate a caller-callee or nesting relation between nodes. This graph is then traversed from the root down and all nodes that have a common predecessor on the current level are clustered together. When a cluster is within the cluster size limit it becomes fixed, when clusters are too large the next level is inspected. This process

```
1  const maxsize;
2  var loophierarchy: graph;
3  function Cluster(root: node): clusters
4    clusters = empty;
5    if hasChildren(root) then
6      for each child in children(root) do
7        cluster = subtree(child);
8        if cluster.size <= maxsize
9          clusters.add(cluster);
10       else
11         clusters.add(Cluster(child));
12     end for
13   else
14     return {root};
15   end  if
16   return clusters;
17 end function
```

Algorithm 2.8: Pseudo code of a Clustering algorithm for solving part of the partitioning problem, as can be found in [65].

continues until all clusters are fixed. The advantage of this clustering approach is the clusters are disjunct in time, which makes it possible to find a dynamic schedule for e.g. an FPGA.

2.1.2 Partitioning and Estimation

Generally partitioning algorithms are targeted at optimizing certain criteria, while satisfying constraints on others. Different papers have focused on different optimization criteria like area, speed, and power, but also communication, memory, and loop frequency. It is evident that estimation strategies as discussed in Section 2.2 are essential to hardware software partitioning. Some papers on estimation even presented schemes tailored for partitioning schemes [89, 30, 66, 51, 95]. However, when combining estimation- and partitioning schemes, one must consider the different levels of speed and precision. For example, when a very precise partitioning scheme like dynamic programming is applied, it makes less sense to incorporate estimators with large error margins, because it is useless to exactly partition a system using only partially accurate criteria. On the other hand a greedy algorithm might benefit from more exact estimates by letting it find a better local optimum.

In fact, when considering a certain granularity of partitioning it is good to use estimates of a corresponding granularity. Indeed, if we look at the granularity of partitioning in Table 2.1, it corresponds to the granularity levels mentioned in Section 2.2.4.7 with the addition of operations, for which estimation is trivial. The table does not show a particular trend in the granularity of partitioning.

2.1.3 Dynamic versus Static Solutions

Most papers on hardware/software partitioning have focused on finding static partitions between hardware and software. With the advent of RC, however, the need for dynamic solutions to partitioning arose. Dynamic partitioning solutions define which part of a system is executed in which partition and at which time. This allows a system to exploit the dynamic reconfigurability of e.g. an FPGA, possibly reducing power, area, or other requirements.

Examples of these dynamic partitioning techniques can be found in [65, 29, 86, 84] among others. In [65] a clustering algorithm is used where each cluster is guaranteed not to overlap other clusters in time. This results in a dynamic partitioning where each cluster can reside on the same FPGA at different intervals in the program's run-time. However, the paper does not mention the problem of configuration time required to load clusters onto the FPGA.

The CRUSADE system [29] also produces dynamic solutions. It does this by finding all pairs of non-overlapping task graphs. After allocation every pair of non-overlapping task graphs is merged when the resulting schedule still meets constraints. In order to account for boot time, i.e. reconfiguration time, a reconfiguration option array is introduced that contains various options for programming the reconfigurable logic and the corresponding boot time and cost. The cheapest option that satisfies the timing constraints is then chosen.

The Symphany Tool [86] utilizes a lazy scheduling algorithm to create dynamic schedules for implementation on reconfigurable logic. In order to do this, dummy dependency edges are added to the Hierarchical Control- and Data Flow Graph (H-CDFG) between vertices contending for the FPGA. Vertices are then scheduled to be loaded to the FPGA as soon as it is available, as long as the As Late As Possible (ALAP) schedule of their successor is not violated. When multiple vertices contend, the ALAP time of their successors are used as priority.

A more elaborate approach is presented in [84], where a dynamic scheduling scheme is used to dynamically partition tasks over FPGAs. The scheme starts with assigning priorities to all tasks based on their ALAP schedule, execution time, and reconfiguration delay. When the FPGA is already (partially) configured with the same configuration as the current task, the reconfiguration delay will be lower or zero. The task with the highest priority is then selected and allocated on the FPGA at a certain location and a certain time. The authors of [84] describe the factors that influence the allocation policy:

1. Reconfiguration Prefetching
   Large FPGA configurations can have large reconfiguration delays. In order to hide the impact of these delays configurations of such tasks can be loaded in advance.

2. Configuration Pattern Re-utilization
   Different tasks may share parts of their configuration patterns and when they are loaded onto the FPGA, those parts will be reused, speeding up reconfiguration.

3. Candidate Eviction
   When a task does not fit on an FPGA, other tasks need to be removed from the

FPGA. When tasks are removed they might be reloaded in the future, yielding extra reconfiguration delay. In order to minimize this an eviction cost measure is defined based on the total of all recurrent frequencies of all evicted tasks. When this measure is minimal, the configured task will be able to remain on the FPGA longest.

4. Fitting policy
   When assigning locations on the FPGA to tasks the allocation policy tries to avoid fragmentation.

5. Slack time utilization
   When parts of a configuration already exist on the FPGA, assignment to those locations might lower eviction cost, possibly delaying the start time of the task. The slack available can be used to determine whether this delay is tolerable. The maximum slack allowed for the current task is the total slack divided by the number of levels of tasks in the sub-graph rooted at the current task.

When the selected task is allocated, the priorities of the remaining tasks are updated and the allocation repeats, until all tasks are allocated. The worst-case complexity of this algorithm is $O(n^2 \log n)$ but on average it behaves like an $O(n \log n)$ algorithm.

2.1.4 Synthesizability and Partitioning

Many papers in hardware/software partitioning either preselect the tasks that are partitioned or assume the problem is implementable on an FPGA. In any case it is important to factor out any system part that is not synthesizable. For example, in [7], the PRISM-I system, aimed at automatically translating C-code to a hardware and a software image, prohibits a total input or output bit-width of more than 32 bits, global variables, floating point values, etc. A partitioner is often coupled to or integrated in a synthesis tool and therefore should take into account language restrictions for synthesis.

2.2 High Level Estimation, Metrics, and Profiling

Estimation of different cost parameters has always been an important activity in high level system design. Estimates of e.g. speed, area, or power inform a designer about whether a design will meet requirements, stay within budget, and so forth, thus driving further design choices. Not only cost parameters can be important measures to a designer, others, like loop frequency or data reference locality, might help direct the design process as well. Many tools and algorithms have been developed over the years that help determine metrics and thus make the job of the designer easier. If we go one step further and model the process of selecting candidates for hardware implementation, we need to look at the measures and their meaning. In this section, we discuss different metrics and techniques to estimate them. First, I briefly present the different papers I reviewed. Then, I review different aspects of the estimation schemes by finding ways to classify these metrics. Furthermore, we briefly go into synthesis and optimization and how we may measure its effects.

Paper	Dynamic/ static	Level of design	Application Domain	Data structure	Node model	Granularity	Strategy	Complexity	Error
[60]	Static	Behavioral	Multimedia	CDFG	multiple (non-)linear models	Entire Graph	Hierarchical	linear	2.5%-10%
[95]	Static	Behavioral	Communication	Custom	n/a	Functions	Incremental	$O(1)$ per iteration	about 7%
[31]	Static	Behavioral	Multimedia	CDFG	Bit-width based	Entire Graph	Neural	nonlinear	10% large error with badly trained networks
[71]	Static	System/ Behavioral	Multimedia, Mathematical	VHDL Abstract Syntax Tree	Bit-width based	Entire Application	Scheduling	nonlinear	16%
[81]	Static	Behavioral	Multimedia, General Purpose	PACT HDL Abstract Syntax Tree	Simple	Entire Application	scheduling, allocation	nonlinear	average 25% (maximum of 241%)
[73]	Static	RTL and lower	n/a	Boolean expressions	n/a	n/a	Complexity	linear	within 56%
[51]	Static	Behavioral	Multimedia, Compression, Mathematical, General Purpose, Cryptography	CDFG	unknown	Entire Graph	Hierarchical	linear	60–100%
[85]	Static	Behavioral	Multimedia	DFG	Simple	Entire Graph	Scheduling	$O(nc^2)$	n/a
[66]	Static	Behavioral	n/a	(annotated) CDFG	Simple	Entire Graph	scheduling	$O(speedest.) + O(n^2)$	n/a
[57]	Static	Behavioral	Cryptography	n/a	Simple	n/a	Hierarchical	n/a	n/a
[15]	Static	System/ Behavioral	Multimedia	H-CDFG	Simple	Entire Graph	Hierarchical	$O(n \log n)$	n/a
[89]	Static	Behavioral	Multimedia, General Purpose	(C)DFG	Simple	Node clusters	Hierarchical	linear	n/a
[16]	Dynamic	System	Communication	(annotated) DFG	Bit-width based	Entire Graph	Scheduling	$O(simulation)+ O(n^2)$	n/a

Table 2.2: Different classifications of area estimation found in several papers and indications of estimation error in those papers. For an explanation of the classifications, please refer to Section 2.2.4.

2.2.1 Area, Speed, and Power

Most work in estimation has focused on area, speed-up, and power, probably because they directly correspond with the cost and obvious requirements of designs. We will discuss them here first, before we go on to less obvious areas of estimation.

2.2.1.1 Area

Area estimation has been tackled in various ways by different groups. In [85] a lower bound on area is estimated under certain performance constraints. Specifically, they estimated the number of modules of each type and the area needed for interconnect. The paper explains how lower bound estimates can be determined by scheduling a DFG and accounting for the minimum number of modules and buses required.

Because area estimation tries to predict the results of synthesis tools, [81] tries to find estimates by mimicking high level synthesis. Techniques like force directed scheduling, resource allocation, operation assignment, and interconnection binding, all come from high level synthesis. The algorithm uses a simplified model of an FPGA

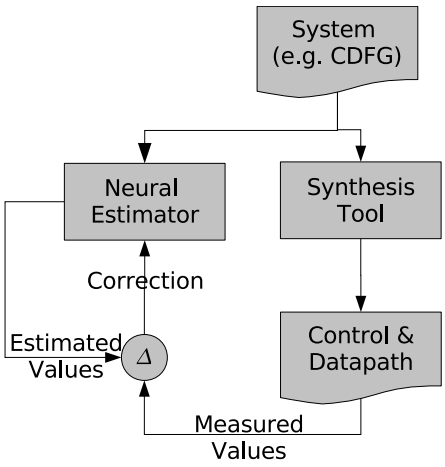


Figure 2.1: The NESTIMATOR neural estimator from [31] working together with a synthesis tool. The results of synthesis are compared with estimates and the neural network is adapted accordingly.

only taking into account standard LUTs and multiplexers as interconnection structures. Furthermore, optimizations during synthesis are not taken into account. Although these simplifications make the estimation process considerably faster, they also reduce the accuracy of the result.

Where some area estimation techniques aim to be deterministic and are based on knowledge about synthesis, the neural estimator (NESTIMATOR) in [31] takes a more non-deterministic approach. The estimator in this paper is first "taught" to correlate characteristics of a behavioral design to synthesis results by providing it with hundreds of examples. The setup is depicted in Figure 2.1. The feed-forward neural network used in this paper consists of 4 layers of neurons. The hidden layers use non-linear sigmoid neurons and the output layer uses linear neurons. The input to the neural network are several metrics characterizing CDFGs:

- Number of allowed time steps
- Maximum allowed clock period
- Average delay and area and variances of each Functional Unit (FU) type
- Number of nodes using each FU type
- Number of nodes
- Average bit-width of nodes and variance (indicates node and interconnect area)
- Average path length and variance (indicates interconnect area)
- Average number of inputs/outputs and variance (indicates interconnect and controller area)

Pattern	Node type(s)	Formula
Constant	INPUT, OUTPUT	$y = C$
Linear	Unsigned ADD and SUBTRACT	$y = p_0 + p_1 n$
Quadratic	(Un)signed integer MULTIPLICATION	$y = p_0 + p_1(n - p_2)^2$
Bi-product	(Un)signed multi-input ADD and SUBTRACT	$y = (m - p_0)(n - p_1) + p_2$
Multi-linear	multi-input AND or OR	$y = p_0 + (p_1 * x/2)(z/2 - 1)$

Table 2.3: Bit-based area models of the main operators in LUTs, as mentioned in [60].(y=area(LUTs), n=bit-width, m=inputs, p_i=experimentally determined constants)

- Average minimal and maximal lifetime of outputs and variances (indicates storage cost)
- Complexity of the CDFG (indicates parallelism)

An approach targeted specifically at partitioning algorithms is introduced in [95]. This paper describes how a preprocessed information data structure holding basic design information can be maintained between successive iterations making it possible to get an updated estimate of the area during every new iteration of the partitioning algorithm in constant time. [30] demonstrates the usefulness of incremental estimation by integrating the technique into the COSYN algorithm.

Another approach concerning estimation during the partitioning process is defined in [66]. Here the optimal cost of a given partitioning for the minimal execution time is calculated. The optimal cost is determined by adding the costs of single nodes while accounting for the sharing of resources. Resource sharing is represented by the sharing factor which is based on the similarity between nodes. Similarity may be defined in multiple ways, but must indicate to what level nodes can share hardware.

In [51] the estimation is executed before the partitioning process and therefore tries to be independent of specific synthesis tools. Because of this, the approach taken in this paper only takes into account the data paths. The estimates are used in a cost function in a partitioning algorithm and only need to be indicators of the actual area. This means that, although disregarding the control paths does result in a significant error, the metric can still be applied in partitioning.

As part of the fine-grained and coarse-grained partitioning strategies targeted specifically at FPGAs in [89] the authors present an area estimation approach. The partitioning strategy utilizes an area estimation algorithm that is based on summation. However, this summation handles computation area and storage area separately. This way the different logic used for storage (flip-flops) and computation(LUTs) is more accurately represented in the estimate.

In an effort to guide optimizations in an SA-C compiler [60] introduced an area estimation technique that captures the impact of compiler optimizations to the area

of a design. The estimator executes between the optimization and synthesis phases. Because the estimator is targeted at a specific compiler, the estimator can use detailed characterizations of the nodes in the DFG, depicted in Table 2.3. The estimate is further refined by looking for common patterns of synthesis optimization, and adjusting the estimate accordingly.

Instead of estimating the resource utilization of an entire design, some papers try to estimate the impact of specific aspects of a design. In [21], for example, we encounter a way to estimate the effect of loop unrolling on the area. The approach models pipelining and full- and partial unrolling of loops and this way accounts for an increase of the number operators and coarser grain array indexing. The paper mentions that unrolling of outer loops, loop strip-mining, loop merging, and optimizing loop-unrolling side-effects should be investigated to make the estimation more accurate.

A narrower approach focussing only on boolean functions is presented in [73]. It demonstrates the use of a complexity measure in estimating the area of high level descriptions. This complexity measure is based on the sizes and probabilities of prime implicants in the on-set of boolean functions and shows an exponential relation with the area needed to implement the functions. This relationship is the basis for the area estimation in this paper.

Estimating area specifically from software oriented languages like C or Matlab has been covered in several papers. The researchers in [57] tried to base the area estimation on extracting relevant operations from the source code, matching modules from a library to the operations, and use timing information to find out where resources can be shared. The accuracy of this approach was somewhat disappointing, because

- the authors of [57] did not use the impact of the control logic in the area estimation
- the approach did not account for optimizations
- the algorithm needs the C description to have a hierarchy close to the hierarchy of the final design.

Using Matlab specifications as a starting point [16] uses simulation to obtain execution traces that drive the estimation process. The traces are used to build a DFG of the program. Together with an FPGA performance model that determines the characteristics of specific operations based on bit-width and clock speed, the DFG is used to estimate the area required by the Matlab code. The authors have chosen to use an area/latency grid to schedule the DFG before estimation. Using this approach they try to account for resource sharing and area constraints at the same time.

As a part of the research around the MATCH Compiler [11] the authors of [71] present an area estimator to provide the compiler with information for automatic design space exploration. The estimation comprises counting all operations and registers and uses a simple heuristic formula (Equation (2.1)) to calculate the number of Configurable Logic Block s (CLBs) that is needed for a design.

$$N_{CLBs} = 1.15 * max(N_f/2, N_r) \tag{2.1}$$

N_f: # of function generators
N_r: # of registers

The algorithm uses a library to count the number of function generators used by the operations. To get more accurate estimates. the paper describes a type refinement pass, which determines the bit–widths needed for each variable and operation before estimation.

The authors of [15] introduced an estimation technique where an entire trade-off curve is calculated. Their algorithm uses H-CDFGs, which are CDFGs with other CDFGs as nodes. From the bottom levels of the H-CDFG up, the CDFGs are scheduled and resources are allocated using different time constraint values. Calculation of the total number of required resources of multiple CDFGs is based on several heuristic and deterministic summation rules that account for resource sharing.

2.2.1.2 Speed

In order to get an approximation of the performance of a design, the research community has proposed many ways of estimating latency and speed–up. Performance measures like these can indicate if a design is within performance constraints and can drive hardware/software partitioning.

An example of latency estimation can be found in [85] for example. Apart from estimating area, this paper also presents a way to calculate a lower bound on latency given certain resource constraints. The lower bound is based on the critical path latency and depending on the resource constraints is increased by the extra delay of every module on the critical path due to module constraints.

In [66] the latency of a hybrid hardware/software system is estimated by determining the critical path of a Co-design graph as defined in the same paper. The Co-design graph is a task graph with annotated nodes and edges. For latency estimation the nodes are annotated with latency information and for edges the graph records whether they are software or hardware edges and whether they indicate a control or a data dependency. The critical path is then determined by finding the longest path in the task graph.

As with area [15] tries to estimate entire trade-off curves for the latency of a design using different timing constraints. Again the H-CDFGs are traversed in a bottom-up fashion and the latency is calculated at every level by accounting for the allocated resources due to the timing constraints.

The COSYN [30] and CRUSADE [29] co-synthesis algorithms estimate finish times of tasks with the longest path algorithm. Described in [30] the algorithm finds the longest path in a DAG taking in to respect both the execution and communication times of tasks and links respectively.

The partitioning algorithms in [89] discussed earlier, also estimate latency by first determining the longest path. This preliminary value is then refined by adding the latency of moving input and output values from and to memory, the latency of transferring data over partition boundaries, and the latency due to synchronization between tasks.

Paper	Dynamic/ static	Level of Design	Application Domain	Data structure	Node model	Granularity	Strategy	Complexity	Error
[35]	Static	Instruction-level	n/a	Intermediate code	n/a	Application	Complexity	n/a	8%
[16]	Dynamic	System	Communicatic	(annotated) DFG	Bit-width based	Entire graph	Scheduling	$O(simulati$ nonlinear	10%
[71]	Static	System/ Behavioral	Multimedia, Mathemat-ical	VHDL Ab-stract Syn-tax Tree	Bit-width based	Entire Graph	Scheduling	linear	13%
[85]	Static	Behavioral	Multimedia	DFG	Simple	entire graph	Scheduling	$O(nc^2)$ (c: time steps)	n/a
[50]	Dynamic	Behavioral	Multimedia, Cryptog-raphy, Com-pression, General Purpose	n/a	n/a	Loops	Simulation	$O(simulati$ $O(1)$	n/a
[66]	Static	Behavioral (during partition-ing)	n/a	(annotated) DFG	Simple	entire graph	Scheduling	$O(n^2 \log n)$	n/a
[19]	Static	Behavioral	Communicatic	n/a	Bit-width based	Functions (=Occam processes)	Complexity	n/a	n/a
[30]	Static	Behavioral (during partition-ing)	n/a	Task graph	Simple	Entire graph	Incremental	linear	n/a
[29]	Static	Behavioral (during partition-ing)	Communicatic	Task graph	Simple	Entire graph	Scheduling	linear	n/a
[15]	Static	System/ Behavioral	Multimedia	H-CDFG	Simple	Entire Graph	Hierarchical, Allocation (bipartite matching)	$O(n \log n)$	n/a
[89]	Static	Behavioral (during partition-ing)	Multimedia, General Purpose	(C) DFG	Simple	Node clus-ters	Scheduling	n/a	n/a
[91]	Dynamic	Behavioral	Multimedia, Cryptog-raphy, General Purpose	Hierarchical Loop Graphs	n/a	Loops	Simulation	$O(simulati$ $O(1)$	n/a
[81]	Static	Behavioral	Multimedia, General Purpose	PACT HDL Abstract Syntax Tree	Simple	Entire Graph	Scheduling	nonlinear	n/a

Table 2.4: Different classifications of speed estimation found in several papers and indications of estimation error in those papers. For an explanation of the classifications, please refer to Section 2.2.4.

Area estimation from Matlab code as presented in [16, 71] was already discussed in the previous section. These two papers also describe delay estimation techniques. In [16] the critical path is used as latency estimate. If area constraints introduce extra delays on the critical path, the latency estimate is changed accordingly.

The authors of [71] take a more elaborate approach. They split the delay estimation between estimating the critical path delay and the interconnection delay. The delay of single tasks on the critical path are not retrieved from a normal library, but are calculated using generic delay formulas describing FUs and using fan–in and bit width as inputs. Libraries can become more compact this way. In general the paper models the delay of an operation with Equation (2.2).

Operation	**Architecture**	$f(\delta, n)$
+	Ripple carry	$2n\delta$
+	CCLA(n/p^2) p = BCLA dimension	if$(n = 1)11\delta$ else $11\delta(10 + \log p(1 + n(n/4 - 1))(n/4 + 1)$
*	Baugh Wooley	if $(n = 1)2\delta$ else $4n\delta$
*	Bisection	if $(n = 1)2\delta$ else $4n\delta$
/	Restoring (Dean)	$(3n^2 + 1)\delta$
/	Non-Restoring (Guild)	$3(n + 1)^2\delta$
/	Non-Restoring with 2-level CLA and Carry-save (Cappa-Hamacher)	$(11n + 12)\delta$
/	Non-Restoring with 1-level CLA and Carry-save (Cappa-Hamacher)	$(9n + 10)\delta$

Table 2.5: Bit-based timing models of the main operators as mentioned in [19].

$$\delta = a + bn_{fan-in} + \sum_{i=0}^{i=n_{fan-in}} c_i m_i \tag{2.2}$$

m_i: bit-width of input i
n_{fan-in}: number of inputs
a, b, c_i: experimentally determined constants

Using estimates for average interconnection length and physical FPGA wire lengths, the algorithm determines the average number of physical wires and programmable interconnect points and therefore the average interconnection delay.

Instead of determining a value for the latency of a design, papers [50, 91] use a temporal profile of a software program to determine the maximum speed-up that can be obtained by moving certain parts of the program to hardware. The more a function, for example, contributes to the total execution time of a program, the larger the potential speed-up when it is moved to hardware. However, [50] mentions that based on the examined programs a) almost 100% of the candidates for hardware implementation contribute 1% or less to the total execution time and b) memory access time and memory access rate have a significant impact on the performance gain that can be achieved. [91], on the other hand, puts the former comment in perspective by pointing at the so-called *90-10 rule*, which states that 90% of the execution time is spent in 10% of the code. It goes on to examine the validity of this rule and concludes that many application do indeed exhibit this behavior. Finally the paper presents a simple formula to calculate the expected speed up of partial hardware implementation based on loop frequency.

In an attempt to define a performance measure that makes it possible to compare hardware and software performance during Co-simulation, [19] describes using the Cycles Per Instruction (CPI) of OCCAM-II to indicate performance. The paper describes how to calculate the CPI for software-bound processes and how hardware-bound processes can be characterized by a so-called equivalent CPI. Both measures use a parameter that depends on the architecture, compiler, synthesis tool, and scheduling policy. These measures have to be determined for every new architecture that needs to be simulated. The hardware measure utilizes bit-based operator models to estimate timing characteristics. The models of the main operators is depicted in Table 2.5.

The algorithm in [81] measures the speed-up of a design by counting the number of control steps that are present in the control nodes in a CDFG and the number of times they are executed. Then it applies different formulas for hardware and software to obtain the execution times of both implementation alternatives. Simply dividing these yields the speed-up factor.

The μ–profiler as described in [35] uses the number of cycles gained by hardware implementation to drive the discovery and selection of custom instructions for Application Specific Instruction-set Processor (ASIP) designs. Both the decrease in computation cycles by moving a pattern in the intermediate code of an application to hardware and the frequency of that pattern contribute to the measure. The author admits this measure is crude, but argues it is useful for early discrimination between potential candidates for hardware implementation and patterns with insufficient prospect for speed gains. In the future this measure may be improved by accounting for Instruction-Level Parallelism (ILP), pipelining, and other software optimizations.

One such optimization, i.e. loop unrolling, is specifically studied in [21]. We already discussed how the authors of this paper estimate area in the previous section and speed–up is handled in the same way. Results, however, show a larger discrepancy between estimates and measurements for speed–up, than for area. The authors explain how this might be caused by e.g. unbalanced pipeline stages due to resource constraints or other loop optimization side-effects and mentions that the underlying model of the algorithm should facilitate more complex algorithms that do account for these effects.

2.2.1.3 Power

In order to increase battery lifetime in mobile and embedded systems, reduce heat dissipation in high performance designs, etc. many researchers have sought to estimate the power usage during system design. In [62], for example, we find a taxonomy of several power estimation techniques at different levels. At the highest level the authors discern several design levels at which power can be estimated. For each level the authors further classified the estimation strategies as depicted in Table 2.7. The different levels in the taxonomy also correspond to the design process. At the early stages power intensive parts can be identified with system-level estimation. Later on when a behavioral description is available, instruction- and behavioral level estimation can help guide the partitioning process and optimization of software and hardware parts. Finally, before actual synthesis is performed, architectural estimation can help choose from different architectures.

Paper	Dynamic/ static	Level of Design	Application Domain	Data structure	Node model	Granularity	Strategy	Complexity	Error
[56]	Static (dynamically determined input probabilities)	Behavioral	Communicatic General Purpose	CDFG, state transition graph	Activity-based model	Entire Graph	Hierarchical	$O(simulati$ $nonlinear$	11.8%
[39]	n/a	System	n/a	n/a	n/a	n/a	Complexity	n/a	n/a
[30]	Static	Behavioral	n/a	Task graph	Simple	Entire Graph	Hierarchical	linear	n/a
[15]	Static	System/Beha	Multimedia	H-CDFG	Simple	Entire Graph	Hierarchical, Allocation (bipartite matching)	$O(n \log n)$	n/a

Table 2.6: Different classifications of power estimation found in several papers and indications of estimation error in those papers. For an explanation of the classifications, please refer to Section 2.2.4.

Level	Class	Based on	Pros	Cons
Architecture	*Analytical*	*Complexity*	• requires little information	• inaccurate activity modeling • no block specific estimates
		Activity	• block specific estimates	• uniform distribution of capacitance
	Empirical	*Fixed Activity*	• non-uniform distribution of capacitance	• disregards data activity
		Activity Sensitive	• regards data activity • strong link to real implementations	• n/a
Behavioural	*Static*	*Activity*	• estimates indicate general trends	• no absolute accuracy
	Dynamic		• easy handling of data dependencies	• much slower than static • requires input specification
Instruction			• applicable to CPUs and ASIPs	• bad estimates for some instructions
System			• can guide optimization efforts early on	• very low accuracy

Table 2.7: Taxonomy of power estimation techniques as presented in [62].

The system-level partitioning algorithm in the TOSCA co-design environment [10] uses several power evaluation metrics to drive system level partitioning. In order to choose the best design alternative several metrics indicative of good power performance for different types of operating modes are proposed in [39]. The different system characterizations are:

- *Fixed Throughput Mode*
  These systems are characterized for power by the *Power to Throughput Ratio*, which is the same as energy per operation. This metric is applicable to e.g. Digital

Signal Processing (DSP) applications.

- *Maximum Throughput Mode*
  In these systems power is characterized by the *Energy to Throughput Ratio*. Energy and throughput, in this case, mean maximum energy per operation and maximum throughput respectively. Microprocessor-based systems are an example of these systems.

- *Burst Throughput Mode*
  Systems that only perform in bursts can be characterized by a modified *Energy to Throughput Ratio*. Energy in this case the energy during computation and the energy during idling per total number of operations. Systems with user-interaction are often in burst throughput mode.

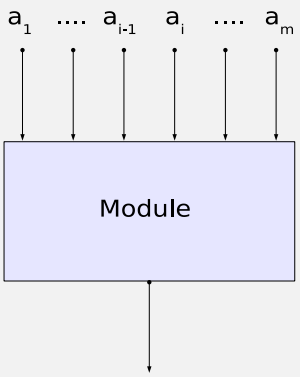
- *Area-Constrained Systems*
  Two metrics are proposed that can characterize Area-Constrained systems: The *Power by Area Product*, which allows to optimize for power, and the *Energy by Area Product*, which allows for optimization of area and power together. Many designs will have some degree of limitation for area.

Apart from presenting these metrics the paper also argues that communication between hardware and software parts contributes significantly to the power consumption of the design. When off-chip buses are involved this contribution can be even higher. In order to estimate the power used by communication from the hardware, TOSCA calculates the bus switching activity using the bus width, required bandwidth, and the encoding scheme for data and addresses.

An example of static power estimation at the behavioral level, as discussed in [62], is presented in [15], along with area and speed estimation techniques, discussed earlier. The paper estimates trade-off curves for power consumption against varying timing constraints. The power estimation differs from the area and time estimation by taking into account the execution frequencies of FUs and the clock frequency.

The authors of [73] use the area prediction method discussed in the previous section together with estimates of the average node switching activity and gate capacitance. Problems with their approach are the need for additional estimators for switching activity and the restriction to single-output boolean functions.

The contribution of control-flow circuits to the power consumption of a system is often assumed to be negligible. This may be a fair assumption for data-flow systems, but for control-flow intensive designs this aspect must be considered during power estimation. One such approach can be found in [56]. This particular approach utilizes both State Transition Graph s (STGs) annotated with branch probabilities and CDFGs. Using a generic model for modules all edge capacitances are calculated using the probability of each state and each transition. Special probabilities are calculated in the presence of loops. Furthermore, the paper describes how the capacitance of the controller itself is estimated with a simple formula based on the number of states in the



Switched Capacitance Matrix

The m-input n-bit module depicted on the left has a corresponding $m \times n$ matrix of switching activities for each bit. The module capacitance can then be estimated with Equation (2.3).

$$C_{mod} = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{i_j} \times c_{i_j} \tag{2.3}$$

a_{i_j} and c_{i_j} are respectively the activity and a module-dependent constant for bit j of input i. The constant is determined by switch-level simulation of the module and the activity represents the probability that the bit will change between successive states.

Figure 2.2: Module power model used in [56].

controller.

In the COSYN-LP algorithm [30] energy levels of tasks in a task graph are estimated by using execution and communication time as starting points. The energy level of a task is then estimated by adding the energy for all fan-out edges and all preceding nodes to the tasks own energy level. This process starts from the bottom at the sink nodes and progresses upwards via the fan-in edges. If after partitioning multiple allocations have the same power level, an alternate estimation strategy is used, where different heuristics are used for processors, FPGAs and links.

2.2.2 Other metrics

In Section 2.2.1 we discussed various methods of estimating the more obvious aspects of hardware design. In the last decade, however, other aspects like memory usage, communication, and design complexity have also been researched as possible driving forces for system design. We will now briefly review some of these metrics and their uses.

2.2.2.1 Communication

Estimating the amount of communication is also present in some area, speed, and time estimation techniques (e.g. [85, 81, 71, 30, 39, 89]), and not without reason: communication requires interconnect circuitry, requires a fair amount of power, and introduces communication delays into the design. Therefore, accurate communication estimation can be a valuable asset to any system designer.

A more focused effort to estimate the communication latency can be found in [51]. This paper specifically estimates the communication between hardware and software segments in a hybrid system. They assume a shared memory model is used for communication between hardware and software and also that communication with hardware only occurs with adjacent hardware modules. The latter is a fair assumption in case of data-intensive designs.

Another paper on communication is [55]. In this paper we see how the amount of communication can be represented by the sum of all edge weights, or Total Edge Weight (TEW). As edge weights the paper uses the amount of data transferred along the edges (in bits).

2.2.2.2 Memory Usage

Few hardware-oriented memory usage estimation strategies have been proposed, but in e.g. [50], we can see that memory usage can have significant impact on the speed of a design. Furthermore, communication with the memory system and refreshing of volatile memory can impact power usage. And finally, memory modules, extra interconnect, and memory management circuits require area. As with communication it seems clear that memory estimation can be very useful to a designer.

While many software profilers measure or characterize memory usage [39, 33, 48, 25] and some hardware/software partitioning algorithms [89, 29] take into account memory aspects of a design, to the best knowledge of the author there has been little research into memory usage in hybrid architectures.

One example of memory size estimation for hybrid architectures can be found in [59], where the size of individual data dependencies is estimated. The described algorithm focuses mainly on dependencies between loop iterations and requires single-assignment code, but it does give valuable information on memory size even in the early stages of design, when only a partially fixed execution ordering is present.

The authors of [103] point out the difficulty of hardware synthesis of programs with pointers. Traditional context-insensitive pointer analysis does not suffice. To answer this problem the paper describes a context-sensitive method of analyzing pointers in a program based on symbolic transfer functions. These functions capture how a function influences the program state. The program state and the symbolic transfer functions are represented as boolean expressions. The pointer analysis scheme presented utilizes binary decision diagrams to speed up the analysis, making context-sensitive pointer analysis feasible.

In [25] we find a study on reference locality in software programs resulting in some metrics that characterize data reference locality in a program. These metrics are based on hot data streams derived by bursty tracing [48]. Originally targeted at cache optimization on modern processors, this information may also help optimize local memory utilization in hybrid architectures or help decide whether a function can be efficiently implemented in hardware, because a function with an intensive and erratic memory access pattern might not benefit from hardware implementation at all.

2.2.3 Software metrics and comparability

In defining a candidate selection model for hybrid architectures not only hardware metrics are important, but also software metrics. In previous sections we have already discussed software measures that may also describe hardware aspects ([25]) or may indicate code most susceptible to optimizations ([91]), but software measures also make comparisons of hardware and software implementations possible. In [47], for example, we find software energy models that enable the authors to compare different hardware / software partitions. However, such comparisons do yield some problems:

- *Differences in precision*
  The precision of a software/hardware measure might be less accurate than its hardware/software equivalent. This means comparisons are only as good as the least accurate measure.

- *Differences in algorithms*
  Different measures are often determined using different algorithms, which makes it unclear if they are comparable in a straightforward way. Look, for example, at cyclomatic complexity; it's not at all clear if software (C) and hardware (VHDL) cyclomatic complexity (see Section 3.2.3) are equivalent or comparable. To the best of my knowledge no investigation in comparability between such measures exists.

2.2.4 Classifying metrics

Having discussed different estimation strategies and metrics for several aspects of a design we now move on to classification. In the following few sections I discuss different aspects of the presented metrics and estimation strategies, and explore possible classifications.

2.2.4.1 Dynamic vs. Static

One aspect of estimation strategies is whether they try to analyze a design statically, at compile-time, or dynamically, at run-time or during simulation. In the following, we refer to these as static estimation and dynamic estimation respectively.

From the papers discussed so far it seems static estimation has been dominant in the field of hardware/software partitioning and hardware synthesis. We can explain this if we take into account the many iterations partitioning algorithms may go through. If every iteration performs a potentially expensive simulation, partitioning can take a long time. This suggests that static estimation techniques aim to be fast more than they aim to be accurate.

Static estimation is also the dominant strategy in area estimation. Because area is almost always assumed to be fixed during run-time, this is only logical. When we look at hybrid architectures, however, the assumption that area is fixed during run-time does not have to apply. Future area estimation could take advantage of simulation to get dynamic area profiles.

Power estimation is often based on area estimates or area estimation techniques and thus power estimation, too, is mostly based on static estimation. In [62], however, we do find some cases of dynamic techniques like dynamic behavioral activity-based estimation

(Table 2.7). Power consumption, especially that of control-intensive designs, depends on the input of an algorithm as well as the implementation itself. Therefore dynamic estimation can be useful for reasoning about power in designs.

The few dynamic estimation strategies discussed so far [50, 91, 16, 25, 62] all concentrate on speed and memory usage. These characteristics are often dependent on the input and change during run-time. Still, hardware estimation mainly focuses on static estimation. If we look at software profiling and estimation, however, we find more dynamic techniques. It seems dynamic estimation techniques are more suitable for control-intensive designs, while static estimation is more suitable for data-intensive designs.

2.2.4.2 Level of design

Different estimation techniques require different levels of detail in a design and have various levels of accuracy. Therefore, it is useful to categorize estimation according to the design step it is targeted at. We have already seen a similar taxonomy for power estimation techniques in [62].

It is not the aim of my thesis to present an exhaustive study of system design steps, therefore we use a simplified model of system design as a tool for discussing estimation. We discriminate three levels of design: System level, Behavioral level, and RTL/Instruction level. As can be seen from the Figures 2.2, 2.4, and 2.6, the behavioral level is the predominant level in the reviewed papers. This is mainly because the papers were selected based on their relevance to *high level* synthesis.

1. System level
   In the early stages of design a lot of high level decisions have to be made in a relatively short time. In order to give the designer information on different design alternatives, fast system level estimation is necessary. In this phase accuracy is not very important yet and therefore, estimates should be only indicative of actual values. On the system level components are not yet (fully) specified using behavioral descriptions, or are modeled by a mathematical model. Papers on the MATCH compiler for Matlab models, for example, are considered to be on the system level in this view.

2. Behavioral level
   Later on in the design process system components will be refined with specific functional descriptions. This is the phase where many hardware/software partitioning algorithms are applied. The added detail potentially provides more accuracy in the estimates. This makes it possible to check various constraints on the design with greater certainty.

3. RTL/Instruction level
   After partitioning and hardware specification accurate estimation is possible for the hardware parts of the system. On the software side final estimates can be made using instruction level estimation. Estimation on these levels is often slow, because of the large amount of detail.

2.2.4.3 Data structures

The speed and precision of estimation often depend on the data structures and the strategies that are used. Different data structures might capture different aspects of a system, or have different levels of efficiency. Traditionally, hardware synthesis has utilized data flow graphs as data structures and conversely many estimation techniques focus on these graphs. Software estimation techniques, however, are more control oriented and so use other representations like control data flow graphs and call graphs. Most literature on hardware/software estimation uses one or more of the following data structures:

- DFG
  A DFG is a *DAG* where every node represents an operation and every edge represents a data dependency. Because there is no control information present in the DFG, the execution flow of the DFG is straight-forward and analysis is relatively easy. Many algorithms in hardware synthesis are based on these structures. The dependencies give information on which tasks can run in parallel. The absence of control information makes DFGs less suitable for representation of higher level functional specifications, like a C-program. A different term for DFG is *Task Graph*. These terms are often used interchangeably. Nevertheless, a DFG is often assumed to have finer grain nodes, like operations, while Task Graphs have coarser nodes, e.g. another DFG.

- Control Flow Graph (CFG)
  In the traditional Von-Neumann computing paradigm, an often used representation of programs during e.g.compilation or analysis is the CFG. In this directed but cyclic graph, edges denote the transference of control from one node to the other, where nodes represent basic blocks, i.e. blocks without branches or jumps. While this graph can accurately represent the control constructs in higher level languages, its disadvantage is the single thread of control inherent to the control dependencies. This makes discovering parallelism using CFGs difficult.

- H-CDFG
  To account for both data and control dependencies in high level designs, the research community came up with the CDFG. This graph is a data flow graph extended with control edges denoting control dependencies between nodes. Some papers utilize other representations of the control dependencies, e.g. in [41, 15] a Hierarchical CDFG or Hierarchical Sequence Graph is used, which captures control constructs as nodes in a DFG. A loop then becomes a node in a DFG, while the loop body is represented as a DFG on a lower level. Because of the presence of loops and branches, the run-time flow through a CDFG is not known in advance and estimation becomes more difficult. High level synthesis tools often use these representations because VHSIC Hardware Description Language (VHDL), C, and other imperative languages make use of these control constructs.

- STG
  Control-intensive designs often result in a non-trivial controller circuit. The impact of such a circuit on performance characteristics, like area and power, cannot be

neglected anymore. Furthermore, the power consumption and speed of the data-path can vary significantly depending on the state of the controller. In [56] a STG is used for modeling the controller in order to estimate the power consumption of the controller and the impact of the controller on data-path power consumption. Exact metrics are hard to determine, because of the significant data-dependencies inherent to control intensive designs. Therefore state transition probabilities are determined beforehand. This way the average characteristics can be determined.

- *Others*
  Apart from the (C) DFG representations, some other data-structures exist that are less common in hardware estimation. In software estimation, for example, we find Call Graphs and Abstract Syntax Tree (AST), among others. However, these models, like CFGs, do not always directly represent data dependencies, which makes determining parallelism inaccurate.

2.2.4.4 Strategies

Different strategies have been proposed for different data structures and models, but some approaches are more alike than others. It can be useful to group similar estimation algorithms into categories and characterize them. This section proposes several such categories, but does not aim to be an exhaustive list, nor are the categories mutually exclusive.

- *Scheduling*
  Mainly applied in speed estimation, scheduling in estimation mostly provides the timing for nodes in e.g. a DFG. After scheduling an indication of the latency of a circuit can be obtained by finding the longest path. Scheduling in estimation is borrowed from actual synthesis in order to obtain more accurate speed estimates. There are many types of scheduling algorithms, like list scheduling and force-directed scheduling. While scheduling often produces good estimates of the latency of a design, it can be time consuming (optimal scheduling is NP-complete). Note that scheduling is also used in several papers to acquire area estimates. Indeed, when scheduling without any bounds on resources, the maximum number of used resources for each resource type define the minimum required area.

- *Allocation*
  Another technique borrowed from synthesis is allocation. By actually allocating variables and operators to registers and FUs respectively, an indication of the number of resources is obtained. One such allocation strategy often found in estimation and synthesis is weighted-bipartite graph matching [49]. Matching algorithms are used in estimation to mimic resource allocation of data paths during synthesis. The idea behind this approach is to find a matching, i.e. a set of edges without common vertices, on a bipartite graph with one set of vertices representing variables (or operations), the other set of vertices representing registers (or FUs), and the edges representing possible mappings. An advantage of using synthesis-like allocation in estimation is that separate estimates for registers, FUs, and possibly interconnect can be obtained with most allocation schemes.

- *Weighted Sum/Hierarchical*
  One powerful template used in many algorithms is the Hierarchical approach. In estimation this is used especially in hierarchical models. By estimating metrics on subgraphs of a CDFG, for example, the next level in the CDFG hierarchy can be estimated more quickly. A commonly used Hierarchical approach is the weighted sum approach. In this approach the system is first divided in atomic parts, i.e. components in a library. Then the weighted sum of the parts indicates the estimate of the system. Weights can represent many aspects of the component like I/O bit-width, slack, and criticality. This approach is mostly used in area and power estimation.

- *Neural*
  When it is not obvious how the value of a proposed metric can be obtained, like hardware implementability, a neural network can be applied. Neural networks can be trained to recognize certain aspects of complex systems and is similar to linear regression. While this makes it possible to quantify hidden or complex aspects of a system, there are some drawbacks. First, defining a correct neural network, gathering a large enough training set, and training the network is time-consuming. Second, the trained network is not transparent, which makes it difficult at least to prove that results are correct. Furthermore, trained networks are specific to the data set used. For example, if a neural area predictor is trained on designs synthesized with tool A, then it may not be applicable to tool B.

- *Correlation and Complexity*
  Another way of estimating metric values is by correlating metrics like area, speed, etc. with other metrics that are easy to determine. For example, [73], correlates the area of boolean functions with the sizes of the minterms in the on-set of a boolean function represented by a complexity measure. Correlating different metrics does require extensive datasets, however. One class of measures that may be correlated to various metrics is the set of (software) complexity metrics. It seems plausible that (software) complexity is in some way related to area, speed, etc.

- *Simulation*
  An obvious estimation strategy used in some partitioning strategies is simulation. Instead of synthesizing a design, which is slow and can use up resources, only simulation of a functional model is performed. This strategy can result in fairly accurate measurements, because the system is evaluated at run-time, but is much slower than other approaches. In a Hierarchical approach, simulation might be applied on small parts at lower levels of the system, to balance speed and accuracy.

- *Incremental*
  Iterative hardware/software partitioning approaches often need to re-evaluate different aspects of a system for each iteration. Some partitioning algorithms, therefore utilize an incremental estimation scheme. These algorithms assume that temporary partitionings only change very little between successive iterations and therefore adjusting the previous estimation instead of re-estimating the measures can be beneficial.

2.2.4.5 Application Domains

Many estimation strategies discussed here have been targeted at specific application domains or have only been validated using a limited set of applications. In the Tables 2.2, 2.4, and 2.6 these application domains are specified. In my study of estimation I made the following inventory of application domains.

- *Multimedia*
  The multimedia application domain comprises audio, video, image processing, an 3d applications, among others. Common examples are MPEG2 encoding, image filtering, etc. In the field of Reconfigurable Computing much attention is given to this domain. One reason for this is the high performance requirements set by these applications. Because these applications often have a high degree of parallelism, these performance requirements can efficiently be met when using ASICs or FPGAs. Another reason is the continuing high demand for multimedia in mobile devices.

- *Mathematical*
  Another domain that might potentially exhibit a high level of parallelism is the domain of mathematical problems. Examples of such problems are matrix multiplication, logical closure, finite element method, fractals, etc.

- *Cryptography, Compression, and Error correction*
  With an increasing amount of private or classified information in digital form, security has become very important. Furthermore, the vast amounts of data transported over the Internet and stored on backup devices, require some form of compression and error correction. All three of these kinds of applications work on streams of data in more or less a serial manner. Many algorithms have been developed to tackle these problems, many of which do show a certain degree of parallelism. Examples are DES, Reijndael, MD5, Reed-Solomon, gzip, huffman, etc.

- *Communication*
  Some papers validate their findings using descriptions of communication circuits. Examples of such circuits are an Ethernet controller, a digital receiver, or the ILC16 HDLC link controller. Papers that target such systems, however, are not proven to be applicable to C-level descriptions.

- *General Purpose*
  Other application domains exist, but for this paper we group them in the domain of general purpose applications. This class therefore has many different levels of potential parallelism. Some applications in this class are an SQL server, bubble sort, binary search, specint, etc. Estimation that does not target a specific domain, makes them more usable in more general tool chains required for the acceptance of Reconfigurable Computing .

2.2.4.6 Use of libraries and component models

Many estimation algorithms make use of component libraries containing information on the estimated properties for single components. The use of such libraries can make estimation both faster and more portable. Speed is gained by working on a coarser grained model and portability is increased because the estimation algorithm can target, for example, other architectures by changing the library.

Depending on the estimation technique, coarser and finer grained libraries are used. Several aspects are influenced by the granularity of the library. Coarser grained libraries hide more details of a design, making estimation faster. However, estimation using coarser grain libraries is less flexible and therefore less accurate. Furthermore, coarser grain libraries tend to be larger than finer grain libraries, because of the increased number of possible configurations for components.

Components in libraries can have precalculated estimates for e.g. area, but some papers introduce component models based on bit-width, number of inputs, type of module, etc. One advantage of such models is the added flexibility in designing a system, e.g. one is not limited to a fixed bit-width. Another advantage is the reduction of the size of the library, e.g. two-operand, three-operand, etc. addition can be modeled by one equation. The drawbacks of using component models are the added computation required to calculate the final estimates and the need to define the required models. The latter, however, has to be performed only once during the creation of the library.

2.2.4.7 Granularity of Estimation

When estimating the value of metrics for certain parts of the design the question arises what those parts are, i.e. what are the elements estimates are determined for. This is relevant for choosing an appropriate estimation technique for a partitioning model. To illustrate this, compare a model partitioning loops when only function-level estimates are available. And in fact estimation granularity almost always corresponds to partitioning granularity. In the papers I have reviewed in this section several levels of granularity of estimation occur which I classify in the following groups.

- *Loop or basic block level*
  Because loop parallelization is well understood, loops are often chosen as candidates for hardware implementation. In an effort to guide this selection process several estimation techniques have been developed that estimate metrics for loops. Sequential parts of code, or basic blocks, and functions can be identified as special kinds of loops in order to make the estimation process more general. In the papers reviewed here, this kind of estimation is mostly found in speed estimation.

- *Function or process level*
  An obvious level of partitioning is the function level, because during design of a program or system different functionalities are already partitioned into functions. It would be logical to estimate on this level as well. Oddly, only two papers [95, 19] in this review acted on the function level.

- *Cluster level*
  Instead of using parts of an application defined in the program description, like functions and loops, clusters of nodes grouped together for other reasons can be the object of estimation. A cluster could be, for example, a segment in a multi-segmented partitioning. This level only appears in one paper reviewed here [89].

- *Application or graph level*
  For the most part the discussed papers estimate metrics for an entire application or system graph. Such estimates are not directly useful for the partitioning process, however, before estimation is applied the system graph could be divided into smaller parts.

2.2.4.8 Error in estimation

In order to utilize an estimation algorithm during design an idea of its precision is needed. However, some measures are not intended to be exact figures, but aim at being comparable. In [56] this comparability is denoted by the *tracking index*. Regrettably, other papers give no quantification of comparability. In fact most papers discussed here do not mention any quantitative error characteristics.

Nonetheless, there are basically three measures characterizing error behavior that are important for use in a partitioning model: average error, worst-case error, and tracking index. An ideal estimate has a low average error, i.e. is very precise, has a worst-case error that is not significantly larger than the average error, i.e. is generally applicable, and has a good tracking index, i.e. is comparable. When defining a new partitioning model, these aspects should be considered.

2.2.5 Characterizing hardware synthesis and optimization

Some estimation approaches we discussed specifically accounted for synthesis optimizations. This can be useful, because optimizations can have large influence on the eventual values of e.g. area, speed, etc. Other papers implicitly account for optimization, e.g. a neural estimator, because the estimator is correlated to the synthesis results. A reason to explicitly calculate for the impact of optimization could be that application of optimizations can be unbalanced, i.e. only applied to specific parts of the design [21].

The optimization-aware estimation techniques discussed in this paper have mainly focused on loop optimizations, like loop strip-mining, loop unrolling, loop pipelining, etc. [60, 21, 56]. However, high level synthesis tools and compilers also use other optimizations, which are not yet accounted for in estimation techniques discussed in this paper. Among those are pointer analysis [83], procedure exlining [94], partial evaluation [18], etc. Accurate estimation tools should somehow account for such optimizations, however, not many such tools have been proposed in literature.

2.3 Reconfigurable Computing Projects and Toolchains

Now that I have presented an overview in current research in hardware/software partitioning and hardware estimation, I discuss several research projects in the field of Reconfigurable Computing . More specifically, I discuss those projects that have proposed some kind of automatic toolchain for hardware/software partitioning and/or software to hardware translation. The main focus, however, is on the Delft Workbench project at the Delft University of Technology.

2.3.1 Delft Workbench and MOLEN

As part of the Computer Engineering department of the faculty of Electrical Engineering, Mathematics, and Computer Science at the Delft University of Technology, we have a research project called Delft Workbench, which has been working on Reconfigurable Computing for some time. Members of the project have developed a reconfigurable programming paradigm, with an accompanying platform, called the MOLEN programming paradigm [97] and the MOLEN polymorphic processor [76] respectively. Furthermore, there is a research project targeted at design automation for this retargetable platform, called Delft Workbench [1]. Because these projects are the direct context of my thesis, I discuss them in more detail in this section.

2.3.1.1 MOLEN

The MOLEN programming paradigm features parallel hardware and concurrent hardware processes, but is intended to be sequentially consistent, i.e. the result must be the same as when the program would have been executed sequentially, and is targeted at single-program execution. As is mentioned in [97, 76], this paradigm has been developed to cope with 4 problems commonly associated with Reconfigurable Computing :

- *Opcode space explosion*
  If new instructions are defined for every configuration on a reconfigurable platform, a potentially unlimited amount of opcodes are needed to be able to implement a broad number of applications, however, a typical architecture has only a limited amount of unused opcodes available.
- *Limitation of the number of parameters*
  Several Reconfigurable Computing approaches offer only a limited amount of input and output parameters. The maximum amount of parallelism that can be attained, therefore, is limited too.
- *Lack of parallel execution support*
  Many architectures do not facilitate executing sequential data-independent operations or configurations in parallel.
- *Lack of modularity*
  The configurations used in reconfigurable systems are often specific to a certain platform or technology. This makes it quite laborious and thus expensive to port configurations to another platform.

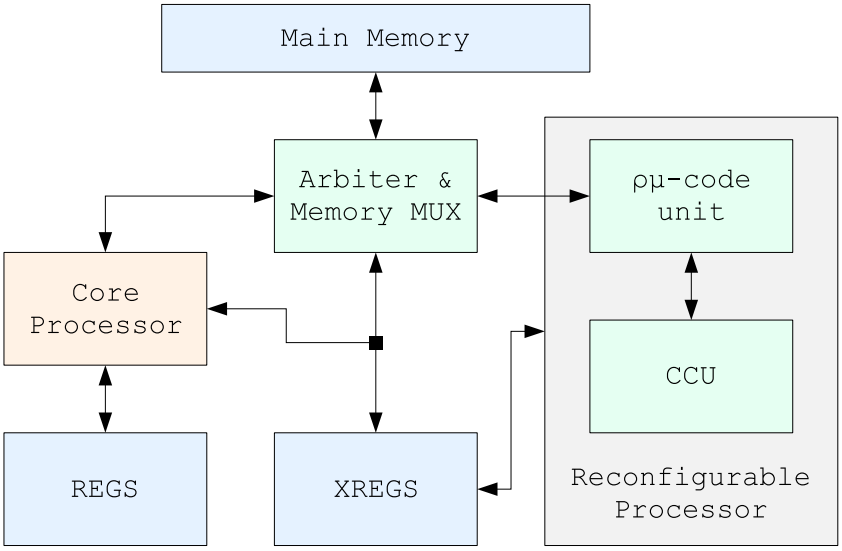


Figure 2.3: A basic overview of the general MOLEN platform.

In order to provide solutions for these problems the MOLEN programming paradigm suggests a limited instruction set extension. This extension provides instructions to load and execute configurations, a large register set for parameter passing, and the possibility to execute different configurations on the reconfigurable unit in parallel.

Loading configurations is implemented using configuration microcode, which is code that performs the actual configuration. This way different types of reconfigurable units can be configured without the need for changing the MOLEN architecture providing a much needed degree of modularity. A so-called `SET` operation is defined that loads the configuration microcode and initiates the configuration procedure.

When a configuration is completed the added functionality can be executed using an `EXECUTE` instruction. The `EXECUTE` instruction uses one opcode in the base opcode space and provides $2^{(n-o)}$ (where n = no. bits per instruction, o = no. of bits per opcode) additional configured operations, addressing the problem of the opcode space explosion. The `EXECUTE` instruction loads an execution microcode program into the reconfigurable unit. This program is then executed using the configuration previously loaded by the `SET` operation. Multiple available configurations may be executed in parallel. Explicit synchronization among the core processor and the different configurations can be performed using the a special instruction (`BREAK`).

Before the `EXECUTE` instruction can commence, however, the necessary data should be provided to the register set in the MOLEN architecture responsible for parameter passing (`XREGS`). Therefore, instructions (`MOVTX`, `MOVFX`) for moving data between the core processor and memory on one hand and the core processor and the `XREGS` on the other have been added as well.

The basic structure of the MOLEN platform is depicted in Figure 2.3. First instructions are fetched from memory and partially translated by the arbiter in order to decide whether they are redirected to the core processor or the reconfigurable unit.

The reconfigurable unit comprises of a Reconfigurable Micro-code ($\rho\mu$-code) unit and a Custom Computing Unit (CCU). The $\rho\mu$-code unit interprets and executes the configuration- and execution microcode, and the CCU is the actual configurable hardware part, e.g. an FPGA.

2.3.1.2 Delft Workbench

In Chapter 1 we saw that tools that support development for reconfigurable platforms are essential in order for them to be successful. Furthermore, I introduced the Delft Workbench project, which researches the development on reconfigurable architectures. More specifically, the project discerns four main objectives [1]:

- *Program Analysis and Performance Prediction*
  The Delft Workbench aims to identify functions in a program that might benefit the most from hardware implementation. These functions are then characterized by performance and area metrics, in order to find a set of functions with optimal increases in performance given the constrained area of the target platform. The result is a (semi-)automatic selection of a set of functions for migration to hardware.

- *C-to-VHDL mapping*
  In order to implement software functions in hardware, a designer would traditionally translate them to VHDL manually. Within the Delft Workbench, effort is being made to automate this process. One example of such automation is the C–to–VHDL compiler, that is being developed. It can translate C programs or candidate functions to VHDL. A problem with such a compiler, however, is the lack of interactive design space exploration it allows. For this purpose Delft Workbench envisions a library of FPGA configurations with an accompanying performance model, which helps the designer evaluate different design alternatives.

- *Retargetable Compiler*
  When a set of candidate functions is determined, the Delft Workbench provides a retargetable compiler that can compile these functions to a MOLEN architecture. Functions are translated to code for configuring the FPGA, moving the needed parameters, starting the execution, and retrieving the output. The compiler must deal with all compilation issues for the GPP as well as the added difficulties introduced by adding a reconfigurable unit.

- *Integration and Validation*
  The output of the retargetable compiler can now run on a real MOLEN implementation, allowing actual performance statistics, like FPGA reconfiguration time, to be obtained. These statistics are used as feedback for the Delft Workbench tool-flow. A refined set of candidate functions and a refined schedule are determined using the feedback information. This process iterates until the designer is satisfied with the results. The iterative nature of the process implies a tight coupling among the different tools in the tool-flow.

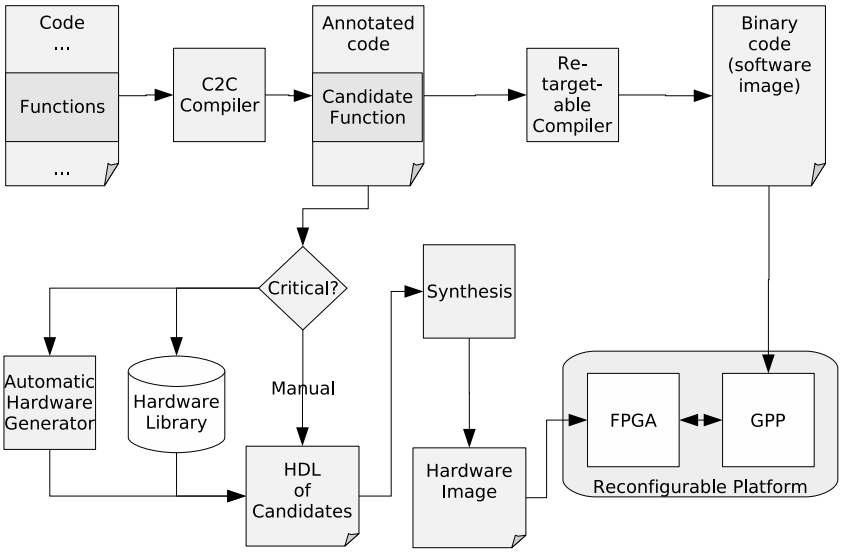


Figure 2.4: A basic overview of the general Delft Workbench tool-flow.

The Delft Workbench project focuses mainly on the MOLEN programming paradigm as its target platform. The tools and tool-flow envisioned by Delft Workbench are depicted in Figure 2.4. At the moment, partitioning of a program for the MOLEN platform is still manual. The Delft Workbench project proposes a code profiler and partitioner that automate this task. The goal of this profiler/partitioner is to increase the speed of the application, while staying within the bounds on the limited area of the reconfigurable unit. In order to decide where parts of a program should reside, the Delft Workbench specifies the need for a decision model based on metrics collected during profiling. Currently, no such model exists. And as mentioned in Chapter 1 it is the goal of my thesis to devise a preliminary version of such a model.

2.3.2 Other toolchains

Apart from Delft Workbench there are several other toolchains and research projects in the domain of Reconfigurable Computing . It is not the goal of my thesis to study these in detail. However, I briefly mention three of these alternatives here.

- *SPARK*
  SPARK[5, 43] is a C-to-VHDL high-level synthesis framework that aims to migrate software components (functions) to hardware. It focuses on control-intensive designs and optimizations and transformations for VHDL generation. Therefore, the compiler has a modular design of optimization passes, which makes it easy to implement new optimization passes and research different combinations of optimizations. At the moment, SPARK does not fully support the American National Standards Institute standard for the C programming language (ANSI-C) standard and in fact only supports a restricted subset of ANSI-C.

- *ROCCC (SA-C)*
  ROCCC[4, 40], too, is a C-to-VHDL compiler that aims to migrate software components to hardware. However, it focuses on loops instead of functions and on data-intensive designs instead of control-intensive designs. The tool is an evolution of SA-C, which was mentioned in Section 2.2, and therefore is highly applicable to streaming image processing.

- *MATCH*
  MATCH[11, 71], which was mentioned in Section 2.2, is a MATLAB compiler that targets at heterogeneous systems consisting of CPUs, DSPs, ASICs, and FPGAs. This system focuses on using Custom Off-The-Shelf (COTS) components from component libraries and aims not to be more than 2-4x slower than a manual implementation.

2.4 Conclusions

In this chapter we have reviewed the liturature on the subjects of hardware/software partitioning and hardware estimation. We have seen that there are many different approaches to these subjects. The novel aspects of the model presented here are its capability of estimating from the C-language level and the short time it takes to make predictions. Furthermore, we have discussed the Delft Workbench project and other reconfigurable support tools.

3 Software Metrics

In the previous chapter, we have reviewed earlier research into estimation of hardware characteristics and hardware/software partitioning. Virtually all research has based its estimation or partitioning strategies on synthesis-like schemes, simulation, or some form of summation of components. Such approaches often use some form of estimation library of elementary components. These libraries can be of different granularity - compare a basic arithmetic library with an image processing library - and sophistication - i.e. some have only simple values, others have component models based on e.g. bit-width.

The problem with these approaches is that they can be relatively slow and they often do not work well on higher level designs (C-programs). Furthermore, libraries can have large space requirements, especially when they are of a coarser grain. In our approach we do not translate the candidates to actual hardware components, but instead relate software metrics to hardware metrics in an analytical model. Indeed, it seems logical that a function with a high complexity would need more resources, than a less complex function. Furthermore, a relation between the number of memory accesses, variables, and operators on the one hand and area and speed-up on the other, seems plausible. One advantage of this approach is that software metrics can be determined very fast in most cases and, because software project management makes use of software metrics as well [26], some metrics might be available without cost. Another advantage is that the need for a library is eliminated, because the analytical model will take its place.

In this chapter I discuss software metrics that can be used in a model for hardware/-software partitioning. First, I elaborate on the concept of software metrics and how to classify and evaluate them. Then, I discuss multiple metrics in more detail.

3.1 Classifying Software Metrics

Before I start discussing software metrics, it is good to give a definition of what they are. Many different notions of software metrics can be found in literature. Often, no explicit definition is given. However, some authors provided a definition based on measurement theory, e.g. Fenton[37] and Munson[70]. They state that metrics are in fact measures, i.e. *an empirical objective assignment of a number (or symbol) to an entity to characterize a specific attribute.* Software metrics, therefore, are measures based on software entities. In this report I will use this definition of software metrics.

There are many different kinds of software metrics available, each targeted at different aspects of software development. Some are aimed at early project phases, others are postponed until later in the development process. Some indicate resource requirements of the project, others indicate fault behavior of the final product. And so on. Comparable to the classifications of estimation in Section 2.2.4, software metrics can be classified based on different aspects. In the following, we discuss several important

classifications with respect to the quantitative model.

First, both Munson and Fenton discriminate between primitive and derived measures. Primitive measures are those measures that are not composed of other measures. Thus, derived measures are measures that are composed of multiple primitive measures. Munson points out that derived measures should be used with much care. Often, derived measures are composed of primitive metrics without a careful analysis of the compatibility of these metrics. As an illustration, consider the the number of distinct variables in a program and the number of loops. Addition of these primitive metrics does not make sense, because they are completely different.

Second, one can make a distinction between static and dynamic measurement. As in Section 2.2.4, static measurement applies to analysis of the code before compilation, while dynamic measurement analyses the program while it runs. When applying software measurement to hardware estimation, one would expect static software measurement to be more applicable to area estimation, because area is largely dependent on the structure and functionality of the code. Dynamic software measurement would probably give more information on speed characteristics, because speed is also dependent on the dynamic behavior of a system, like the temporal contribution of a function to program run-time. However, these notions are intuitive only. In my research I have focused on static software measurement.

Another classification of software metrics is according to project phase or level of design. There are specific measures for the requirements analysis, design, implementation, etc. Because hardware/software partitioning in Delft Workbench is performed on C-code, I focus on metrics used during the implementation phase, i.e. at the behavioral level. Specifically, I focus on so-called, *(source) code metrics* or *measures of source code* [93, 37, 52, 70].

Furthermore, I would like to mention the taxonomy of software metrics defined by Fenton [37]. He discriminates between entities and attributes of entities within software measurement. Each entity and attribute has its own set of metrics associated with it. He defined the following entities and attributes:

- Entities:
  - *Processes*
    The tasks and activities associated with the software development.
  - *Products*
    The deliverables that are generated during a projects lifetime.
  - *Resources*
    The required items for every process.
- Attributes:
  - *Internal attributes*
    Attributes that are inherent to the entity it belongs to.

- *External attributes*
  Attributes that also depend on the entity's environment.

A similar classification can be found in [70], where Munson discerns four domains of measurement: people-, process-, product-, and environment metrics. Using these two classifications, my quantitative model will be restricted to metrics concerning internal attributes of product entities.

Then, it is necessary to present an important classification of metrics as determined in measurement theory, called *scales*. Metrics can be qualitative or quantitative to different extents, allowing different sets of operations. For example, it is not allowed to multiply two temperatures in Celsius, because that would have no meaning. Measurement theory defines four such scales:

1. *Nominal* scales assign entities to categories. There is no relation between the categories other than that they are mutually exclusive.
   operations allowed: $\{=, \neq\}$

2. *Ordinal* scales also assign entities to categories, but there is a sense of ordering in these categories.
   operations allowed: $\{<, \leq, =, \neq, \geq, >\}$

3. *Interval* scales add meaning to the distance between values.
   operations allowed: $\{<, \leq, =, \neq, \geq, >, +, -\}$

4. *Ratio* scales are interval scales with an absolute zero point. Most interval measures in practice are also ratio measures.
   operations allowed: $\{<, \leq, =, \neq, \geq, >, +, -, \div, \times\}$

5. *Absolute* scales are ratio scales that can only be determined in one way. For example the number of people in a room. The distinction between absolute and ratio scales is only rarely made, often when the method of measurement is very important.
   operations allowed: $\{<, \leq, =, \neq, \geq, >, +, -, \div, \times\}$

Because the proposed quantitative model should enable comparing the performance of different functions, candidate software metrics should at least be on an ordinal scale in order to be considered for our model. Preferably, the model should be based on ratio scales, because area and speed are also ratio measures. However, ratio scales among software metrics are scarce.

Finally, I mention the classification according to the nine properties or axioms formulated by Weyuker in [101]. She, proposed these properties in an attempt to make reasoning about the qualities and deficiencies more formal and standardized. The properties are targeted at complexity measures and try to capture intuitive aspects of complexity. In this report, I refer to these properties as *W1*, *W2*, etc. Below the properties are listed in the same order as they are found in [101]. Every property is presented as an algebraic expression where possible. In these expressions, P, Q, and R represent

programs, $\mu(X), X \in \{P, Q, R, ...\}$ is the metric value for program X, and # represents the concatenation operation.

- *Property 1 (W1)*

$$\exists P \; \exists Q \; [\mu(P) \neq \mu(Q)] \tag{3.1}$$

This means that the measure should not be completely trivial. For example, the knot measure, which measures the number of intersecting control edges in a control graph, is trivial in a structured language, where intersecting control edges do not exist.

- *Property 2 (W2)*

$$\begin{aligned} Z_{\hat{\mu}} &= \{X | \mu(X) = \hat{\mu}\} \\ &|Z_{\hat{\mu}}| < n, n \in \mathcal{N} \end{aligned} \tag{3.2}$$

Let $\hat{\mu}$ be any nonnegative metric value and let $Z_{\hat{\mu}}$ be the set of all programs that show this value. Then W2 states that $|Z_{\hat{\mu}}| < n$ for some finite number n. In other words, there are only a finite number of programs that have a complexity of $\hat{\mu}$.

- *Property 3 (W3)*

$$\exists P \; \exists Q \; [P \neq Q \wedge \mu(P) = \mu(Q)] \tag{3.3}$$

Property W3 comprises the possibility of two different programs to have the same metric value. If this does not hold true every program has a unique number and in fact is a nominal measure.

- *Property 4 (W4)*

$$\exists P \; \exists Q \; [P \equiv Q \wedge \mu(P) \neq \mu(Q)] \tag{3.4}$$

This property states that different implementations of the same function can have different metric values. In other words, a metric that has this property does not measure just the functionality of a program.

- *Property 5 (W5)*

$$\forall P \; \forall Q \; [\mu(P) \leq \mu(P \# Q)] \tag{3.5}$$

This means that the complexity of individual program components should never be larger than the complexity of the program as a whole.

- *Property 6a (W6a)*

$$\exists P \; \exists Q \; \exists R \; [\mu(P) = \mu(Q) \wedge \mu(P \# R) \neq \mu(Q \# R)] \tag{3.6}$$

Property 6b (W6b)

$$\exists P\ \exists Q\ \exists R\ [\mu(P) = \mu(Q) \wedge \mu(R\#P) \neq \mu(R\#Q)] \tag{3.7}$$

These two properties indicate the interaction of concatenated components. For example, assume P uses some of the same variables as R, while Q does not. Thus, R would be more dependent on P than Q. One could imagine a complexity metric that would, therefore, assign different values to $mu(P\#R)$ and $mu(Q\#R)$.

- *Property 7 (W7)*
  Let Q be a permutation of the statements in P.

  $$\exists P\ \exists Q\ [\mu(P) \neq \mu(Q)] \tag{3.8}$$

  Property W7 describes how a different ordering of statements can have an impact on the degree of complexity. As an illustration, compare two **if**-statements executed in sequence to the same statements nested in eachother.

- *Property 8 (W8)*

  $$\forall P\ \forall Q\ [P = Q \wedge \texttt{name}_P \neq \texttt{name}_Q \Rightarrow \mu(P) = \mu(Q)] \tag{3.9}$$

  Let P be a copy of Q with a different name, then $\mu(P) = \mu(Q)$. This property seems trivial, however, textual metrics like the number of characters in a function might assign different complexities to P and Q.

- *Property 9 (W9)*

  $$\exists P\ \exists Q\ [\mu(P) + \mu(Q) < \mu(P\#Q)] \tag{3.10}$$

  This property relates the notion that the composition of program parts adds not only the complexity of each program part, but also the complexity of module interaction. In other words, the the whole is more than the sum of it's parts.

Although these properties constitute a formal way of reasoning about software complexity measures, Weyuker has not proven this set to be a necessary or sufficient set of properties for good software measures. This notion is discussed in detail by Cherniavsky and Smith in [24], where they present a complexity measure ($|\ |_1$) that satisfies all Weyuker's properties and continue to show that this measure can assign significantly different values to almost identical code. Basically, they state that Weyuker's properties should not be used as axioms, nor should it be used outside the domain of imperative languages.

For my research I focus on imperative languages only (C). Therefore, these properties can be useful, although they can only be regarded as rules of thumb. Other sets of properties have been presented, for example in a broad review of software metric properties in [104] and by Lakshmanan et al. in [61]. Lakshmanan et al. specify additional properties for metrics targeted at control flow or more precise properties that characterize aspects of sequencing and nesting. Because I consider other metrics than

just control flow metrics and these metric properties are not the main focus of this report, I do not discuss these alternate properties further.

3.2 Candidate metrics

Now that I have discussed the important aspects of software metrics, I present several candidate metrics for inclusion into our model. Every metric is accompanied with a short description and a discussion according to the relevant classifications mentioned in Section 3.1. A summary of all metrics with some classifications can be seen in Figure 3.1.

3.2.1 Lines Of Code (LOC)

Probably the earliest metric used for software measurement was the number of Lines Of Code (LOC) a program contained. This metric originates from the time when programs were still written on *punched cards*[17] and more complex metrics were too time-consuming. Although this metric seems fairly straightforward, some difficulties arise when defining a line of code: should it include comment lines?, should it account for different outlining styles?, how do you compare LOC for different programming languages?, do you account for included libraries?, etc. Clearly, any measurement practice using LOC should provide a detailed description of what a line of code signifies.

Several software measurement models based on LOC provide such a definition, but by far the most commonly used model is COnstructive COst MOdel (COCOMO). First published by Boehm in [17], this model estimates the cost, effort, and schedule of software projects using a primitive metric called 1000 Delivered Source Instructions (KDSI). Boehm defined KDSI as the number of LOC or punched cards that are represented in delivered products, without counting comment lines. No distinction is made between programming languages or styles. The entire COCOMO model is based on the effort equation (Equation (3.11)), which indicates the number of man months (MM) required for a value of KDSI.

$$MM = 2.4(KDSI)^{1.05} \tag{3.11}$$

COCOMO and its successor COCOMO II are models used for project management. However, we are building a model for hardware/software partitioning. Therefore, we need an applicable definition of LOC. First of all, comment lines and outlining style have no influence on synthesis. Furthermore, we only consider single functions and finally, we only use the language C. Therefore, I use the statement count per C function as LOC metric.

The main advantages of the LOC metric are that it is very fast to measure and it is a ratio scale. However, LOC does not account for any control or data flow information. Although this might be an acceptable omission in a large project where only information on the project level is required, like in COCOMO, this becomes a serious deficiency when considering (considerably smaller) functions, because on the function level one cannot assume control constructs and data structures will be uniformly distributed. While I

Name	Scale	Derived	Weyuker								
			1	2	3	4	5	6	7	8	9
Total operands N_1	Ratio	no	+	+	+	+	+	−	−	+	−
Total operators N_2	Ratio	no	+	+	+	+	+	−	−	+	−
Unique operands η_1	Ratio	no	+	−	+	+	+	+	−	+	−
Unique operators η_2	Ratio	no	+	−	+	+	+	+	−	+	−
Loads	Ratio	no	+	+	+	+	+	−	−	+	−
Stores	Ratio	no	+	+	+	+	+	−	−	+	−
Variable Definitions	Ratio	no	+	−	+	+	+	+	−	+	−
Avarage path length	Ratio	no	+	−[a]	+	+	+	+	+	+	−
DU(G)	Ratio	no	+	−	+	+	+	+	+	+	+
LOC	Ratio	no	+	+	+	+	+	−	−	+	−
Maximum path length	Ratio	no	+	−[a]	+	+	+	−	+	+	−
NPATH	Ratio	no	+	−	+	+	+	−	+	+	+
Oviedo's Data Flow Comp.	Ratio	no	+	−	+	+	−	+	+	+	+
Cumulative Nesting Depth	Ratio	no	+	−	+	+	+	−	+	+	−
Maximum Nesting Depth	Ratio	no	+	−	+	+	+	−	+	+	−
Average Nesting Depth	Ratio	no	+	−	+	+	−	−	+	+	−
Halstead length	Ratio	yes	+	+	+	+	+	−	−	+	−
Halstead vocabulary	Ratio	yes	+	−	+	+	+	+	−	+	−
Cyclomatic Complexity	Interval	no	+	−	+	+	+	−	−	+	−
Piwowarski	Interval	yes	+	−	+	+	+	−	+	+	−
Prather's mu measure	Ordinal	no	+	+	+	+	+	−	+	+	−
Scope Number	Ordinal	no	+	+	+	+	+	−	+	+	−
AICC	Ordinal	yes	+	−	+	+	−	+	−	+	[b]
SynC	Ordinal	yes	+	+	+	+	+	−	+	+	[b]
Gong and Schmidt	Ordinal	yes	+	−	+	+	+	−	+	+	[b]
Halstead Volume	Ordinal	yes	+	+	+	+	+	+	−	+	[b]
Scope Ratio	No Scale	yes	+	−	+	+	−	+	+	+	[b]
Halstead Difficulty	No Scale	yes	+	+	+	+	−	+	−	+	[b]
Halstead Effort	No Scale	yes	+	+	+	+	−	+	−	+	[b]
Halstead Level	No Scale	yes	+	−	+	+	−	+	−	+	[b]

[a] '+' when length of `switch`-statement considered to be dependent on number of cases.
[b] Addition not defined for non-interval scales.

Table 3.1: An overview of the different software measures discussed in this section, containing the classifications I have not chosen a specific subset for.

include LOC in my experimentation of metrics in Chapter 5, it cannot be expected to correlate well with hardware measures.

3.2.2 Halstead's Software Science

A few years before COCOMO, Halstead presented his Software Science measures in [44], mainly as a way to estimate the programming effort. Because LOC only represents the quantity of the code, Halstead tried to account for the variability of the code. Furthermore, his measures were independent of source outlining, because the measures are based on tokens, instead of lines. Halstead based his measures on the following primitive metrics:

- the number of unique operands (η_1)
- the number of unique operators (η_2)
- the total number of operands (N_1)
- the total number of operators (N_2)

The quantity of code is represented by N_1 and N_2, while the variability is represented by η_1 and η_2. Again, at first glance these measures seem unambiguous. However, several definitions of operands and operators have been used in the research community. As with LOC clear definitions of operands and operators are needed when using these measures. For my model, I assume operators are tokens that represent actions and indirectly the structure of the hardware, e.g. C-operators, compound statements, etc. Operands on the other hand represent the use of a data item, e.g. the total number of occurrences of variables, constants, types, etc. In order to characterize memory or register accesses, I separately determine the number of *loads*, i.e. the number of variables and dereference operations, and *stores*, i.e. the number of assignments and effect operators ('++','–'). Variable definitions do not infer actions, nor do they imply an instance of data use. Therefore, variable definitions will be discarded in my model when counting operators and operands. However, they will be counted in a separate metric *variable definitions*. A further definition of operands and operators is presented in [44].

3.2.2.1 Metrics derived by summation

Halstead composes several derived measures. First, he presents two measures derived by summation:

- *Vocabulary*

$$\eta = \eta_1 + \eta_2 \tag{3.12}$$

  This derived metric indicates the number of different tokens used in the program. Actually, this implies this metric can also be determined directly and therefore is not really a derived measure.

- *Length*

$$N = N_1 + N_2 \tag{3.13}$$

  The implementation length is defined by the total number of tokens in the program. Again, this metric is actually a primitive measure.

As mentioned these metrics are not derived metrics. Nevertheless, in [70], Munson makes a strong argument that in fact these two measures do not add new information.

3.2.2.2 Metrics derived by logarithms

A second set of derived measures is derived by application of the logarithm of the primitive measures:

- *Expected Length*

$$N' = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \tag{3.14}$$

  Halstead also defined another length equation N', which is defined in terms of the unique operators and operands. This second notion of length is called the *expected length* and seems to be validated by several experiments [44, 22]. It must be noted, however, that these experiments assumed specific datasets and programming languages. For example in [36], the need for a correction of the expected length for PASCAL programs is established. By adjusting the expected length to specific datasets, the general applicability of the metric is lost. In fact, one can question whether the metric was valid to begin with, because N' is a highly subjective value.

- *Volume*

$$V = N \log_2 \eta \tag{3.15}$$

  The volume metric is presented as an alternative size metric of a program. Because of the logarithm this metric is no longer a ratio scale.

Again Munson is critical about these "new" measures. According to Munson, Halstead did not give valid reasons for constructing these measures. In the end, one cannot add new information by combining old information.

3.2.2.3 Complex derived measures

The following set of measures are based on the potentially new parameters η_1' and η_2', which indicate the minimum number of operators/operands needed to implement the required functionality:

- *Program Level*

$$L = \frac{V'}{V} \tag{3.16}$$

$$L' = \frac{2\eta_2}{\eta_1 N_2} \tag{3.17}$$

  This measure indicates how close the program resembles the optimal implementation of the program. V' or *potential volume* indicates the minimal Volume of a program. Because determining the optimal volume is not trivial, Halstead provides an estimate.

- *Difficulty*

$$D = 1/L \approx \frac{\eta_1 N_2}{2\eta_2} \tag{3.18}$$

  This measure is the inverse of the program level.

- *Intelligence Content*

$$I = L'V = \frac{2\eta_2 N \log_2 n}{\eta_1 N_2} \tag{3.19}$$

  This derivation aims to indicate how much actual information the program contains. It is supposed to be highly correlated with the potential volume.

- *Effort or total number of elementary discriminations*

$$E = DV = \frac{V^2}{V'} \approx \frac{\eta_1 N_2 N \log_2 n}{2\eta_2} \tag{3.20}$$

  This measure is supposed to indicate the effort required to implement the program.

Regrettably, Munson shows that these optimal values are not possible to determine with either certainty or precision. Furthermore, none of these derived measures has been validated. The bottom line is that only Halstead's primitive measures are valid candidates for inclusion in my model.

3.2.3 Cyclomatic Complexity

While the previously mentioned metrics were based on the program text, McCabe proposed a complexity measure based on control graphs in [68] called *cyclomatic complexity* which is an application of the cyclomatic number or circuit rank from graph theory to software measurement. In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits. The following equation shows the relation between $v(G)$ and the number of edges e, the number of vertices n, and the number of connected components.

$$v(G) = e - n + p \tag{3.21}$$

Control graphs are not strongly connected by default. Therefore, McCabe assumes a back edge from the exit node to the entry node. There are p such edges and so, McCabe defines the cyclomatic complexity as in Equation (3.22). Furthermore, he provides a simplified version of his metric in case of structured programs defined in terms of the number of predicates in a function (Equation (3.23)). The definitions are as follows,

$$v(G_P) = e - n + 2p \tag{3.22}$$

$$v(G_P) = \pi + 1 \tag{3.23}$$

where G_P stands for the CFG of program P and π for the number of predicates in P. In the partitioning model I propose here, we assume only structured programs are employed. Therefore, I use the simplified definition of the cyclomatic complexity.

The main advantage of this metric is its sole focus on control structures, in other words adding or removing simple statements does not influence the complexity. Because of this, I expect the cyclomatic complexity to indicate the complexity of control circuits after synthesis. However, because this measure only focuses on control aspects of a function, it will probably only be useful in a model that contains data flow metrics as well. Furthermore, because the circuit rank is an actual property of graphs, this measure is not a derived measure, although a high correlation with n and e is to be expected.

3.2.4 Nesting level

Another aspect several researchers have associated with complexity is nesting level. Indeed, when building programs with deep nested structures, one would reckon them to be harder to comprehend and more errorprone than "flatter" programs. In the following, we discuss several metrics that are aimed to measure or account for the nesting level.

3.2.4.1 Maximum Nesting Depth

$$\begin{aligned} &1)\delta(I) = 1 \\ &2)\delta(F_1; ...; F_s) = max\{\delta(F_1), ..., \delta(F_s)\} \\ &3)\delta(F(F_1, ..., F_n)) = 1 + max\{F_1, ..., F_n\} \end{aligned} \tag{3.24}$$

The most basic form to incorporate nesting level in a metric is by counting the maximum nesting level in a function. Zuse's depth metric, as presented in [37, 80], is one such metric. It is defined as a recursive function on S-graphs (see [37] for details) and the lowest level of nesting is assumed to have a depth of one. This metric is a real property of code and is not derived from other primitive measures.

3.2.4.2 Piwowarski

$$\texttt{PIWO}(G) = V'(G) + \sum_{i=0}^{i=\pi} \texttt{depth}_\texttt{i} \tag{3.25}$$

In [78] Piwowarski suggests to extend the cyclomatic complexity with nesting level information. For this purpose, he first alters how predicates of `switch` statements are counted. Instead of counting a `switch` statement as its equivalent $n-1$ `if` statements, Piwowarski counts such a statement as just one predicate. On top of this change, he adds the nesting depth of each predicate to his metric, where the nesting level of the lowest level is assumed to be zero. Piwowarski showed that several functions that had different qualitative notions of complexity were impossible to discern by the cyclomatic complexity, while his proposed measure could discriminate between them.

3.2.4.3 Gong and Schmidt

$$\epsilon_i = 1 - \frac{1}{\texttt{depth}_\texttt{i}}$$
$$\epsilon = (\sum_{i=0}^{i=\pi} \epsilon_i)/\pi$$
$$c(G) = v(G) + \epsilon \tag{3.26}$$

Gong and Schmidt also extended the cyclomatic complexity with nesting level information. However, they took another approach then Piwowarski. Instead of adding the nesting level of each predicate, they use a degree of nesting (η). Furthermore, they use a graph theoretical notion of nesting based on postdomination, that differs from the notion of depth used in the previous nesting level metrics. A node A post-dominates another node B if all paths from A contain B. With that concept Gong and Schmidt define that the depth of a predicate is the number of predicates in the subgraph rooted at that predicate, containing all paths up until the earliest post-dominating node of that predicate. Because the degree of nesting cannot be larger than one, addition and subtraction of this metric is not allowed. The Gong and Schmidt nesting metric, therefore, is no more than an ordinal metric.

3.2.4.4 Discussion

Because the behavior of nested control structures is interdependent, the complexity of nested structures can be larger than the sum of their respective complexities. The nesting level, therefore, represents that added complexity and can be a valuable asset to our model. However, the most nested structure of a function (indicated by the depth metric Equation (3.24)) might not be a good indicator for hardware area, because other less deeply nested structures are not accounted for. Piwowarski (Equation (3.25)) and Gong & Schmidt (Equation (3.26)) did add this aspect to their measures. However, they also base their metrics on the cyclomatic complexity, which makes the meaning of those measures unclear. Because of this, I propose the following extra measures as candidate measures for the quantitative model, which to the best of my knowledge have not yet been proposed elsewhere:

- *Cumulative Nesting Depth*

$$\texttt{CUMNEST} = \sum_{i=0}^{i=n_b} \texttt{depth}(\texttt{b}_\texttt{i}) \tag{3.27}$$

  By summing the nesting depth of all n_b basic blocks in a function, we obtain the cumulative nesting level. Where the maximum depth level only indicates the most "difficult" part of a routine, the cumulative nesting depth tries to represent the "difficulty" of the entire routine. This metric thus is a measure of the size of a function and therefore may be useful in area estimation.

- *Average Nesting Depth*

$$\texttt{AVGNEST} = \frac{\sum_{i=0}^{i=n_b} \texttt{depth}(\texttt{b}_\texttt{i})}{n_b} \tag{3.28}$$

With a notion of cumulative nesting depth we can define the average nesting depth as in Equation (3.28). Instead of capturing a notion of total difficulty of a routine, this metric indicates the average difficulty of any nested structure. Because nested structures, like loops, have the potential to be parallelized in hardware, the average nesting depth might be useful in speed-up estimation.

- *Loop Complexity*

$$\begin{aligned} &1)\texttt{LOOPCOM}(F(\emptyset)) && =1 \\ &2)\texttt{LOOPCOM}(F_1;\cdots;F_n) && =\sum_{i=1}^{n}\texttt{LOOPCOM}(F_i) \\ &3)\texttt{LOOPCOM}(F(F_1,\cdots,F_n)) && =1.1\texttt{LOOPCOM}(F_1;\cdots;F_n) \end{aligned} \tag{3.29}$$

This new complexity metric captures the fact that nested loops often loop over the same code and would therefore represent less increase in hardware as consecutive loops, that introduce new code apart from the previous loops. The recursive definition of the Loop Complexity (Equation (3.29)) is therefore multiplicative for nesting and additive for sequencing of loops, where 1) represents a loop that doesn't contain another loop, 2) represents consecutive loops, and 3) represents nested loops.

3.2.5 Scope Number/Ratio

Gong and Schmidt used the notion of postdomination to define the depth of a predicate. The same concept is employed by Harrison and Magel in [46] to define the scope number and the scope ratio. They define the complexity of a node (c_i) to be one, except for the complexity of predicate nodes. The latter is defined as the sum of the complexities of all nodes in the subgraph rooted at the predicate containing all paths up until, but not including, the earliest post-dominating node of that predicate. The scope number S_N is the sum of the complexities of all nodes and the scope ratio S_R is the scope number divided by the number of nodes N:

$$S_N = \sum_i c_i \tag{3.30}$$

$$S_R = \frac{S_N}{N} \tag{3.31}$$

While the algorithm to determine the scope number is clear, the exact meaning of this number is not. However, it is clear that adding more statements in sequence, as

well as nesting statements, increases the complexity. Furthermore, nesting implies a larger increase of the scope number than sequencing. Harrison and Magel do point out, however, that the scope number can be misleading, because the assumption that non-predicate nodes have a complexity of one is not justified. The main reason for using the scope ratio is to compensate for this problem.

Because the meaning of the scope number is unclear, its applicability to hardware estimation is uncertain. However, because the scope number grows when statements are added in sequence and in case of nesting grows superlinearly, it might be an indicator for hardware area. The scope ratio, or the scope number per node, is even less clear. However, we can say that to some extent it indicates the proportion of nodes that are control nodes. Nevertheless, it is not clear how my model can use the scope ratio for hardware estimation. Therefore, I only incorporate the scope number in my quantitative model.

3.2.6 Average Information Content Classification (AICC)

$$\texttt{AICC} = -\sum_{i=1}^{\eta_1} \frac{f_i}{N_1} \log_2 \frac{f_i}{N_1} \tag{3.32}$$

In [45], Harrison proposes that there is a relation between program complexity and the language entropy. More specifically, the program complexity is inversely proportional to the average information content of its operators. The equation he derived (Equation (3.32)) is based on the the following equation from information theory for the average number of bits per symbol required for an alphabet of symbols: $s_1, s_2, \cdots, s_q$,

$$H = -\sum_{i=1}^{q} p_i \log_2 p_i \tag{3.33}$$

where q is the number of symbols in the alphabet and p_i is the chance of s_i occurring in a text. Harrison's Average Information Content Classification (AICC) uses the operators as an alphabet and the chance of a character occurring in the program text determined a posteriori as the quotient of the number of occurrences of that operator (f_i) and the number of occurrences of all operators (N_1). Harrison points out that this measure is only an ordinal measure.

Although, [45] includes results showing strong correlation between the AICC and the average error span, we need to determine if the AICC can be useful in a hardware software partitioning model. If we consider Equation (3.32) and assume every operator has the same frequency $f = N_1/\eta_1$, then the value of the AICC will be as follows:

$$\texttt{AICC} = -\sum_{i=1}^{\eta_1} \frac{f_i}{N_1} \log_2 \frac{f_i}{N_1} = -\sum_{i=1}^{\eta_1} \frac{N_1/\eta_1}{N_1} \log_2 \frac{N_1/\eta_1}{N_1} = -\eta_1 \cdot \frac{1}{\eta_1} \log_2 \frac{1}{\eta_1} = \log_2 \eta_1 \tag{3.34}$$

We can now see that the AICC will increase when the number of unique operators increases. Although the assumption that every operator has the same number of occurrences in the code is not realistic, this derivation does imply the AICC represents the diversity of the operators in use by a program. Low AICC values could thus mean that hardware allocated for operators is more likely to be reused.

```
1 int value = 10;
2 if( value > 0 ) {
3     // this is always executed
4 } else {
5     // this is never executed
6 }
```

Algorithm 3.1: `if`-statement with only one possible path.

3.2.7 Path Measures

According to several researchers software complexity corresponds to the number of possible paths through a program. It is indeed difficult to keep track of many paths through a program during implementation, as well as during testing and maintenance. Many researchers have used path counts as an indicator of complexity. Notice that the cyclomatic complexity also counts paths, i.e. it counts the number of linearly independent paths. Apart from the number of paths in a program, also the length of paths can be valuable information. Finally, there have been some special measures related to paths. In the following sections, we discuss several path-related measures.

3.2.7.1 Actual number of paths

The most natural notion of the number of paths is the total number of possible paths in a CFG. Although this does not seem particularly difficult to determine, there are some serious problems with using this metric. For one, the definition is ambiguous, because it is not clear how much partial evaluation of the code should be performed. To illustrate this, consider the `if`-statement in Algorithm (3.1), which would normally have two control paths. When we partially evaluate the code, however, we can deduce that the `else`-part will never be executed. It is not clear, whether we should count one or two paths for this structure. Another problem with this metric surfaces when we consider loops. Because loops can have an indeterminate number of iterations at compile-time, the exact number of paths can not be represented by a finite number.

While intuitively the number of predicates, for example, correlates with hardware area, it is not clear how the number of paths will correlate with the area. This also holds true for latency and speed-up. Therefore, I do not make any predictions as to the usefulness of the path count in area and speed estimation here. For the results of the use of this metric see Chapter 5.

3.2.7.2 NPATH

In order to address the problems of the actual number of paths, Nejmeh proposed a static acyclic path count measure called NPATH[72]. NPATH is static in the sense that no partial evaluation is performed to discover paths that are never taken, and acyclic in the sense that loops are only considered to have a finite number of paths. NPATH is a recursive measure with specific expressions for each type of statement, as can be seen in

Statement	NPATH expression
`if`	$\texttt{NP}(\langle \texttt{if} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{condition} \rangle) + 1$
`if-else`	$\texttt{NP}(\langle \texttt{if} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{else} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{condition} \rangle)$
`while`	$\texttt{NP}(\langle \texttt{while} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{condition} \rangle) + 1$
`do-while`	$\texttt{NP}(\langle \texttt{do} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{condition} \rangle) + 1$
`for`	$\texttt{NP}(\langle \texttt{for} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{condition} \rangle) + \texttt{NP}(\langle \texttt{initialization} \rangle) + \texttt{NP}(\langle \texttt{iteration} \rangle) + 1$
`switch`	$\texttt{NP}(\langle \texttt{condition} \rangle) + \sum_{i=1}^{i=n} \texttt{NP}(\langle \texttt{case}_\texttt{i} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{default} - \texttt{range} \rangle)$
`?` operator	$\texttt{NP}(\langle \texttt{if} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{else} - \texttt{range} \rangle) + \texttt{NP}(\langle \texttt{condition} \rangle)$
`goto`	1
`break`	1
Expressions	Number of conditional operators in expression
`continue`	1
`return`	1
Sequential	1
Function call	1

Table 3.2: Detailed expressions for the NPATH complexity metric for statements in C presented in [72]

Table 3.2. The total NPATH complexity (NP) is defined as the product of the complexity of all statements in a program/function:

$$\texttt{NP}(f) = \prod_{\forall \texttt{statement}_\texttt{i}} \texttt{NP}(\texttt{statement}_\texttt{i}) \tag{3.35}$$

According to Nejmeh, the NPATH metric does not correlate well with LOC, Halstead's primitive metrics, or the cyclomatic complexity. This means the information contained in this metric measures separate aspects of code. Furthermore, Nejmeh shows that NPATH can be a useful metric during testing. As with the actual number of paths, mentioned above, I will not speculate on the relation of this metric with the area or speed of the function here.

3.2.7.3 Longest and Average Path Length

A measure that is particularly useful to in measuring the delay of functions or programs is the longest path length. This metric indicates the number of edges along the longest path in the CFG, in other words it stands for the worst case delay of the function. Nevertheless, when there are many possible paths, the worst case delay may not be a good measure of the average delay of a function. Instead, the average path length would be a more suitable candidate. Because the length of a path can be of indeterminable length at compile-time, I use the same notion of paths as the NPATH measure for these path length measures. In hardware estimation I expect the maximum path length to correspond to the critical path length, and therefore

indicative of the hardware latency and speed-up. The average path length on the other hand is an indication of the slack[38] of the tasks in the resulting circuit. Tasks with more slack could complete in time with less resources. Therefore, I expect the ratio between the longest and average path length to correlate with the area of the function.

Other path related measures have been proposed, but this is not the place for a detailed evaluation of path measures. Furthermore, the chance that new path measures have a high correlation with the measures mentioned here is high.

3.2.8 Prather's μ measure

$$\begin{aligned} \mu(P_1) = 1, \mu(D_1) = \mu(D_2) = 2^P \\ \mu(F_1; \cdots ; F_n) = \sum_{i=1}^{n} \mu(F_i) \\ \mu(F(F_1, \cdots , F_n)) = \mu(F) \max\{\mu(F_1), \cdots , \mu(F_n)\} \end{aligned} \tag{3.36}$$

As a part of his presentation of an axiomatic theory of software complexity measures [79], Prather presents a new testing metric, defined using this theory. The theory is based on defining metrics for three basic structured programming constructs: sequence, selection, and repetition. The testing metric μ presented in this paper is based on a specific test strategy (multiple condition test strategy). In Equation (3.36) the μ measure is defined for S^D graphs as presented in [37], with one modification: $\mu(D_i)$ is 2 instead of 2^P, where P is number of conditional operators in the condition. This modification accounts for the lazy evaluation used in the C programming language. P_1 stands for a non-control statement, D_1 stands for an if-statement, and D_2 stands for a loop. The behavior of the measure is additive for sequencing and multiplicative for nesting.

While this measure is designed to correlate with the number of tests required by the multiple condition test strategy, principal component analysis Table 4.1 suggests the measure relates to program length. Longer functions tend to be more complex and thus may result in more area. For this reason, Prather's μ metric will be incorporated in the quantitative model.

3.2.9 Basili-Hutchens complexity

$$\begin{aligned} 1)\texttt{SynC}(P) &= \begin{cases} 1 + \log_2(k+1) & \text{if } F \in S \\ 2(1 + \log_2(k+1)) & \text{otherwise} \end{cases} \\ 2)\texttt{SynC}(F_1; \cdots ; F_n) &= \sum_{i=1}^{n} \texttt{SynC}(F_i) \\ 3)\texttt{SynC}(F(F_1, \cdots , F_n)) &= \texttt{SynC}(F) + 1.1 \sum_{i=1}^{n} \texttt{SynC}(F_i) \end{aligned} \tag{3.37}$$

In [13], a hierarchical measure based on the cyclomatic complexity is presented by Basili and Hutchens (SynC). They have tried to devise a metric that discriminated between

structured and unstructured statements. They did not, however, define specifically how this distinction can be made. Therefore, any application of this metric should provide clear definitions of structured and unstructured statements. The definition of the Basili-Hutchens complexity for prime S-graphs uses the logarithm of the number of predicates, .i.e. the cyclomatic complexity. By using the logarithm, they assign lower complexity to `switch`-statements, compared to `if`-statements. To show this, assume a sequence of `if`-statements F_{if} and an equivalent `switch`-statement F_{switch} and consider:

$$\begin{aligned} \texttt{SynC}(F_{switch}) &= 1 + \log_2(k+1) \\ \texttt{SynC}(F_{if}) &= \sum_{i=1}^{k}(1 + \log_2(2)) = 2k \\ 1 + \log_2(k+1) \leq 2k \Leftrightarrow & \quad \texttt{SynC}(F_{switch}) \leq \texttt{SynC}(F_{if}) \end{aligned} \tag{3.38}$$

Applying this measure to a hardware/software partitioning model for an automatic toolchain will most probably discard the distinction between structured and unstructured statements, because practically no synthesis system to date supports these constructs. There are, however, some possibilities to include this metric. First, the model could support unstructured code. This would imply that manual changes are required, and thus, the model includes these changes in the estimate. While this would fit in the Delft Workbench, which is a semi-automatic - i.e. it supports manual changes and iterative improvement - toolchain, I assume structured code only. Second, another distinction between structured and unstructured code is employed. In this case, I would like to propose hardware resource requirements as the criterion. Particularly, I define structured statements as non-loop statements, and unstructured statements as loop statements. It is not clear in advance, however, whether this metric will yield good results or not.

3.2.10 Data Flow Measures

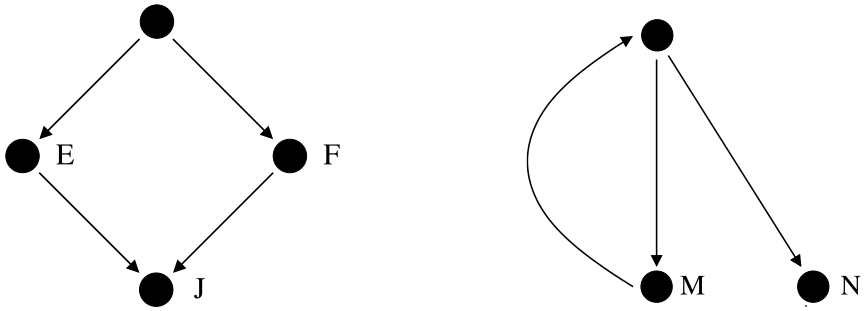
Where complexity in software has traditionally been associated with control flow, in hardware data flow has been more important. Therefore, I should give some attention to data flow complexity measures. Since there is no comparable alternative of software measurement theory and practice in hardware development yet and indeed control flow has been the main focus in software complexity research, there are far less data flow complexity measures available. In the following sections, I discuss two such metrics.

3.2.10.1 Tai's DU(g) measure

In [92], Tai details a data-flow measure based on control-flow graphs. This seemingly contradictory measure is based on assigning a pseudo-variable to the control structures in a control graph and then counts the so-called definition-use tuples, or d-u tuples for short. A 'definition' stands for the assignment of a value for the pseudo-variable and a 'use' stands for the use of the pseudo-variable. Specifically, Tai defines which tuples to count and how to assign the pseudo-variable definitions and uses. The algorithm for assigning definitions and uses can be seen in Algorithm (3.2). The specific definition of the measure Tai called DU(G) is the maximum number of d-u tuples in G, where:

```
1  G := a structured control graph beginning with a condition;
2
3  /* Allocate definition use pairs */
4  procedure ALLOCATE(G)
5      for each control structure in G do
6          case control structure of
7              IF    :
8                  assign 'use' to condition;
9                  assign 'definition' to block containing path
10                     with maximum number of conditions;
11             WHILE :
12                 assign 'use' to condition;
13                 assign 'definition' to loop-block;
14             OTHERS:
15                 /* Tai did not define other constructs,
16                    however switch-statements, for-loops,
17                    do-while loops, etc. can be defined
18                    analogously. */
19         end case
20     end for
21 end procedure
```

Algorithm 3.2: The 'definition' and 'use' allocation algorithm defined by Tai in [92] for the $DU(G)$ measure.



(a) `if`-construct (b) `while`-construct

Figure 3.1: The two control constructs for which $DU(G)$ is defined in [92]. The black dots are blocks, which in turn can contain other blocks.

1. the number of Gs output definitions is maximum,
2. blocks can have no more than one 'definition' and no 'use',
3. all conditions have one 'use' and no definition.

The requirements of this metric are satisfied by the allocation algorithm in Algorithm (3.2). Let m be the number of input definitions of the pseudo-variable to G, $\texttt{H}(G)$ be the maximum number of 'use's in any path in G, $\texttt{INSIDE}(G)$ the number of d-u tuples

in G generated by definitions inside G, and G be one of the cases in Figure 3.1, then:

$$\texttt{DU}(G) = \texttt{INSIDE}(G) + m\texttt{H}(G) \tag{3.39}$$

$$\texttt{INSIDE}(G) = \begin{cases} \min\{\texttt{H}(E), \texttt{H}(F)\} + \texttt{DU}(J) + \sum_{X \in (F,E,J)} \texttt{INSIDE}(X) & \text{if } G \equiv \texttt{if}\text{-case} \\ \texttt{H}(M) + \texttt{DU}(N) + \sum_{X \in (M,N)} \texttt{INSIDE}(X) & \text{if } G \equiv \texttt{while}\text{-case} \end{cases}$$

$$\texttt{H}(G) = \begin{cases} 1 + \max\{\texttt{H}(E), \texttt{H}(F)\} + H(J) & \text{if } G \equiv \texttt{if}\text{-case} \\ 1 + H(N) & \text{if } G \equiv \texttt{while}\text{-case} \end{cases}$$

In this equation, m stands for the number of input definitions of the graph G. When the graph G is the CFG of a function, m is equal to one. However, when $\texttt{DU}(G)$ is called recursively, $m = p + q + 1$ in case of an `if`-construct, where p and q are the number of `if`/`while`-constructs in the different branches of the `if`-construct. In case of a `while`-construct $m = r + 1$, where r is the number of `if`/`while`-constructs in the loop block.

According to Tai, the difference between $\texttt{DU}(G)$ and other control-graph metrics is significant. Basically, it captures the potential impact of the control flow on the data flow in some way. This metric may, therefore, relate to hardware area and/or speed, because any affect on the data flow can influence the final hardware design.

3.2.10.2 Oviedo's Data Flow Complexity

Where Tai introduced a pseudo-variable in CFGs, Oviedo defines a data flow complexity metric based on the actual variables in a program [75]. In order to explain how this measure is defined, I first have to explain some terms. A *definition* of a variable is either an assignment statement or function argument specification. A *block* is a sequence of statements that is always executed entirely and in the same order. A definition is *locally available* to a block, when it is defined in that block. A variable use is *locally exposed* when the variable was not (yet) locally available at the time. When a block b_i and a block b_j in a CFG are connected via a path and block b_i contains a locally available definition of variable v and there are no blocks along the path from b_i to b_j that have a locally available definition of v, then the definition of v in block b_i is said to *reach* block b_j. Now, let R_i be the set of definitions that reach block b_i and V_i the set of locally exposed variable uses in b_i, then Oviedo defines the data flow complexity of a block $DF(b_i)$ as

$$\texttt{DF}(b_i) = \sum_{j=1}^{||V_i||} |(r_i \in R_i | r_i \text{ defines } v_i)| \tag{3.40}$$

which is the number definition-use pairs that exist in block b_i and cross block boundaries. The total program P (or function) data flow complexity then surmounts to

$$\texttt{DF}(P) = \sum_{\forall b_i \in P} \texttt{DF}(b_i) \tag{3.41}$$

which sums all complexities of the block of program P. Where Tai's $DU(G)$ measure used a pseudo-variable that was closely tied to the CFG, Oviedo's $DF(G)$ is actually based on the data-flow in a program. As with Tai's metric, the $DF(G)$ may be useful for my model, because of it's basis on data flow. In contrast with Tai, Oviedo's measure has a stronger relation to the actual data flow of the program, and therefore might be a better choice for the quantitative model.

3.3 Conclusions

In this chapter, I introduced the concept of software metrics. We discuessed several classifications of these metrics. Furthermore, several metrics were specifically introduced and evaluated for inclusion in the quantitative model. Indeed, we can indicate several ways the metrics could correlate with hardware characteristics. In Chapter 5 we find which metrics do indeed show such a correlation.

Statistical and Quantitative Model Building

4

Now that the building blocks of my model have been presented, I establish how to build the model. I do this by establishing what a model is, how one should define a model, and how we determine the quality of the model. Several elements in this chapter are based on the work of Munson[70] and Lay[64]. Furthermore, the extensive information on linear regression theory available on the website of the course ST111 - Regression and analysis of variance, as offered by the Applied Mathematics department of the University of Southern Denmark [63], has been of great use for the contents of this chapter.

4.1 Models and Prediction Systems

According to Fenton[37], a model is *an abstract representation of an object*. Because this definition is rather general, Fenton focuses on a specific subset of models, *which are abstract representations of relationships between attributes of entities*. In my case, we seek a quantitative representation of the relationship between software measures and hardware measures. Such a quantitative model or mathematical model provides a system of equations that relate multiple measures to each other. The measures that the model aims to determine are called dependent variables and the variables that the model is based on are called independent measures. This naming scheme reflects the fact that the dependent variable is dependent on the other variables and the independent variables should not have a common source of variance, i.e. they should be independent of each other. In general, a quantitative model is of the form,

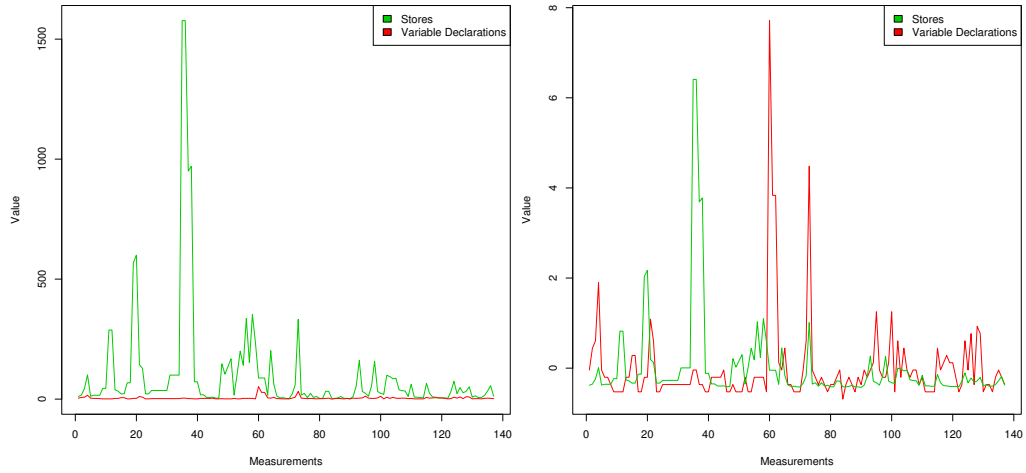
$$\mathbf{y} = F(\mathbf{x}) + \epsilon \tag{4.1}$$

where (y) is a vector of dependent measures, (x) is a vector of independent variables, F is some function of the independent variables, and ϵ is the error component or random error, because the model will probably not reflect reality perfectly. The random error is a random variable and therefore can not be exactly determined. Usually the random error is represented by the residual vector

$$\hat{\epsilon} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{y} - F(\mathbf{x}) \tag{4.2}$$

where $\hat{y}$ is the predicted value for y. There are many possible definitions of the function F, e.g. F might be a linear function, a nonlinear function, a neural network, etc. However, because there has been no prior quantitative model for hardware/software partitioning that was based on software metrics, I focused on linear models. A model will then be of the form,

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b} + \epsilon \tag{4.3}$$



(a) Original (b) Normalized (z-scores)

Figure 4.1: An example of metrics normalization. in the original version the metrics have values in different orders of magnitude and with different means, while in the normalized version both metrics are in the same range.

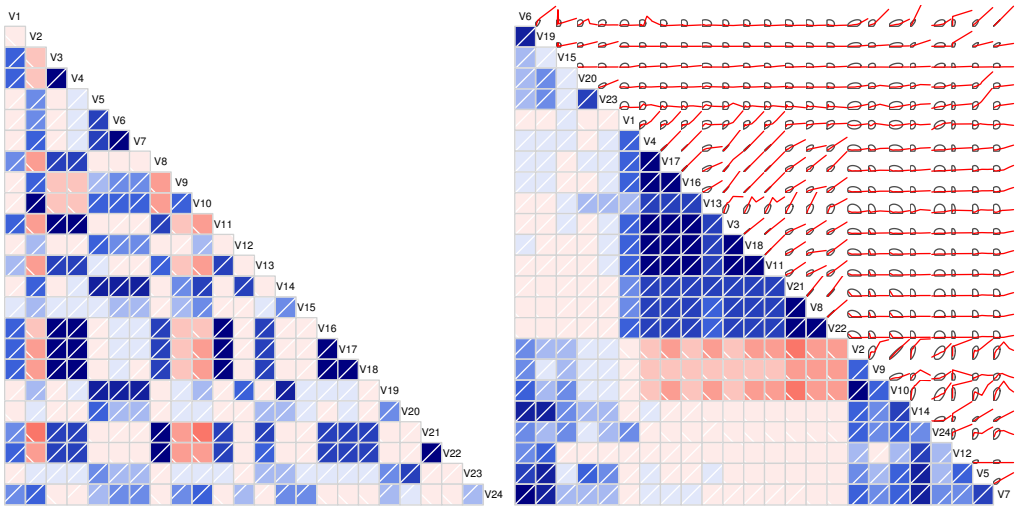
which is a set of linear equations for each dependent measure y_i based on the independent variables in $\mathbf{x}$, the matrix $\mathbf{A}$ holds the linear coefficients, and vector $\mathbf{b}$ holds the intercept values. Many statistical methods that can be applied on linear models, require the independent variables to be normally distributed. It seems plausible to assume that generally many functions are of comparable complexity and the more functions deviate from the mean complexity, the less common they are. Therefore I assume the data in this study to have a normal distribution.

However, as Fenton points out, when predicting values of dependent measures, a model alone will not suffice. For this purpose, he defines a so-called *prediction system*, which is a mathematical model accompanied by a set of procedures for determining the parameters of the model ($\mathbf{x}$), and interpreting the results. In fact, my quantitative model will be such a prediction system. More specifically, the descriptions of the software metrics in Chapter 3 comprise the procedures for my proposed prediction system.

4.2 Normalization of Metrics

Different metrics measure different aspects of the source code, and therefore the numbers we extract from a dataset will not directly be comparable. One of the obstacles in comparing these raw numbers is that they have different normal distributions. In order to compare the measures, therefore, I normalize the metrics to the standard normal distribution with mean μ zero and standard deviation σ one. This can be done with the formula,

$$\mathbf{Z} = \frac{\mathbf{V} - \mathtt{h}(m)\mu}{\mathtt{h}(\mathbf{n})\mathbf{\Sigma}} \tag{4.4}$$



(a) Unordered (b) Ordered

Figure 4.2: A graphical representation of the covariance matrix. The original version as well as a version with reordered rows and columns is presented.

where $\mathbf{V}$ is an $n \times m$ matrix of n measurements of m measures, μ is an $1 \times m$ vector of the means of the m measures, $\mathtt{h}(t)$ is a $t \times 1$ vector of ones, $\mathbf{\Sigma}$ is an $1 \times m$ vector of the standard deviations of the m measures, and the division is on a per element basis. This process is discussed in [70] and there the resulting values are called *z-scores*. With these transformed values, it is now possible to identify outlying values in the measurements and compare them to outlying values of other independent values.

4.3 Linear Interdependence of Metrics

As mentioned in my discussion of Halstead's Software Science metrics in Section 3.2.2, Munson[70] has some serious criticisms about derived metrics. Often there is no justification for the way several primitive metrics are composed and no empirical evidence to validate its use. Apart from this problem, Munson identifies the problem that metrics often measure the same thing. Take for example the measures LOC and statement count. Clearly, these metrics both measure the size of the source code and can be expected to have a large correlation. That this is also the case for other metrics can be seen in [74, 32, 93]. In statistics, this correlation between two measures can be characterized by the covariance

$$\mathtt{Cov}(x, y) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) \tag{4.5}$$

where x_i and y_i are measurement values from a set of n measurements. Or with the *Pearson product moment* for normalized measures

$$r_{xy} = \frac{1}{n-1} \sum_{i=1}^{n} z_x z_y \tag{4.6}$$

where $-1.0 \leq r_{xy} \leq 1.0$. The covariance indicates the amount of variance in measure X accounted for by a corresponding variance in measure Y. In Figure 4.2, the covariance between the different measures mentioned in Section 3.2 is presented.

4.4 Principal Component Analysis (PCA)

Because the software measures we use are apparently not independent, there will be a problem in employing them in a mathematical model, which normally requires these measures to be independent. Therefore, we need a way to extract a *linearly independent* or *orthogonal* set of measures. Statistics, luckily, provides such a procedure: Principal Component Analysis (PCA). PCA maps the measures into orthogonal attribute domains, i.e. groups of independent measures. Each principal component represents a different aspect of the objects we measure.

In order to describe PCA, assume the measures set $\mathbf{M}$ of n measures with a multivariate distribution μ and the square $n \times n$ matrix of covariances $\mathbf{\Sigma}$ and a corresponding matrix of covariances $\mathbf{S}$ determined from the set of measurements.

$$\begin{aligned} \mathbf{M} &= (m_1, \cdots, m_n) \\ \mu &= (\mu_1, \cdots, \mu_n) \\ \mathbf{\Sigma} &= (\sigma_{ij} = \mathtt{Cov}(m_i, m_j) | i, j = 1 \cdots n) \end{aligned} \tag{4.7}$$

Because Σ is a symmetrical matrix with real entries, i.e. $\sigma_{ij} \in \Re$, the spectral decomposition theorem states there exists a decomposition

$$\mathbf{S} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^{\mathbf{T}} \tag{4.8}$$

where $\mathbf{P}$ is the matrix with eigenvectors as columns and $\mathbf{\Lambda}$ is the matrix with just the eigenvalues along the diagonal. Note that the eigenvalues in $\mathbf{\Lambda}$ need to be sorted in decreasing order from left to right and $\mathbf{P}$ needs to change accordingly. In fact this decomposition provides us with an orthogonal set of (eigen)vectors and so, each eigenvalue stands for a linearly independent metric domain or domain metric d_i underlying the original set of metrics or raw metrics. The eigenvalues represent how much variance can be explained by a principal component. The sum of the eigenvalues equals the number of principal components. Therefore, the percentage of variance accounted for by a principal component i can be calculated as the ith eigenvalue divided by the number of principal components. Munson describes this procedure in [70]. The next step is to characterize the relationship between the raw metrics and the domain metrics using a product moment correlation as follows:

$$r_{m_i d_j} = \frac{P_{ij}\sqrt{\lambda_j}}{\sigma_i} \tag{4.9}$$

In this equation λ_j is the jth eigenvalue along the diagonal of $\mathbf{\Lambda}$ and σ_i is the standard deviation belonging to the raw metric m_i. The results of the PCA can be found in Table 4.1. The results of normal PCA are often unintelligible, because no objective for determining the orthogonal set is prescribed. Several additional rotations of the eigenvectors exist that remedy this problem. I employ the so-called *Variance Maximizing*

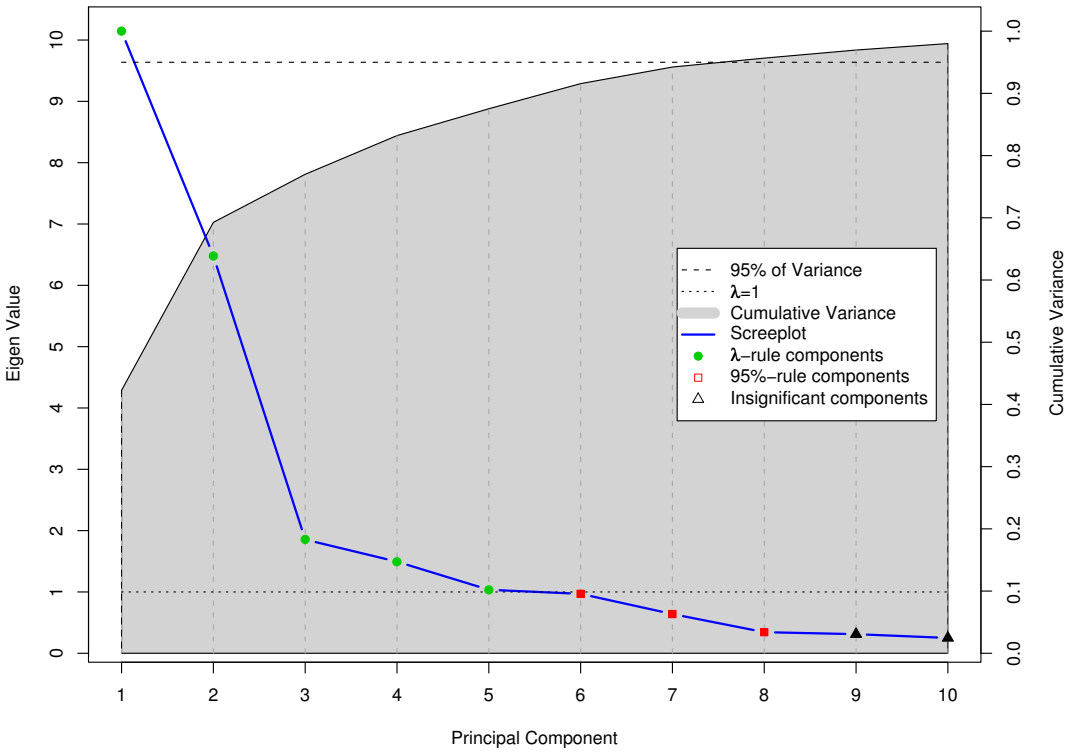


Figure 4.3: A combined graph of the eigenvalues (Screeplot) and the cumulative variances of the first twelve principal components of the set of software metrics I have considered

rotation (or VariMax), which maximizes the variance of the principal components. The results in Figure 4.1 are actually the rotated components.

According to Munson there are two main objectives of the PCA. First, to obtain a set of independent measures **D** and second, to reduce the number of distinct metrics. The second objective comes from the observation that there are more raw measures than actual sources of variation, because the raw measures were not independent.

In order to obtain the first objective, we need to derive a domain metric d_i for each principal component. For this purpose, we use the PCA of the z-scores of the raw measures. The PCA generates for each principal component a set of eigenvectors that can transform the measures to the orthogonal domain measures. The sum of the pairwise multiplication of the raw metric and their corresponding entry in the eigenvector, result in a new z-score representing the principal component of the eigenvector. The values of all domain metrics can thus be determined as folllows,

$$\mathbf{D_Z} = \mathbf{ZT} \tag{4.10}$$

where $\mathbf{Z}$ is the $n \times m$ matrix of the n measurements of the m raw measures, $\mathbf{T}$ is the $m \times p$ matrix of the eigenvectors corresponding to the selected p principal components, and $\mathbf{D_Z}$ is the $n \times p$ matrix containing the set of values of the p domain measures for the n measurements.

And for the second objective, Munson mentions there are several possibilities to select a subset of principal components. One such possibility is to choose only the components

Metric	Principal Components							
	PC 1: Length	PC 2: Control	PC 3: Nesting	PC 4: Diversity	PC 5: Data	PC 6: Volume	PC 7: Loop Complexity	PC 8: Paths
AICC	0.011	-0.008	-0.194	-0.821	-0.017	0.001	0.021	0.129
Avg. Nesting Depth	-0.043	-0.025	0.633	-0.009	0.009	-0.006	-0.047	0.061
Avg. Path Length	0.370	0.014	-0.005	-0.016	-0.010	-0.012	-0.020	-0.067
Basili-Hutchens	0.364	-0.058	0.015	0.007	-0.019	-0.001	0.049	0.033
Cum. Nesting Depth	0.008	-0.373	-0.017	-0.013	0.202	0.005	0.029	0.224
Cyclomatic(McCabe)	0.031	-0.411	0.028	-0.001	-0.030	0.013	0.015	-0.128
Gong and Schmidt	0.029	-0.410	0.040	-0.001	-0.030	0.013	0.016	-0.121
Loads	0.001	0.007	0.008	0.005	-0.010	0.569	0.013	-0.024
Loop Complexity	-0.024	-0.064	-0.034	-0.005	-0.011	-0.017	0.957	0.020
Max. Nesting Depth	-0.042	-0.042	0.605	-0.014	-0.001	-0.001	0.035	-0.004
Max. Path Length	0.370	0.026	-0.045	-0.012	-0.007	-0.001	-0.010	-0.015
NPATH	-0.046	-0.131	-0.077	-0.024	0.009	-0.020	-0.017	-0.900
Oviedo def-use pairs	0.212	0.278	0.028	0.050	0.368	0.094	0.122	-0.182
Piwowarski	0.023	-0.378	0.128	0.028	-0.022	0.020	0.139	0.032
Prather's mu	0.345	-0.018	0.213	0.049	0.015	-0.001	-0.014	-0.009
Scope Number	0.372	-0.020	-0.020	-0.002	-0.017	-0.006	-0.006	0.015
Statements	0.352	-0.049	-0.042	-0.010	0.027	0.005	-0.026	0.056
Stores	0.372	0.023	-0.028	-0.005	0.002	0.002	0.014	-0.001
Tai def-use pairs	0.010	-0.517	-0.170	0.007	0.046	-0.013	-0.193	0.106
Variable Declarations	-0.129	-0.006	-0.071	0.007	0.697	0.017	0.009	-0.006
Operands	0.025	-0.003	-0.011	-0.001	0.021	0.534	-0.005	0.009
Operators	-0.046	-0.037	-0.002	-0.009	-0.029	0.613	-0.015	0.025
Unique Operands	0.025	-0.057	0.063	-0.029	0.573	-0.060	-0.055	0.052
Unique Operators	0.012	0.031	0.284	-0.564	0.032	0.006	-0.020	-0.172
Eigen Value	10.145	6.479	1.854	1.494	1.036	0.969	0.640	0.344
Variance	0.423	0.270	0.077	0.062	0.043	0.040	0.027	0.014
Cum. Variance	0.423	0.693	0.770	0.832	0.875	0.916	0.942	0.957

Table 4.1: Results of Principal Components Analysis with VARIMAX Rotation of the 24 Software Metrics considered.

that have an eigenvalue higher than 1.0. Another possibility is to choose the minimal number of most principal components needed to account for at least a certain percentage of the variance, let's say 95%. Both methods are illustrated in Figure 4.3. It goes to far to discuss these methods in depth and therefore, I use Munson's choice for the former

method. Note that eigenvalues of 1.0 would be the result of complete random data.

4.5 Derived metrics and Ordinal Scales

There are some additional problems with using the measures from Section 3.2 in a linear model. First, there were several derived measures, which by definition are dependent on other (primitive) measures. When incorporating these measures in a quantitative model, no extra sources of information are added, i.e. it will not add a new principal component. Instead the quality of the PCA will deteriorate, because a subset of the information is duplicated and therefore overly represented. Therefore, I do not include measures in my model, that only contain measures that are already represented in the model.

Furthermore, some measures discussed earlier were only on an ordinal scale. This means only comparison operators are applicable to these measures and therefore, ideally they should not be used in linear regression nor in PCA. However, in practice, ordinal scales with more than a few possible values are often assumed to be interval- or ratio scales. The advantage of such an assumption is that normal quantitative analysis becomes possible. Therefore, we assume ordinal scales with more than a few possible values, like the scope number, can be used in normal quantitative analysis.

4.6 (Multiple) Linear Regression

When all independent and dependent variables have been determined for the set of candidate functions, we would like to fit the data with a linear model. As mentioned in section Section 4.1, such a model can be written as

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b} + \epsilon \tag{4.11}$$

, where $\mathbf{y}$ and $\mathbf{x}$ are available and the model parameters $\mathbf{A}$ and $\mathbf{b}$ are unknown. The model parameters can be determined, with linear regression. There are several linear regression approaches, however, the best known is the least squares method. Because I use this method of linear regression, I outline the process here. In principle, linear regression comes down to solving a set of linear equations. This linear system is traditionally written as,

$$\mathbf{y} = \mathbf{X}\beta + \epsilon \tag{4.12}$$

where, $\mathbf{y}$ is the *observation vector*, $\mathbf{X}$ the *design matrix*, β the *parameter vector*, and ϵ *the residual vector*. The observation vector and design matrix hold the observed values of the dependent and independent variables, respectively. Additionally, the design matrix has one column with only ones, which represents the intercept coefficient. The parameter vector holds the linear model parameters, also known as *regression coefficients*. Finally, the residual vector contains the residuals between the observed and predicted values.

However, there will most likely not be an exact solution β to these equations, because the model can't exactly represent reality. Therefore, an approximate solution $\hat{\beta}$ is needed.

The approximation should minimize the error and therefore minimize the length of the the residual vector. Hence the name least squares method. The approximate solution $\hat{\beta}$ can be obtained by solving the following normal equations:

$$\mathbf{X^T X}\hat{\beta} = \mathbf{X^T y} \tag{4.13}$$

The solution $\hat{\beta}$ then becomes:

$$\hat{\beta} = (\mathbf{X^T X})^{-1}\mathbf{X^T y} \tag{4.14}$$

Of course, this involves calculating the inverse of a large matrix, as well as three non-sparse matrix multiplications. Which makes this equation sensitive to an ill-conditioned design matrix, i.e. when small variations in the design matrix result in large changes in the least squares solution of β. This can be the case e.g. when the columns of the design matrix are linearly independent. In our case the columns consist of linearly independent principal components, therefore we use a numerically more stable method, using the *QR-factorization* of the design matrix, which would yield the equation,

$$\mathbf{R}\hat{\beta} = \mathbf{Q^T y} \tag{4.15}$$

where the solution can be obtained by backsubstitution.

However, so far the approximate solution $\hat{\beta}$ only applies to one dependent variable. For a model with multiple dependent variables, this process can be performed for each one of them. The solutions $\hat{\beta}^{\mathbf{0}}, \ldots, \hat{\beta}^{\mathbf{n}}$ then, correspond to the linear model as follows:

$$\begin{bmatrix} \hat{\beta}^{\mathbf{0}} \\ \vdots \\ \hat{\beta}^{\mathbf{n}} \end{bmatrix} = \begin{bmatrix} \hat{\beta}^{\mathbf{0}}_{\mathbf{0}} & \cdots & \hat{\beta}^{\mathbf{0}}_{\mathbf{m}} \\ \vdots & \ddots & \vdots \\ \hat{\beta}^{\mathbf{n}}_{\mathbf{0}} & \cdots & \hat{\beta}^{\mathbf{n}}_{\mathbf{m}} \end{bmatrix} = [\,\mathbf{b}\;\mathbf{A}\,] \tag{4.16}$$

4.7 Model Quality and Significance

After linear regression is performed, it is useful to determine the quality or significance of the resulting model. There are several techniques to evaluate model quality, some of which I use in this work. In the following sections, I list and discuss these techniques.

4.7.1 ANalysis Of VAriance (ANOVA)

ANalysis Of VAriance , or ANOVA, discriminates between different sources of variance and uses these to determine several model quality indicators. With these indicators one can reason about the quality and significance of the model in question. ANOVA is based on the following equation for one independent variable,

$$\sum_{\forall i}(y_i - \bar{y})^2 = \sum_{\forall i}(y_i - \hat{y}_i)^2 + \sum_{\forall i}(\hat{y}_i - \bar{y})^2 \Leftrightarrow SS_{TOT} = SS_E + SS_R \tag{4.17}$$

where y_i stands for an observation of a dependent variable, $\hat{y}_i$ for a prediction for that variable, and $\bar{y}$ for the average value of the observations of that variable. The first term

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio
Regression	SS_R	p	$MS_R = \frac{SS_R}{p}$	$F_c = \frac{MS_R}{MS_E}$
Error	SS_E	$n-p-1$	$MS_E = \frac{SS_E}{n-p-1}$	
Total	SS_{TOT}	$n-1$		

Table 4.2: All elements of the ANOVA for Multiple (Linear) Regression. n is the number of observations and p is the number of independent variables.

(SS_{TOT}) signifies the variance in the observations, the second term (SS_E) signifies the variance of the error of the model, and the final term (SS_R) signifies the variance in the predictions. These terms are also called the *sums of squares*. When considering multiple independent variables, the sums of squares are defined in matrix algebra as follows,

$$\begin{aligned} SS_R &= \beta^{\mathbf{T}}\mathbf{X}^{\mathbf{T}}\mathbf{Y} - \left(\tfrac{1}{n}\right)\mathbf{Y}^{\mathbf{T}}\mathbf{U}\mathbf{U}^{\mathbf{T}}\mathbf{Y} \\ SS_E &= \mathbf{e}^{\mathbf{T}}\mathbf{e} = \mathbf{Y}^{\mathbf{T}}\mathbf{Y} - \beta^{\mathbf{T}}\mathbf{X}^{\mathbf{T}}\mathbf{Y} \\ SS_{TOT} &= SS_E + SS_R = \mathbf{Y}^{\mathbf{T}}\mathbf{Y} - \left(\tfrac{1}{n}\right)\mathbf{Y}^{\mathbf{T}}\mathbf{U}\mathbf{U}^{\mathbf{T}}\mathbf{Y} \end{aligned} \tag{4.18}$$

where all variables are analogous to the regression equation $\mathbf{y} = \mathbf{X}\beta + \epsilon$ and U is an $n \times 1$ vector with one-valued elements.

Next to the sums of squares the ANOVA also uses the concept of *degrees of freedom* of each sum of squares. For SS_{TOT}, this concept is based on the deviations of the observational data from the observational mean $d_i = y_i - \bar{y}$. The sum of all deviations should equal zero, and therefore, when all but one deviations are known, the last value is fixed too. In other words there are $n-1$ observations that can have an arbitrary deviation. Therefore, we say that SS_{TOT} has $n-1$ degrees of freedom. For the other sums of squares the definitions can be found in Table 4.2. In this table also the *mean square* and *F-ratio* are mentioned. The F-ratio is actually related to the F-statistic with $(p, n-p-1)$ degrees of freedom. Using F_c we can test the following hypothesis,

$$H_0 : \forall i[i < p \Rightarrow \beta_i = 0] H_1 : \exists i[i < p \Rightarrow \beta_i \neq 0] \tag{4.19}$$

in other words, the null hypothesis states that there is no relation between the dependent and independent variables whatsoever and the alternative hypothesis assumes there is at least one independent variable that relates to the dependent variable. Using F_c we reject H_0 when

$$F_c > F(1-\alpha; p, n-p-1) \tag{4.20}$$

where α is the agreed upon significance level. There are two statistical concepts important for choosing a significance level: the importance of *type I errors* and *type II errors*. The first kind of error stands for rejecting the H_0 hypothesis, although it was true. And the second kind occurs when we accept the H_0 hypothesis, while actually H_1 is true. When choosing a lower significance level, the risk of a type II error increases. Because the hypotheses used here (Equation (4.19)) only tests whether the model explains something or nothing, I decided that it is more important to make sure we do not have a

type I error. Therefore, I assume a significance level of $\alpha = 0.01$ for the remainder of my thesis.

4.7.2 Performance Indicators

With the results of the ANOVA we can determine several traditional performance indicators. One such indicator is the the so called Coefficient of Determination (R^2), which indicates how much of the variance of the dependent variable can be explained by the regression equation.

$$R^2 = \frac{SS_R}{SS_{TOT}} \tag{4.21}$$

When R^2 is closer to one, the error is smaller and the regression equation explains more of the variance in the dependent variable.

Another indicator is the Mean Squared Error (MSE) of the model or in other words the expected deviation of the predictions from the mean.

$$\texttt{MSE}_{\texttt{model}} = E((y - \hat{y})^2) \tag{4.22}$$

This definition implies the mean squared error is a random variable and therefore, can only be estimated. Commonly, this is done by using the sample mean over all values used to fit the model or over a separate set of validation samples.

$$\widehat{\texttt{MSE}}_{\texttt{fit}} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.23}$$

$$\widehat{\texttt{MSE}}_{\texttt{predict}} = \frac{1}{m}\sum_{j=1}^{m}(y_j - \hat{y}_j)^2 \tag{4.24}$$

$$\tag{4.25}$$

In Equation 4.25, n is the number of samples used in the linear regression, y_i is the ith such sample, m is the number of samples in the validation set, and y_j is the jth such sample. Because $\texttt{MSE}_{\texttt{fit}}$ is determined using the values the model was fitted to, it doesn't actually tell anything about the predictive quality of the model. $\texttt{MSE}_{\texttt{predict}}$ is more suitable for this purpose.

The problem, however, is that often there aren't enough samples to split them into two separate sets. In order to evaluate model quality in that case, one can use the Predicted Residual Sum of Squares (PRESS) statistic. This statistic predicts the value for one instance using a model derived based on the remaining $n - 1$ observations. The statistic is calculated as follows,

$$\texttt{PRESS} = \sum_{i=1}^{n}(y_i - \hat{y}_{(i)})^2 \tag{4.26}$$

where y_i is the actual value of sample i, and $\hat{y}_i$ is the predicted value using a regression model based on the other $n - 1$ observations.

The disadvantage of the MSE and PRESS statistics is that they can only be used to compare different models. It is not really useful to calculate them for just one model. Often several combinations of the independent variables are considered for the final model. In that case, these statistics can help decide which model is the best.

In order to derive a more general notion of the error of a linear model, often the Rooted Mean Squared Error Percentage (RMSE%) is determined. It gives an indication of the average error of the model. This indicator is defined as,

$$\texttt{RMSE\%}_{\texttt{fit}} = \frac{n\sqrt{\widehat{\texttt{MSE}}_{\texttt{fit}}}}{\sum_{i=1}^{n} y_i} \texttt{RMSE\%}_{\texttt{PRESS}} = \frac{n\sqrt{\texttt{PRESS}}}{\sum_{i=1}^{n} y_i} \tag{4.27}$$

where y_i stands for the independent variable observations and n is number of these observations. In short, these indicators represent the expected error percentage for fitted data and predicted data.

4.7.3 Residual Analysis

Linear regression is performed under certain assumptions on the dependent and independent variables. These assumptions, which are called the *model assumptions*, must be satisfied for the model to be valid. For (multiple) linear regression the following assumptions apply:

1. Independence
   The dependent variables y_i are independent of each other. Of course, they are still dependent on the independent variables x_i.

2. Normality
   The dependent variables have a normal distribution $y_i \sim N(\mu, \sigma^2)$.

3. Homoscedasticity
   Every dependent variables has the same variance σ^2.

4. Linearity
   There is a linear relation between the independent and dependent variables.

Commonly these assumptions on model variables are converted to assumptions on random errors ϵ_i. The main reason for this translation is that the fourth assumption becomes somewhat simpler. The translated assumptions are:

1. The random errors are independent of each other.

2. The random errors have a normal distribution $\epsilon_i \sim N(\mu, \sigma^2)$.

3. The random errors have the same variance σ^2.

4. The random errors have a mean μ_i of zero.

In other words, the random errors should be independent and all have a distribution of $N(0, \sigma^2)$. Because the random errors are random variables, we use residuals when evaluating these assumptions. However, the residuals in the residual vector $\hat{\epsilon}$ are not suitable for this purpose, because these *raw residuals* have different normal distributions. Therefore, we need to convert the raw residuals to *standardized residuals* s_i as follows,

$$s_i = \frac{\hat{\epsilon}_i}{\sqrt{1 - h_{ii}}} \tag{4.28}$$

where h_{ii} are elements on the diagonal of the so called hat-matrix. This matrix transforms a vector of observations $\mathbf{y}$ into a vector of predictions $\hat{\mathbf{y}}$ and is defined as

$$\mathbf{H} = \mathbf{X}(\mathbf{X^T X})^{-1}\mathbf{X^T} \tag{4.29}$$

where $\mathbf{X}$ is the design matrix. The standardized residuals can be used to evaluate the four assumptions mentioned earlier. In the following subsections I discuss how one can evaluate these assumptions on a regression model.

4.7.3.1 Independence

The Independence assumption is usually evaluated subjectively. As mentioned in module 4 of [63] this assumption is largely dependent on how the data were obtained or how the experiment(s) were performed. For example, if the outcome of observation $i+1$ would be dependent on the outcome of observation i, the assumption would not hold. In our case, the observational data consists of metric values obtained from functions. These metric data are completely and solely dependent on the function specification and therefore, I consider this assumption valid in the remaining part of my thesis.

4.7.3.2 Normality

In order to check the normality of the errors, one can compare the distribution of the standard residuals with the normal distribution in a so called Q-Q Plot, which is a plot where the quantile values of two distributions are plotted. A *q-quantile* is a subset of the ordered set of observational data with $\frac{n}{q}$ elements, or an interval in a distribution where $\frac{100}{q}\%$ is represented. When the points in the plot (roughly) lie on the line $x = y$, then the distributions are (about) the same. When the plot does not imply that the errors are normally distributed, one can consider transforming the data into a normally distributed dataset. Common transformations are the log-transformation, square-root transformation, etc.

4.7.3.3 Homoscedasticity

The Homoscedasticity assumption requires the errors to have equal variances. Therefore when we plot all the predicted values for the samples against the standardized residuals, we would need to see a somewhat uniform spread of random valued points around the value zero. This plot is called the *residual plot*. In case the plot shows a pattern or trend in the residuals, there may be a problem with the experimentation or the underlying problem might not linear (assumption four). When there are a few outlying points, these points may not fit into this model.

4.7.3.4 Linearity

In order to investigate whether the model adheres to the fourth assumption, a simple approach would be to plot each independent variable against the dependent variable, and evaluate whether the plots can be considered linear. However, more precise evaluation needs to consider the relationship of this variable with respect to the other independent variables. In order to do that we can investigate a so called *partial residual plot*, which plots the predicted values for the samples against the partial residuals $p_{i,j}$, defined as

$$p_{i,j} = \beta_j x_{i,j} + r_i \tag{4.30}$$

where i is the number of the observation and j is the number of the independent variable. The linearity assumption is valid when the resulting graph resembles a straight line. In case the assumption does not hold, the independent variables could be transformed into a linear metric.

4.8 Conclusions

In this chapter, I presented the theory of the modeling procedure. I defined the necessary terminology, presented the necessary theoretical background for the principal component analysis and linear regression, and discussed how the performance and quality of the final model can be evaluated.

5 Model Building and Results

In this chapter, I present the actual quantitative model for hardware/software partitioning. This process involves several mathematical operations, which have been discussed in the previous section. In the following sections, I present the methodology and the results of the linear regression. Furthermore, I will discuss the presented results according to the theory presented in Chapter 4.

5.1 Methodology

In order to build the quantitative model for hardware/software partitioning, I took several steps. In this section, I elaborate these steps. Several times I refer to the *CD-ROM accompanying my thesis*. For more specific locations see Appendix B.

5.1.1 Acquire dataset

The first step in building the model was collecting a dataset of candidate functions that would function as a source of observational data. For this purpose, a collection of 135 C-language functions was made. The functions come from a broad range of application domains, like cryptography and multimedia. An overview of the dataset can be seen in Table 1.2. The broad range of origins of the candidate functions was needed, in order not to build a narrow model that would be applicable to a small set of applications only. An additional consequence of this approach is that there are many different algorithms and memory access patterns present in the dataset.

Because of the tools that were used to obtain the observational data (see Section 5.1.2), there were some restrictions on what language features of C could be present in the candidate functions. For example, at the time of writing DWARV does not support the `float` and `double`i-types, nor are `while`-loops and do-while-loops supported. Therefore, the functions were adapted in order to adhere to these and other restrictions. For a more complete description of the restrictions imposed by these tools, the reader is refered to [98].

The original code as well as the modified code can be found on the CD-ROM accompanying my thesis. See Appendix B

5.1.2 Generate Observational Data

Using the collected C-code, the observational data for the dependent and independent variables was obtained. However, for many metrics that were discussed in Chapter 3,

```
run
-ifn vhdl_sources.prj
-ifmt mixed
-ofn adpcm_encode_nopointer.ngc
-ofmt NGC
-p xc2vp30-7-ff896
-opt_mode Speed
-opt_level 1
-top adpcm_encode_nopointer
-iobuf NO
```

Figure 5.1: Example of the options used for the synthesis with the Xilinx ISE I.33 synthesis tool.

there was no tool to extract the observational values from the candidate functions. Therefore, the Elsa/Elkhound compiler frontend, from UC Berkeley [2, 69], was extended with code for extracting metrics from candidate functions. Using this modified tool, the metrical data was extracted. One of the main advantages of using software metrics is the small amount of time it takes to determine their value. The mentioned tool required on a 2.4GHz AMD Athlon64 only 7.5 seconds in total.

The Elkhound/Elsa source code and the modifications can be found on the CD-ROM accompanying my thesis. See Appendix B

In order to obtain area information from the high-level source code in the dataset, either manual or automatic transformation from C to VHDL was needed. Because of the limited amount of time and resources, two automatic C-to-VHDL translation tools were used to obtain VHDL code:

- Delft Workbench Automated Reconfigurable VHDL generator (DWARV)
  DWARV is a high-level C-to-VHDL compiler that was built as part of the Delft Workbench project at the TU Delft. It aims to support as much of the ANSI-C standard as possible and generates straightforward code. At the moment of writing it does not contain any optimization passes. Of the original number of roughly 140 kernels collected, 135 kernels were compiled to VHDL and 127 of those VHDL files were correctly synthesized.

- SPARK
  As mentioned in Section 2.3.2, SPARK is a C-to-VHDL high-level synthesis framework that aims to migrate software components (functions) to hardware. In contrast with DWARV it is much more restrictive and supports less features of the ANSI-C standard. In fact only 41 sources were compiled using the same original

dataset as DWARV. SPARK does, however, incorporate optimization and scheduling passes, which makes it a good case for comparison for our model.

For a more exhaustive comparison between these two VHDL generators, we refer the reader to [98]. The resulting VHDL codes were then synthesized using the *Xilinx ISE I.33* Synthesis tool using the options in Figure 5.1. In this work we only focus on area measures and therefore, we have not performed any simulations of the VHDL source codes. The area of the FPGA that was the target hardware platform of our model (*Virtex II Pro*), can be measured by counting the the following elements:

- Flip-Flops
  These are D-type Flip-Flops.
- Look-Up Table s (LUTs)
  These are 4-input LUTs.
- Slices
  A basic element consisting of 2 LUTs, 2 D-Type Flip-Flops, and some extra elements like multiplexers and carry chains.
- Multipliers
  These are 18bit by 18bit multipliers
- States
  This is the number of states in the Finite State Machine (FSM)

In addition, the synthesis tool gives an indication of the Period of the design, while generally this not an important aspect to know in the early stages of development, we have incorporated it in the modeling process.

The synthesis scripts can be found on the CD-ROM accompanying my thesis. See Appendix B

5.1.3 Perform Statistical Analysis

The observational data were then transformed into matrix representations and loaded into the open source statistical package *R* [3]. Using this package a PCA was performed. Then the observational data were transformed in principal component data. This dataset was then used to perform the regression analysis. Using several *R* language features and newly implemented functions the quality of the resulting linear model was evaluated.

The R commands and programs can be found on the CD-ROM accompanying my thesis. See Appendix B

Variable	Intercept	PC 1: Length	PC 2: Control	PC 3: Nesting	PC 4: Diversity	PC 5: Variables	PC 6: Volume	PC 7: Loops	PC 8: Paths
Slices	6.37e+03	8.84e+02	3.54e+02	-1.50e+01	-8.28e+02	5.64e+02	3.49e+03	-4.08e+00	-5.25e+02
Flip-Flops	7.13e+03	3.30e+02	1.56e+03	-1.11e+02	-3.64e+02	2.03e+03	5.48e+03	9.75e+02	-8.50e+02
LUTs	9.51e+03	1.09e+03	5.78e+02	-4.28e+01	-1.52e+03	8.75e+02	5.20e+03	-1.10e+02	-1.34e+03
Multipliers	5.94e+00	-9.35e-02	2.27e-01	6.18e-01	2.23e+00	9.54e-01	-9.00e-01	-1.21e+00	-2.14e+00
Period	6.70e+00	1.52e-01	-1.17e-01	2.65e-01	1.35e-01	-5.28e-02	2.51e-04	-2.37e-01	-1.94e-01
States	1.64e+02	6.23e+00	1.90e+01	5.84e+00	-6.69e+00	5.22e+01	1.06e+02	1.60e+01	5.23e+00

Table 5.1: Model Parameters for the DWARV models.

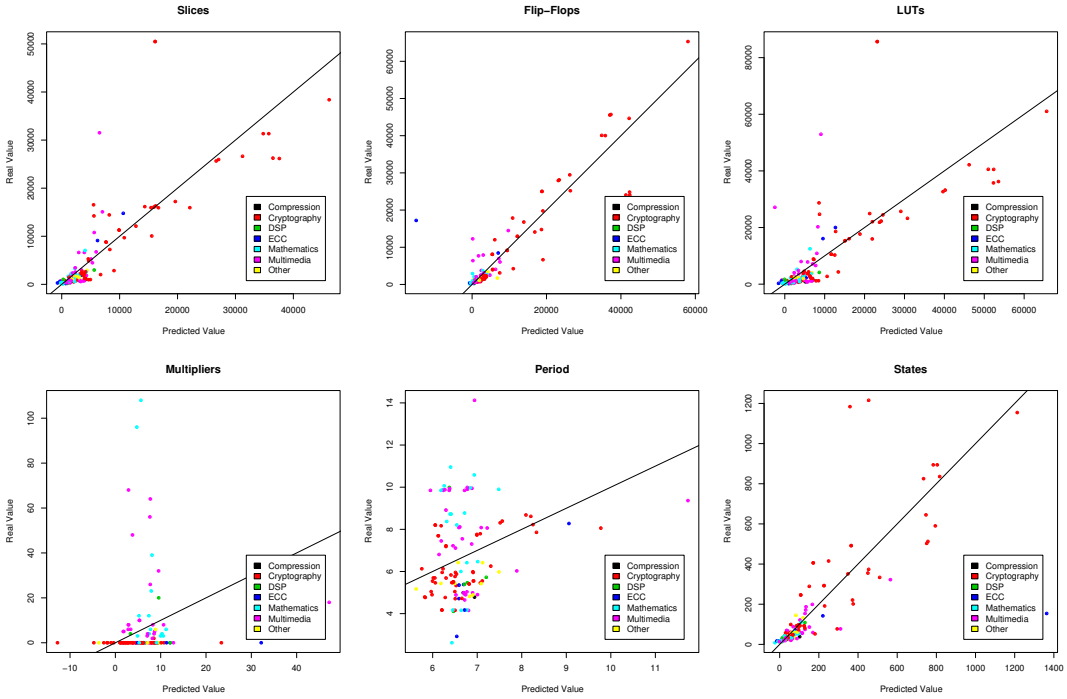


Figure 5.2: Model Predictions: DWARV

5.2 Results with DWARV

This section contains the results of the linear regression analysis. First, I present the actual model. Then, I evaluate the linear model assumptions as described in Section 4.7.3. And finally, I discuss the performance indicators for each model.

5.2.1 Predictions and model parameters

The models generated by the linear regression are listed in Table 5.1. Figure 5.2 depicts the predicted versus the actual values for the different models.

In Figure 5.2, we find a clear relation between the model predictions for the number of slices, flip-flops, LUTs, and States and their respective actual values. For the period and the number of multipliers, however, the model does not give useable predictions.

One obvious reason why the Multipliers-model does not perform well is the absence of a metric that represents the number of multiplications. Furthermore, the large number of zeroes in the multiplier observations make classical linear regression unreliable. The predictions for the period suggest there is no relation between the period and any of our principal components or software metrics.

The graphs for the number of slices and the number of LUTs are quite similar. Because every slice contains 2 LUTs, the number of slices seems to be dominated by the number of LUTs. There are also two outliers in the top-left corner of both plots. One outlier makes heavy use of expression lists, which our metrics implementation does not account for at the moment. The other makes heavy use of constant expressions. These make Halstead's [44] metrics unrepresentative of the actual complexity of the system.

Apart from LUTs, slices also contain flip-flops, carry chains, and multiplexers. However, the flip-flops graph does not resemble the slices graph as the LUTs graph does. It is not clear why this discrepancy exists. The flip-flop model appears to be the best of the six models.

Another observation we make is that some application domains occupy specific areas in the graphs. Especially, the Cryptography functions dominate the upper right part of the graphs for slices, flip-flops, LUTs, and states. A possible implication of this behavior is that different application domains may need different models. Or perhaps, the model can incorporate the application domain as a categorical variable.

5.2.2 Normality

Figure 5.3 depicts the Q-Q Plots for the six models. We can discern that all plots have so-called 'heavy tails', i.e. large discrepancies on the left and right hand side of the plot with respect to the normal line. This implies that the normality assumption does not apply. Nevertheless, the models do show predictive strength, as can be observed in Figure 5.2.

The distribution of the samples seem to be lognormal as opposed to normal. In the future, therefore, transformations should be applied to the dataset in order to satisfy the Normality assumption. More specifically, a `log` or `loglog`-transformation could transform the data for the dependent variables into a normally distributed dataset.

5.2.3 Homoscedasticity

The residual plots in Figure 5.4 show different types of behavior. For slices, flip-flops, LUTs and states we find smaller standard residuals closer to zero. Because the dataset contains more smaller functions, they are overrepresented during linear regression, which could explain this behavior.

Furthermore, these plots have several outliers which pose problems for the homoscedasticity assumption. Ideally, we would expect a homogeneous random distribution of points tightly around the 0-line. Apart from the outliers, we do find most standard residuals close to the 0-line. Therefore, I assume this assumption is (weakly) satisfied

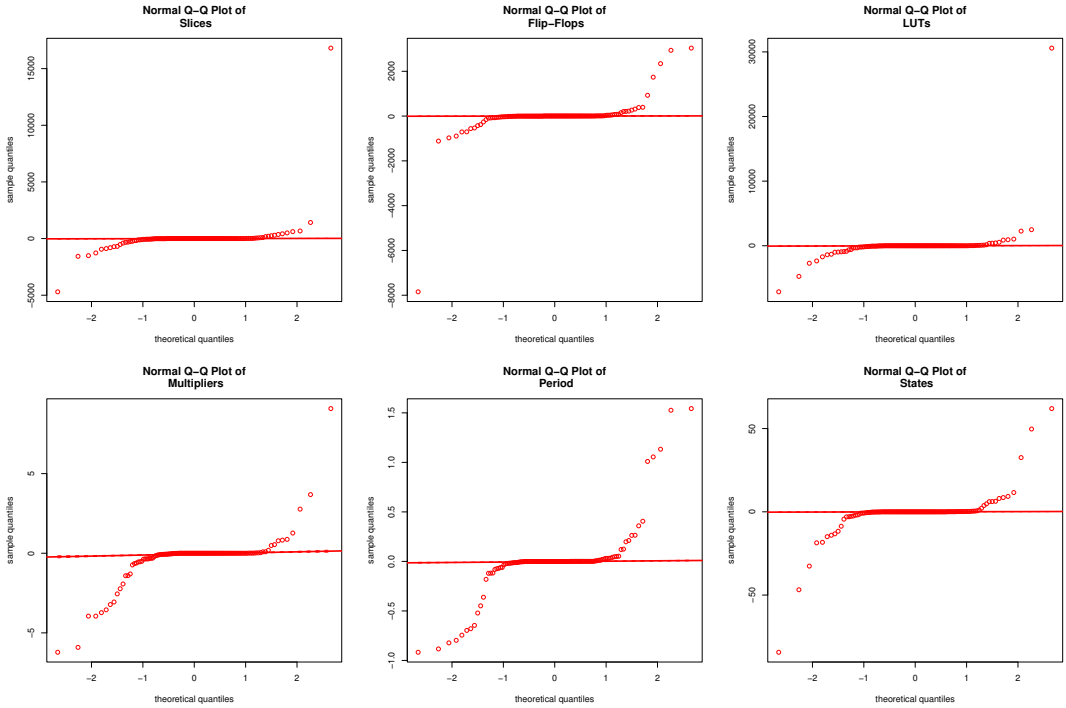


Figure 5.3: Q-Q Plots: DWARV

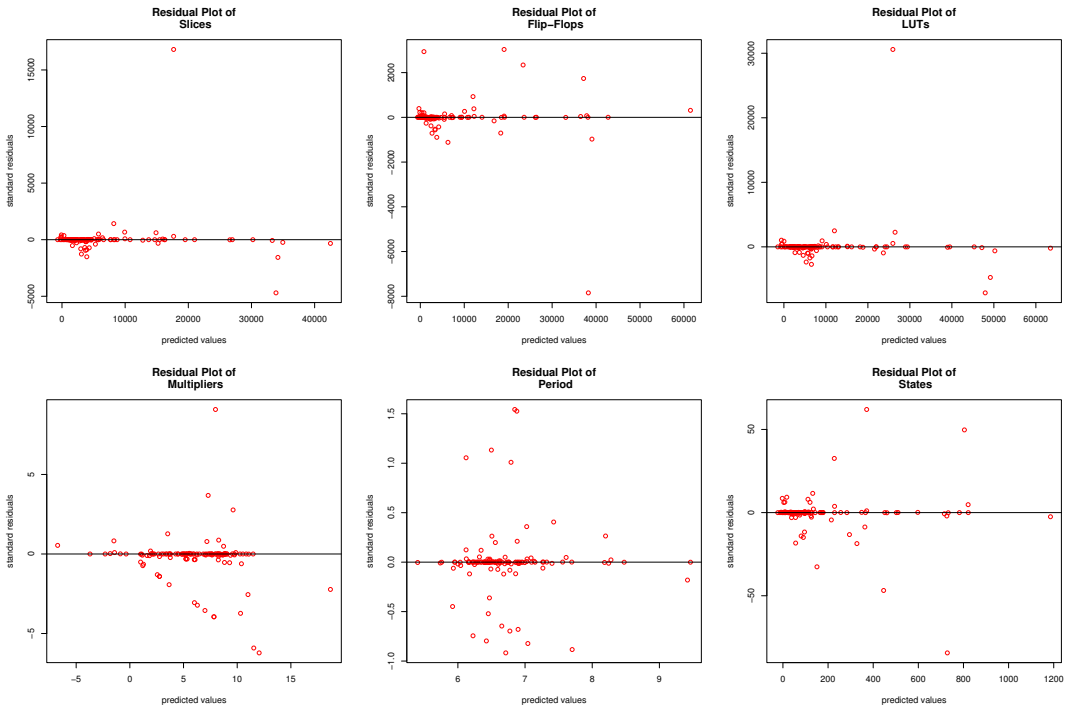


Figure 5.4: Residual Plot: DWARV

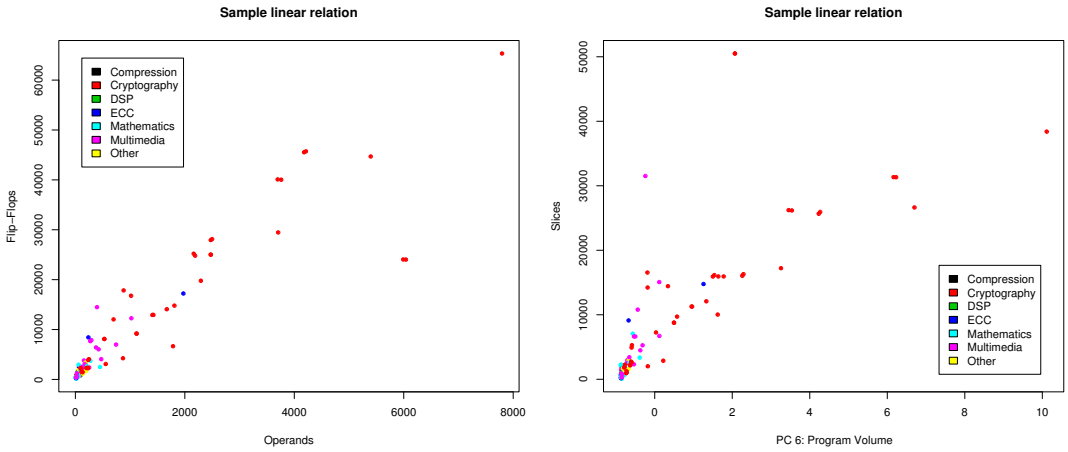


Figure 5.5: Metric Plot example showing a linear relation: DWARV

for the slices, flip-flops, LUTs and states models. In the future an outlier analysis can improve the quality of the model.

The residual plot for the number of multipliers clearly is not randomly distributed, i.e. there are linear patterns, and therefore the homoscedasticity assumption does not hold for this model. For the period, we see that the density of the plot becomes higher closer to the 0-line. This would imply the homoscedasticity assumption cannot hold for this model, as well.

5.2.4 Linearity

In order to test the linearity assumption, I created 48 partial residual plots. Because of the amount of graphs, these results can be observed in Section A.1. Generally, these graphs do not show a clear linear relation. Principal Component 1 (Program Length), 5 (Data Size), and especially 6 (Program Volume) do have a somewhat linear relation with the models for slices, flip-flops, LUTs, and states.

Nevertheless, there are not enough grounds to maintain the assumption of linearity for most of the principal components, based on the partial residual plots. However, we can observe good linear relations when we use the original metrics (see Figure 5.5). In the future, therefore, the use of raw metrics should be investigated.

Variable	R^2	RMSE%$_{\text{fit}}$	RMSE%$_{\text{PRESS}}$	p-value
Slices	0.717	85.0%	88.93%	< 2.2e-16
Flip-Flops	0.920	47.2%	69.2%	< 2.2e-16
LUTs	0.628	101.8%	108.4%	< 2.2e-16
Multipliers	0.047	287.5%	296.5%	0.6732
Period	0.095	29.8%	30.1%	0.1458
States	0.795	70.2%	101.7%	< 2.2e-16

Table 5.2: Performance indicators for the several dependent variables in the linear model for DWARV.

5.2.5 Model Performance

In Table 5.2, several performance indicators are listed (see Section 4.7.2 for a description). First, when we consider the coefficient of determination (R^2), we see that the variance in the data is not explained well by the models for the period and the number of multipliers. In contrast, we see that the other models perform relatively well for this aspect, especially the model for the number of flip-flops.

When we look at the errors of the different models, we see that the errors are relatively large. Although, the errors of the multipliers model is unacceptably large, the errors for the other models are sufficient for early prediction. Furthermore, when we also compare the graphs for slices and LUTs in Figure 5.2, we can observe that the error is larger for the smallest functions, i.e. the functions that have a low resource utilization. Because the smaller functions are larger in number, this influences the total expected error of the model.

Another observation we can make is that the expected error for the fitted data (RMSE%$_{\text{fit}}$) for the number of flip-flops and states is considerably lower than their respective expected error for the predicted data (RMSE%$_{\text{PRESS}}$). Apparently some of the data points that are omitted during the calculation of the PRESS statistic have significantly different values from the rest of the data points. Therefore, these data points are important for the model. In the future these samples should be investigated in more detail.

Considering the p-value (for complete ANOVA tables refer to Section A.2), we can conclude that the assumption that the models explain nothing at all is extremely small for all models except the multipliers and the period models, i.e. we reject the H_0 hypothesis for these models. Actually, it is quite plausible that the latter two explain nothing (67.3% and 14.6% chance respectively).

In short, the models for the number of slices, flip-flops, LUTs, and states can be considered useful within an early prediction environment.

Variable	Intercept	PC 1: Length	PC 2: Control	PC 3: Nesting	PC 4: Diversity	PC 5: Variables	PC 6: Volume	PC 7: Loops	PC 8: Paths
Slices	5.16e+03	-1.27e+03	-3.58e+02	-2.49e+03	-5.93e+03	-1.60e+03	6.12e+03	3.23e+03	-1.39e+03
Flip-Flops	3.02e+03	-7.34e+02	-6.24e+02	-1.02e+03	-2.32e+03	-6.13e+02	2.43e+03	7.40e+02	-3.28e+02
LUTs	7.48e+03	-1.72e+03	-1.04e+02	-3.80e+03	-9.26e+03	-2.62e+03	9.67e+03	5.64e+03	-2.43e+03
Period	5.93e+00	1.38e-01	3.48e-02	-1.51e-01	5.30e-01	-2.68e-01	6.05e-01	-7.19e-01	1.93e-01
Multipliers	3.17e+00	4.33e-01	1.65e+00	1.76e+00	1.93e+00	4.21e+00	8.88e-01	2.99e+00	5.82e-01
States	1.20e+01	5.73e-01	3.62e-01	-1.81e+00	-1.54e+00	-2.21e-01	3.17e+00	3.72e-01	-4.07e-01

Table 5.3: Model Parameters for the SPARK models.

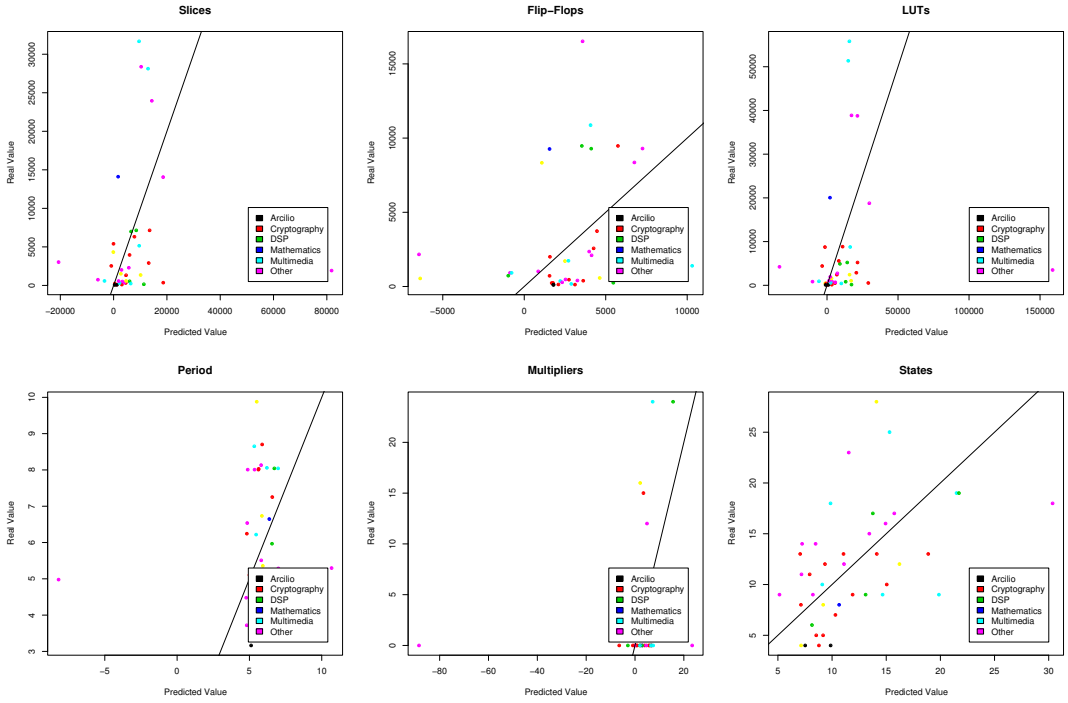


Figure 5.6: Model Predictions: SPARK

5.3 Results with SPARK

Because the proposed quantitative model is primarily targeted at the DWARV C-to-VHDL compiler, the same procedure has been followed using the SPARK VHDL generator. This allows us to see whether our modeling process can apply to other compilers as well. Because of the relatively poor performance of the model in case of SPARK, I do not evaluate the four linear modeling assumptions.

Variable	R^2	$\texttt{RMSE}_{fit}$	$\texttt{RMSE}_{\texttt{PRESS}}$	p-value
Slices	0.4767	131.3%	157.6%	4.041e-3
Flip-Flops	0.3342	122.8%	156.5%	7.752e-2
LUTs	0.4566	152.4%	184.9%	6.615e-3
Period	0.2182	29.1%	35.6%	0.3789
Multipliers	0.4572	160.5%	212.7%	6.521e-3
States	0.5482	36.3%	46.5%	5.585e-4

Table 5.4: Performance indicators for the several dependent variables in the linear model for SPARK.

5.3.1 Predictions and model parameters

As with the models for DWARV, the models for SPARK that were generated by the linear regression are listed in Table 5.3 and Figure A.7 depicts the predicted versus the actual values for the different models.

The main observation here is that the predictions seem to be particularly uncorrelated with the actual values. There is a large error in the predictions and we cannot find any linear relation between the predictions ($\hat{y_i}$) and the actual values (y_i), while in a good model these should be equal within a reasonable error margin.

5.3.2 Model Performance

When considering the performance indicators for the SPARK models in Table 5.4, we can again see the models do not perform particularly well. First, the coefficient of determination is rather low for all models, indicating there is much variance not explained by the model. Second, there are large errors for 4 of the models. The models for Period and States have somewhat lower error percentages. Finally, we find that the significance, i.e. *p-value* (for complete ANOVA tables refer to Section A.5), of the models is not as high as with the DWARV models, in fact H_0 cannot be rejected for the flip-flops model and the Period model for the chosen significance level.

5.3.3 Discussion

There are several possible reasons for the bad performance of this model. In the first place, the model is based on a subset (41 sources) of the original dataset (135 sources), because SPARK was not able to compile several sources in the dataset. The reduced amount of samples diminishes the quality of the model.

A second reason for this behavior are the optimizations and extensive scheduling performed by SPARK. DWARV only performs the necessary non-optimizating transformations. In contrast SPARK features dead code elimination, constant propagation, code motions, scheduling, etc. These transformations may change the function to a point where the software metrics don't capture its complexity anymore.

Furthermore, SPARK restricts its output to a specified amount of resources, which a designer can specify. In this work I used the original resource constraints shipped with

SPARK. Because SPARK attempts to schedule the design according to these constraints, the complexity of the design is changed.

5.4 Conclusions

In this section, I presented the steps of the modeling process and the results of that process. We saw that the DWARV models can make acceptable predictions for use in the early design phases. Furthermore, the modeling process for SPARK did not yield satisfactory results. In short, we saw encouraging preliminary results and clear points of future research and improvement.

Conclusions and Future Research

6

In this chapter we conclude the work presented in this thesis. Furthermore, we present several directions for future research and improvement of the quantitative model.

6.1 Conclusions

In this project I set out to build a preliminary quantitative prediction model for hardware/software partitioning. More specifically, I developed a linear regression model for prediction of several hardware characteristics based on software metrics. In order to establish the context of such a model, I first reviewed the literature on hardware/software partitioning, as well as hardware estimation. Subsequently, I presented a set of software metrics and discussed how these metrics could relate to hardware. Using a tool that could gather these metrics from C source code, I developed, I then collected the software metrics and hardware characteristics of 135 C kernels and performed a linear regression analysis, resulting in a quantitative model. The following list summarizes the main contributions and conclusions of my project:

- We have developed a preliminary *Quantitative Model* for early prediction in a hardware/software partitioning environment. Although the models for Slices, Flip-Flops, LUTs, and States have a relatively large error (69.2%-101.7%), we have seen that the predictions and the actual values show a clear linear relation, as well as that these models explain the larger part of the variance of the independent variables (62.5%-92.0%). In spite of the relatively large errors, our model can still be useful for two main reasons.

    1. The model makes it possible to predict hardware characteristics from the C-language level. There are not many other approaches that allow this and those mainly focus on C-dialects like SA-C, or use expensive synthesis algorithms as force-directed scheduling, allocation and binding, etc.
    2. Our model can be used for early design space pruning and design support. For example, the model can omit those functions that will probably not fit on the reconfigurable logic. Furthermore, the model may also identify functions that are near trivial in size. Such functions may not exploit a sufficient amount of parallelism. A designer can use the predictions to aggregate and segregate functions in order to obtain more optimal candidate functions.

- Although the models for Slices, Flip-Flops, LUTs, and States can provide useful predictions early on, of the four basic assumptions of classical linear regression only

the *Homoscedasticity* assumption is weakly satisfied. There are several transformations and modeling techniques that can be applied to remedy this problem (see Section 6.2). Despite these problems, we can reject the H_0 hypotheses for these models, meaning the models do not make predictions at random.

- The Multipliers and Period models do not perform satisfactory. Apparently, the chosen software metrics do not correlate with these measures. Both models account for almost none of the variance in the data (4.7%-9.5%) and are not sufficiently significant to reject the H_0 hypothesis. The relatively low prediction error for the Period model is the result of the small range of possible values for this measure.

- The model can derive estimates in a short time. The predictions for 135 C kernels spread over 81 files were made in 7.5 seconds. During the early stages of development this is an important feature, because of the iterative nature of design space exploration and the rapid succession of changes in the code. A designer can get an impression of the effects of his changes on hardware in a matter of moments.

- Different application domains correlate differently with hardware characteristics. For one, this makes the quality of the model less accurate, because it does not make this distinction. Furthermore, this would imply there are still software code aspects that are not represented by one of the software metrics.

- The linear model for SPARK performs significantly worse than the model for DWARV. Although, the optimizations, scheduling passes, and resource constraints of SPARK may be the cause of the bad performance, the set of observations was significantly smaller (41 kernels) than the one for DWARV (135 kernels). Therefore, we cannot claim this with sufficient certainty.

In short, we have seen that a quantitative model for hardware/software partitioning based on software metrics can be feasible. Furthermore, we can state that there is a clear correlation between aspects of C source code and hardware characteristics. This is an important fact for emerging toolchains for developing heterogeneous computing systems and RC systems, because it makes it possible to argue about hardware costs in the very early stages of development.

6.2 Future Research

Because of the importance of early predictions in hardware-software codesign, further research in this area is highly recommended. Specifically for the quantitative model presented here, we can point out several areas of possible improvement and extension. In the following, several of these areas will be identified and discussed.

- *Advanced modeling techniques*
  Apart from classical linear regression, several more advanced modeling techniques exist. For example, because the observational data for the number of multipliers contains many zeroes, classical linear regression does not yield a good prediction

model. When instead we use Generalized Linear Model (GLM), the independent variables do not need to be normally distributed. In case of the number of multipliers, for example, a Poisson-distribution might be more applicable.

Another advantage of GLM is the correct use of categorical and ordinal dependent variables. Instead of using ordinal scales as ratio scales, this technique can use so called *(ordered) factor columns* for nominal and ordinal scales.

- *Domain specific models*
  Because of the different behavior of different application domains, it may be necessary to build different models for each application domain. It can be expected that the linear regression will yield better results, when there are no groups of observations that behave differently.

- *Latency, Throughput, and Speed-Up*
  The models that were presented here all characterized in some way a kernel's resource consumption. However, other important aspects of a hardware design are its latency, throughput, and speed-up. In the future detailed simulations of the kernels in the dataset can provide the necessary data to build a prediction model for these measures as well.

- *Data transformations*
  In order to satisfy the linear modeling assumptions, the data could be transformed using transformations as the logarithm, the square root, etc. These transformations should not be applied blindly. Every transformation should be backed up by a theory on why such a transformation is necessary. Therefore, careful research in these transformations is needed.

- *Remove irrelevant and redundant metrics*
  Because of the redundancy of the metrics, the presented quantitative model was based on principal component scores. However, because some specific metrics correlate better with some hardware attributes than the principal components they contribute to, it is good to investigate the use of several specific metrics for the linear regression. Furthermore, several metrics that do not show any correlation might be dropped.

- *Increase the number of observations*
  The predictions presented here were made by subsequently isolating one observation, rebuilding the model, and estimating the isolated observation. The reason for this was the relatively small dataset (135 kernels). In order to use a more traditional scheme of preselecting a regression set and a validation set, a larger set of kernels is needed. Furthermore, several application domains are currently overrepresented. In the future, the dataset should be extended and balanced in order to make the dataset larger and more general.

- *Investigate the influence of optimizations and transformations*
  As the quantitative model for SPARK seemed to suggest, the model cannot currently predict the effect of optimizations on the resource consumption of the generated hardware. Furthermore, it can be expected that kernels that are transformed

into hardware by hand are even more difficult to predict. Research into the effect of automatic and manual optimizations and transformations on the resource consumption will be essential for estimations that remain applicable throughout the design process.

Bibliography

[1] *Delft workbench* [online, cited 19 December 2006], Available from: http://ce.et.tudelft.nl/DWB/.

[2] *Elkhound and elsa* [online, cited 19 December 2006], Available from: http://www.cs.berkeley.edu/~smcpeak/elkhound/.

[3] *The r project for statistical computing* [online, cited 18 April 2007], Available from: http://www.r-project.org/.

[4] *Roccc website* [online, cited 19 December 2006], Available from: http://www.cs.ucr.edu/~roccc/.

[5] *Spark: High-level synthesis using parallelizing compiler techniques* [online, cited 19 December 2006], Available from: http://mesl.ucsd.edu/spark/.

[6] David Andrews, Douglas Niehaus, and Peter Ashenden, *Programming models for hybrid cpu/fpga chips*, Computer **37** (2004), no. 1, 118–120, doi:10.1109/MC.2004.1260732.

[7] Peter M. Athanas and Harvey F. Silverman, *Processor reconfiguration through instruction-set metamorphosis*, IEEE Computer **26** (1993), no. 3, 11–18, doi:10.1109/2.204677.

[8] T. Bäck, U. Hammel, and H.-P. Schwefel, *Evolutionary computation: comments on the history and current state*, IEEE Transactions on Evolutionary Computation **1** (1997), no. 1, 3–17, doi:10.1109/4235.585888.

[9] John Backus, *Can programming be liberated from the von neumann style?: a functional style and its algebra of programs*, Commun. ACM **21** (1978), no. 8, 613–641, doi:10.1145/359576.359579.

[10] A. Balboni, W. Fornaciari, and D. Sciuto, *Partitioning and exploration strategies in the tosca co-design flow*, CODES '96: Proceedings of the 4th International Workshop on Hardware/Software Co-Design (Washington, DC, USA), IEEE Computer Society, 1996, p. 62.

[11] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, *A matlab compiler for distributed, heterogeneous, reconfigurable computing systems*, FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Washington, DC, USA), IEEE Computer Society, 2000, p. 39.

[12] Sudarshan Banerjee and Nikil Dutt, *Very fast simulated annealing for hw-sw partitioning*, Tech. Report UCI–CECS–04–18, University of California, Irvine, Irvine, CA, USA, June 2004.

[13] Victor R. Basili and David H. Hutchens, *An empirical study of a syntactic complexity family.*, IEEE Transactions on Software Engineering **9** (1983), no. 6, 664–672, Available from: http://www.cs.umd.edu/~basili/publications/journals/J17.pdf.

[14] Karthikeyan Bhasyam and Kia Bazargan, *Hw/sw codesign incorporating edge delays using dynamic programming*, DSD '03: Proceedings of the Euromicro Symposium on Digital Systems Design (Washington, DC, USA), IEEE Computer Society, 2003, p. 264.

[15] S. Bilavarn, G. Gogniat, and J. Philippe, *Area time power estimation for fpga based designs at a behavioral level*, In ICECS'2K, Kaslik, Lebanon, December 2000, Available from: http://citeseer.ist.psu.edu/bilavarn00area.html.

[16] Per Bjuréus, Mikael Millberg, and Axel Jantsch, *Fpga resource and timing estimation from matlab execution traces*, CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign (New York, NY, USA), ACM Press, 2002, pp. 31–36, doi:10.1145/774789.774797.

[17] Barry W. Boehm, *Software engineering economics*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[18] A. P. Bohm, B. Draper, W. Najjar, J. Hammes, R. rinker, M. Chawathe, and C. Ross, *One-step compilation of image processing applications to fpgas*, FCCM '01: Proceedings of the 9th Annual IEEE Symposium on field-Programmable Custom Computing Machines (Colorado State University, CO, USA), IEEE Computer Society, May 2001, pp. 209–218, doi:10.1109/FCCM.2001.32.

[19] Carlo Brandolese, *System-level performance estimation strategy for sw and hw*, ICCD '98: Proceedings of the International Conference on Computer Design (Washington, DC, USA), IEEE Computer Society, 1998, p. 48.

[20] Enrico Buracchini, *The software radio concept*, IEEE Communications Magazine **38** (2000), no. 9, 138–143.

[21] João M. P. Cardoso and Pedro C. Diniz, *Modeling loop unrolling: Approaches and open issues*, Lecture Notes in Computer Science, vol. 3133/2004, p. 224, Springer, July 2004, doi:10.1007/b98714.

[22] T Y Chen and S C Kwan, *An analysis of length equation using a dynamic approach*, SIGPLAN Not. **21** (1986), no. 4, 42–47, doi:10.1145/15095.15097.

[23] M. Cherkaskyy, *Theoretical fundamentals software/hardware algorithms*, TCSET '04: Proceedings of the International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (Lviv, Ukraine), February 2004, pp. 9–13, doi:10.1109/TCSET.2004.1365854.

[24] John C. Cherniavsky and Carl H. Smith, *On weyuker's axioms for software complexity measures*, IEEE Trans. Softw. Eng. **17** (1991), no. 6, 636–638, doi:10.1109/32.87287.

[25] Trishul M. Chilimbi, *Efficient representations and abstractions for quantifying and exploiting data reference locality*, PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 2001, pp. 191–202, doi:10.1145/378795.378840.

[26] Mike Cotterell and Bob Hughes, *Software project management*, third ed., McGraw-Hill Publishing Company, Berkshire, England, UK, 2002.

[27] M. Cummings and S. Haruyama, *Fpga in the software radio*, IEEE Communications Magazine **37** (1999), no. 2, 108–112, doi:10.1109/35.747258.

[28] Aravind Dasu and Sethuraman Panchanathan, *Reconfigurable media processing*, Parallel Comput. **28** (2002), no. 7-8, 1111–1139, doi:10.1016/S0167-8191(02)00107-2.

[29] Bharat P. Dave, *Crusade: hardware/software co-synthesis of dynamically reconfigurable heterogeneous real-time distributed embedded systems*, DATE '99: Proceedings of the conference on Design, automation and test in Europe (New York, NY, USA), ACM Press, 1999, p. 22, doi:10.1145/307418.307461.

[30] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha, *Cosyn: hardware-software co-synthesis of heterogeneous distributed embedded systems*, IEEE Trans. Very Large Scale Integr. Syst. **7** (1999), no. 1, 92–104, doi:10.1109/92.748204.

[31] P. Ellervee, A. Jantsch, J. Öberg, A. Hemani, and H. Tenhunen, *Exploring asic design space at system level with a neural networkestimator*, ASIC '94: Proceedings of the Seventh Annual IEEE International ASIC Conference and Exhibit (Campus IT University, Kista, Sweden), IEEE, September 1994, pp. 67–70, doi:10.1109/ASIC.1994.404607.

[32] James L. Elshoff, *Characteristic program complexity measures*, ICSE '84: Proceedings of the 7th international conference on Software engineering (Piscataway, NJ, USA), IEEE Press, 1984, pp. 288–293.

[33] R. Ernst and W. Ye, *Embedded program timing analysis based on path clustering and architecture classification*, ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design (Washington, DC, USA), IEEE Computer Society, 1997, pp. 598–604, doi:10.1145/266388.266562.

[34] Rolf Ernst, Jörg Henkel, and Thomas Benner, *Hardware-software cosynthesis for microcontrollers*, Readings in hardware/software co-design, Kluwer Academic Publishers, Norwell, MA, USA, 2002, pp. 18–29.

[35] M.A. Al Faruque, K. Karuri, S. Kowalewski, and R. Leupers, *Fine grained application profiling for guiding application specific instruction set processors(asips) design*, Master's thesis, Reinisch-Westfälische Hochshule, Aachen, Germany, 2004, Available from: `http://ces.univ-karlsruhe.de/~alfaruque/papers/thesis.pdf`.

[36] L. Felician and G. Zalateu, *Validating halstead's theory for pascal programs*, IEEE Trans. Softw. Eng. **15** (1989), no. 12, 1630–1632, doi:http://dx.doi.org/10.1109/32.58773.

[37] Norman E. Fenton, *Software metrics: A rigorous approach*, Chapman & Hall, Ltd., London, UK, UK, 1991.

[38] Brian Fields, Rastislav Bodík, and Mark D. Hill, *Slack: maximizing performance under technological constraints*, ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture (Washington, DC, USA), IEEE Computer Society, 2002, pp. 47–58.

[39] W. fornaciari, P. Gubian, D. Sciuto, and C. Silvano, *System-level power evaluation metrics*, ICISS '97: Proceedings of the Second Annual IEEE International Conference on Innovative Systems in Silicon (Milano, Italy), IEEE, October 1997, pp. 323–330, doi:10.1109/ICISS.1997.630275.

[40] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers, *Optimized generation of data-path from c codes for fpgas*, DATE '05: Proceedings of the conference on Design, Automation and Test in Europe (Washington, DC, USA), IEEE Computer Society, 2005, pp. 112–117, doi:http://dx.doi.org/10.1109/DATE.2005.234.

[41] Rajesh K. Gupta and Giovanni De Micheli, *Hardware-software cosynthesis for digital systems*, (2002), 5–17.

[42] R.K. Gupta and Giovanni De Micheli, *System-level synthesis using reprogrammable components*, EDAC '92: Proceedings of the Third European Conference on Design Automation (Center for Integrated Systems, Stanford University, CA, USA), March 1992, pp. 2–7, doi:10.1109/EDAC.1992.205881.

[43] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau, *Spark : A high-level synthesis framework for applying parallelizing compiler transformations*, vlsid **00** (2003), 461, doi:http://doi.ieeecomputersociety.org/10.1109/ICVD.2003.1183177.

[44] Maurice H. Halstead, *Elements of software science (operating and programming systems series)*, Elsevier Science Inc., New York, NY, USA, 1977.

[45] Warren Harrison, *An entropy-based measure of software complexity*, IEEE Trans. Softw. Eng. **18** (1992), no. 11, 1025–1029, doi:10.1109/32.177371.

[46] Warren Harrison and Kenneth Magel, *A topological analysis of the complexity of computer programs with less than three binary branches*, SIGPLAN Not. **16** (1981), no. 4, 51–63, doi:10.1145/988131.988137.

[47] Jörg Henkel and Yanbing Li, *Energy-conscious hw/sw-partitioning of embedded systems: a case study on an mpeg-2 encoder*, CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign (Washington, DC, USA), IEEE Computer Society, 1998, pp. 23–27.

[48] M. Hirzel and T. Chilimbi, *Bursty tracing: A framework for low-overhead temporal profiling*, In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), December 2001, Available from: http://citeseer.ist.psu.edu/hirzel01bursty.html.

[49] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, and Yu-Chin Hsu, *Data path allocation based on bipartite weighted matching*, DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation (New York, NY, USA), ACM Press, 1990, pp. 499–504, doi:10.1145/123186.123350.

[50] Axel Jantsch, Peeter Ellervee, Johny Öberg, and Ahmed Hemani, *A case study on hardware/software partitioning*, FCCM'94, Proceedings of the Workshop on FPGAs for Custom Computing Machines (Royal Institute of Technology, Stockholm, Sweden), IEEE Computer Society Press, 1994, pp. 111–118.

[51] Rolf L. Ernst Jörg Henkel, *High-level estimation techniques for usage in hardware/software co-design*, ASPDAC '98: Proceedings of the ASP-DAC'98. Asia and South Pacific Design Automation Conference (Princeton, NJ, USA), IEEE, February 1998, pp. 353–360, doi:10.1109/ASPDAC.1998.669500.

[52] Dennis Kafura and James Canning, *A validation of software metrics using many metrics and two resources*, ICSE '85: Proceedings of the 8th international conference on Software engineering (Los Alamitos, CA, USA), IEEE Computer Society Press, 1985, pp. 378–385.

[53] A. Kalavade and E. A. Lee, *The extended partitioning problem: hardware/software mapping and implementation-bin selection*, RSP '95: Proceedings of the Sixth IEEE International Workshop on Rapid System Prototyping (RSP'95) (Washington, DC, USA), IEEE Computer Society, 1995, p. 12.

[54] Asawaree Kalavade and Edward A. Lee, *A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem*, CODES '94: Proceedings of the 3rd international workshop on Hardware/software codesign (Los Alamitos, CA, USA), IEEE Computer Society Press, 1994, pp. 42–48, doi:10.1145/947193.

[55] Adam Kaplan, Philip Brisk, and Ryan Kastner, *Data communication estimation and reduction for reconfigurable systems*, DAC '03: Proceedings of the 40th conference on Design automation (New York, NY, USA), ACM Press, 2003, pp. 616–621, doi:10.1145/775832.775987.

[56] Kamal S. Khouri, Ganesh Lakshminarayana, and Niraj K. Jha, *Fast high-level power estimation for control-flow intensive design*, ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design (New York, NY, USA), ACM Press, 1998, pp. 299–304, doi:10.1145/280756.280941.

[57] Tae-Woo Kim and Hyunchul Shin, *Hardware cost estimation techniques for c-level description*, ICVC '99: Proceedings of the 6th International Conference on

VLSI and CAD (Ansan-City, Kyunggi-Do, South Korea), IEEE, 1999, pp. 85–88, doi:10.1109/ICVC.1999.820831.

[58] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, Science, Number 4598, 13 May 1983 **220, 4598** (1983), 671–680, Available from: http://citeseer.ist.psu.edu/kirkpatrick83optimization.html.

[59] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, *Data dependency size estimation for use in memory optimization*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **22** (2003), no. 7, 908–921, doi:10.1109/TCAD.2003.814257.

[60] Dhananjay Kulkarni, Walid A. Najjar, Robert Rinker, and Fadi J. Kurdahi, *Fast area estimation to support compiler optimizations in fpga-based reconfigurable systems*, FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Washington, DC, USA), IEEE Computer Society, 2002, p. 239.

[61] K. B. Lakshmanan, S. Jayaprakash, and P. K. Sinha, *Properties of control-flow complexity measures*, IEEE Trans. Softw. Eng. **17** (1991), no. 12, 1289–1295, doi:10.1109/32.106989.

[62] Paul Landman, *High-level power estimation*, ISLPED '96: Proceedings of the 1996 international symposium on Low power electronics and design (Piscataway, NJ, USA), IEEE Press, 1996, pp. 29–35.

[63] Pia Veldt Larsen, *St111 - regression and analysis of variance* [online, cited 19 December 2006], Available from: http://statmaster.sdu.dk/courses/st111/.

[64] David C. Lay, *Linear algebra and its applications*, second ed., Addison Wesley Longman, Inc., Boston, MA, USA, 2000.

[65] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood, *Hardware-software co-design of embedded reconfigurable architectures*, DAC '00: Proceedings of the 37th conference on Design automation (New York, NY, USA), ACM Press, 2000, pp. 507–512, doi:10.1145/337292.337559.

[66] J. A. Maestro, D. Mozos, and H. Mecha, *A macroscopic time and cost estimation model allowing task parallelism and hardware sharing for the codesign partitioning process*, DATE '98: Proceedings of the Design, Automation and Test in Europe Conference (Madrid, Spain), IEEE, February 1998, pp. 218–225, doi:10.1109/DATE.1998.655860.

[67] William H. Mangione-Smith, Brad Hutchings, David Andrews, André DeHon, Carl Ebeling, Reiner Hartenstein, Oskar Mencer, John Morris, Krishna Palem, Viktor K. Prasanna, and Henk A. E. Spaanenburg, *Seeking solutions in configurable computing*, Computer **30** (1997), no. 12, 38–43, doi:10.1109/2.642810.

[68] Thomas J. McCabe, *A complexity measure*, ICSE '76: Proceedings of the 2nd international conference on Software engineering (Los Alamitos, CA, USA), IEEE Computer Society Press, 1976, p. 407.

[69] Scott McPeak, *Elkhound: A fast, practical glr parser generator*, Tech. Report UCB/CSD–2–1214, University of California, Berkeley, Berkeley, CA, USA, december 2002.

[70] John C. Munson, *Software engineering measurement*, CRC Press, Inc., Boca Raton, FL, USA, 2002.

[71] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, *Accurate area and delay estimators for fpgas*, DATE '02: Proceedings of the conference on Design, automation and test in Europe (Washington, DC, USA), IEEE Computer Society, 2002, p. 862.

[72] Brian A. Nejmeh, *Npath: a measure of execution path complexity and its applications*, Commun. ACM **31** (1988), no. 2, 188–200, doi:10.1145/42372.42379.

[73] Mahadevamurty Nemani and Farid N. Najm, *High-level area and power estimation for vlsi circuits*, ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design (Washington, DC, USA), IEEE Computer Society, 1997, pp. 114–119.

[74] Michael B. O'Neal, *An empirical study of three common software complexity measures*, SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing (New York, NY, USA), ACM Press, 1993, pp. 203–207, doi:10.1145/162754.162867.

[75] E. I. Oviedo, *Control flow, data flow, and program complexity*, COMPSAC'80: Proceedings of the Fourth International Computer Software and Applications Conference, November 1980, pp. 146–152.

[76] Elena Moscu Panainte, *The molen polymorphic processor*, IEEE Trans. Comput. **53** (2004), no. 11, 1363–1375, Fellow-Stamatis Vassiliadis and Member-Stephan Wong and Member-Georgi Gaydadjiev and Member-Koen Bertels and Student Member-Georgi Kuzmanov, doi:10.1109/TC.2004.104.

[77] Zebu Peng and Krzysztof Kuchcinski, *An algorithm for partitioning of application specific systems*, Tech. Report R-94-01, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1994, Published in Proceedings of the European Conference on Design Automation EDAC'93, Paris, France, February 22-25, 1993, Available from: `http://informatix.ida.liu.se/publications/cgi-bin/tr-fetch.pl?r-94-01+ps`.

[78] Paul Piwowarski, *A nesting level complexity measure*, SIGPLAN Not. **17** (1982), no. 9, 44–50, doi:10.1145/947955.947960.

[79] R. E. Prather, *An Axiomatic Theory of Software Complexity Measure*, The Computer Journal **27** (1984), no. 4, 340–347, doi:10.1093/comjnl/27.4.340.

[80] Ronald E. Prather, *Design and analysis of hierarchical software metrics*, ACM Comput. Surv. **27** (1995), no. 4, 497–518, doi:10.1145/234782.234784.

[81] Yang Qu and J.-P. Soininen, *Estimating the utilization of embedded fpga co-processor*, DSD '03: Proceedings of the Euromicro Symposium on Digital System Design (VTT Electron., Oulu, Finland), IEEE Computer Society, September 2003, pp. 214–221, doi:10.1109/DSD.2003.1231929.

[82] D. Saha, A. Basu, and R. S. Mitra, *Hardware software partitioning using genetic algorithm*, VLSID '97: Proceedings of the Tenth International Conference on VLSI Design: VLSI in Multimedia Applications (Washington, DC, USA), IEEE Computer Society, 1997, p. 155.

[83] Luc Séméria and Giovanni De Micheli, *Spc: synthesis of pointers in c: application of pointer analysis to the behavioral synthesis from c*, ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design (New York, NY, USA), ACM Press, 1998, pp. 340–346, doi:10.1145/288548.289051.

[84] Li Shang and Niraj K. Jha, *Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable fpgas*, ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design (Washington, DC, USA), IEEE Computer Society, 2002, p. 345.

[85] Alok Sharma and Rajiv Jain, *Estimating architectural resources and performance for high-level synthesis applications*, DAC '93: Proceedings of the 30th international conference on Design automation (New York, NY, USA), ACM Press, 1993, pp. 355–360, doi:10.1145/157485.164929.

[86] U. Nagaraj Shenoy, Alok Choudhary, and Prithviraj Banerjee, *Symphany: A tool for automatic synthesis of parallel heterogeneous adaptive systems*, Tech. Report CPDC-TR-9903-002, Center for Parallel and Distributed Computing, Northwestern University, Evanston, IL, USA, March 1999, Available from: `http://www.ece.northwestern.edu/cpdc/TechReport/1999/03/CPDC-TR-9903-002.ps.gz`.

[87] S. Srikanteswara, M. Hosemann, J. H. Reed, and P.M. Athenas, *Design and implementation of a completely reconfigurable soft radio*, RAWCON '00: Proceedings of IEEE Radio and Wireless Conference (Mobile & Portable Radio Res. Group, Virginia Polytech. Inst. & State Univ., Blacksburg, VA, USA), IEEE, September 2000, pp. 7–11, doi:10.1109/RAWCON.2000.880945.

[88] S. Srikanteswara, R. C. Palat, J. H. Reed, and P. Athanas, *An overview of configurable computing machines for software radio handsets*, IEEE Communications Magazine **41** (2003), no. 7, 134–141, doi:10.1109/MCOM.2003.1215650.

[89] V. Srinivasan, S. Govindarajan, and R. Vemuri, *Fine-grained and coarse-grained behavioral partitioning witheffective utilization of memory and design space exploration formulti-fpga architectures*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **9** (2001), no. 1, 140–158, doi:10.1109/92.920829.

[90] Al Strelzoff, *Functional programming for reconfigurable computing.*, IPDPS '04: Proceedings of the 18th International Parallel and Distributed Processing Symposium (San Jose, CA, USA), IEEE Computer Society, April 2004, doi:10.1109/IPDPS.2004.1303138.

[91] Dinesh C. Suresh, Walid A. Najjar, Frank Vahid, Jason R. Villarreal, and Greg Stitt, *Profiling tools for hardware/software partitioning of embedded applications*, SIGPLAN Not. **38** (2003), no. 7, 189–198, doi:10.1145/780731.780759.

[92] Kuo-Chung Tai, *A program complexity metric based on data flow information in control graphs*, ICSE '84: Proceedings of the 7th international conference on Software engineering (Piscataway, NJ, USA), IEEE Press, 1984, pp. 239–248.

[93] George Triantafyllos, Stamatis Vassiliadis, and José G. Delgado-Frias, *Software metrics and microcode: a case study*, Journal of Software Maintenance **8** (1996), no. 3, 199–224, doi:10.1002/(SICI)1096-908X(199605)8:3<199::AID-SMR129>3.3.CO;2-E.

[94] Frank Vahid, *Procedure exlining: a transformation for improved system and behavioral synthesis*, ISSS '95: Proceedings of the 8th international symposium on System synthesis (New York, NY, USA), ACM Press, 1995, pp. 84–89, doi:10.1145/224486.224506.

[95] Frank Vahid and Daniel D. Gajski, *Incremental hardware estimation during hardware/software functional partitioning*, Readings in hardware/software co-design, Kluwer Academic Publishers, Norwell, MA, USA, 2002, pp. 516–521.

[96] Frank Vahid, Daniel D. Gajski, and Jie Gong, *A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning*, EURO-DAC '94: Proceedings of the conference on European design automation (Los Alamitos, CA, USA), IEEE Computer Society Press, 1994, pp. 214–219.

[97] Stamatis Vassiliadis, Georgi N. Gaydadjiev, Koen Bertels, and Elena Moscu Panainte, *The molen programming paradigm*, Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation (Delft, Netherlands), July 2003, pp. 1–10.

[98] Arcilio Jaime-Raul Virginia, *Comparative study of vhdl generators*, Master's thesis, Delft University of Technology, Delft, The Netherlands, May 2007.

[99] John von Neumann, *First draft of a report on the edvac*, Charles Babbage Institute Reprint Series for the History of Computing, vol. 12, MIT Press, 1987.

[100] Miljan Vuletic, Laura Pozzi, and Paolo Ienne, *Programming transparency and portable hardware interfacing: Towards general-purpose reconfigurable computing.*, ASAP '04: Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures, and Processors, IEEE Computer Society, September 2004, pp. 339–351, doi:10.1109/ASAP.2004.10028.

[101] E. J. Weyuker, *Evaluating software complexity measures*, IEEE Trans. Softw. Eng. **14** (1988), no. 9, 1357–1365, doi:10.1109/32.6178.

[102] Wayne Wolf, *Building the software radio*, Computer **38** (2005), no. 3, 87–89, doi:10.1109/MC.2005.82.

[103] Jianwen Zhu and S. Calman, *Context sensitive symbolic pointer analysis*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **24** (2005), no. 4, 516–531, doi:10.1109/TCAD.2005.844092.

[104] Horst Zuse, *Properties of software measures*, Software Quality Journal **1** (1992), no. 4, 225–260, doi:10.1007/BF01885772.

Detailed Results

A

This appendix contains detailed results for the models for DWARV and SPARK. For SPARK I also included the graphs for evaluating the modeling assumptions, i.e. the Q-Q plots, Residual plots, etc.

A.1 DWARV - Partial Residual Plots

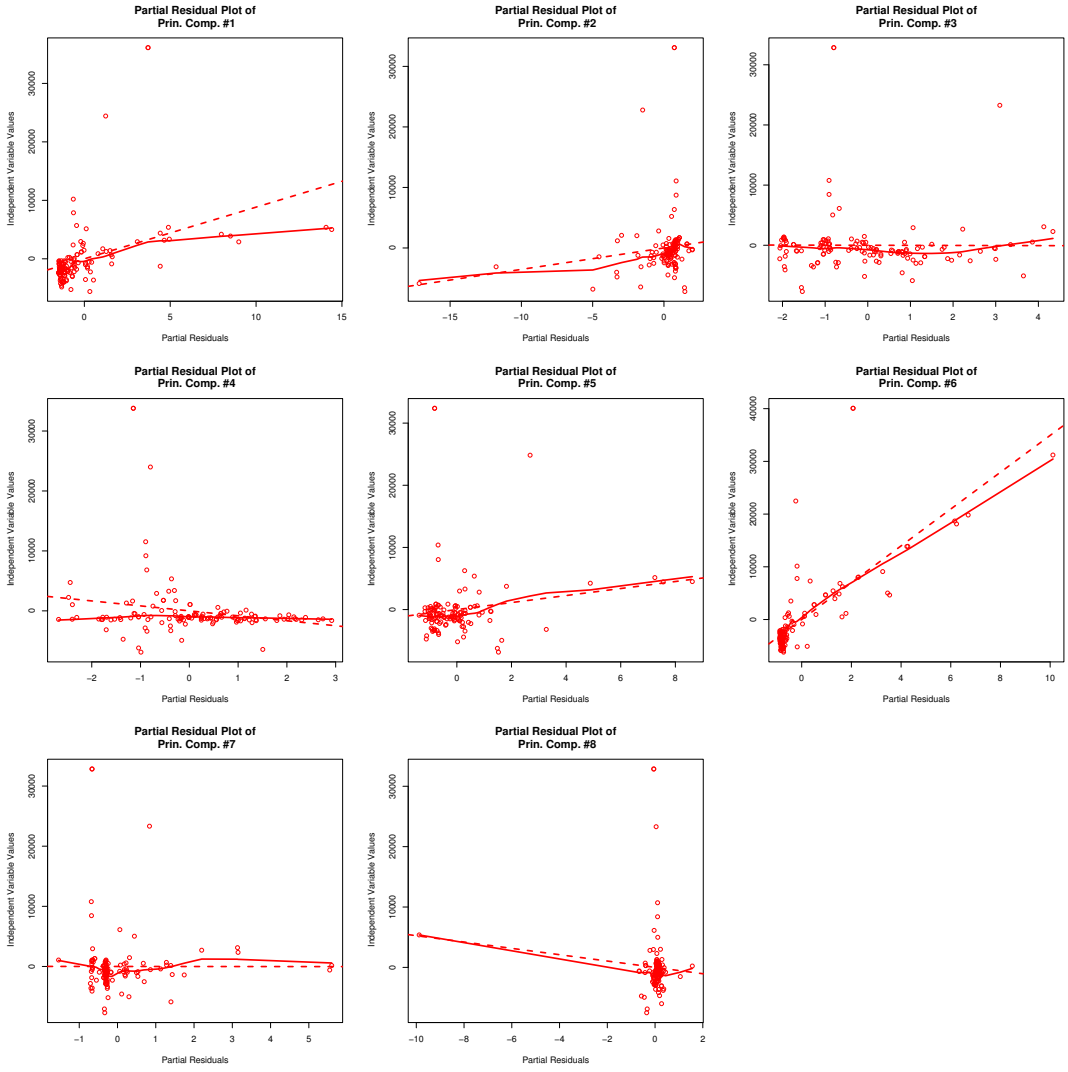


Figure A.1: Partial Residual Plot for Slices: DWARV

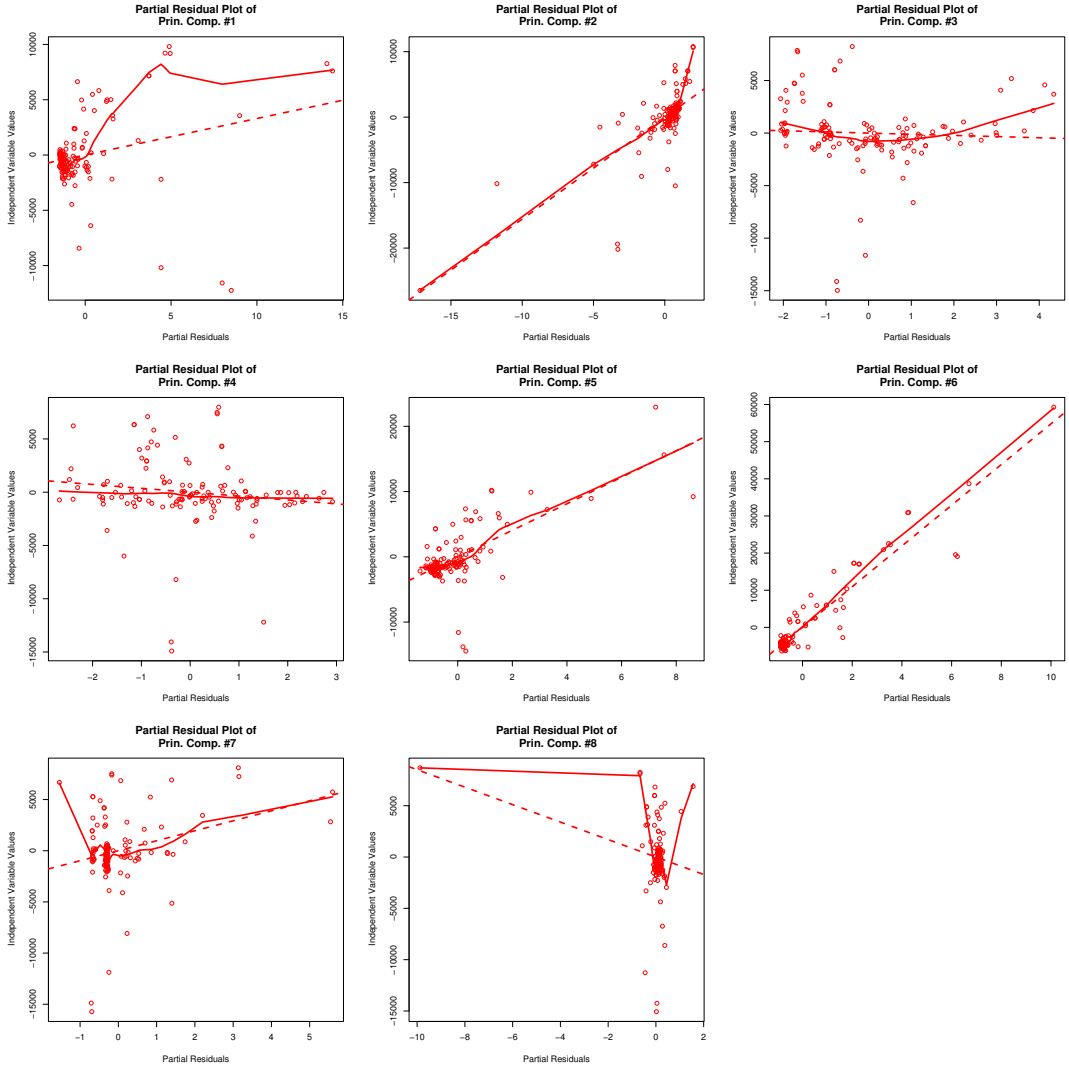


Figure A.2: Partial Residual Plot for Flip-Flops: DWARV

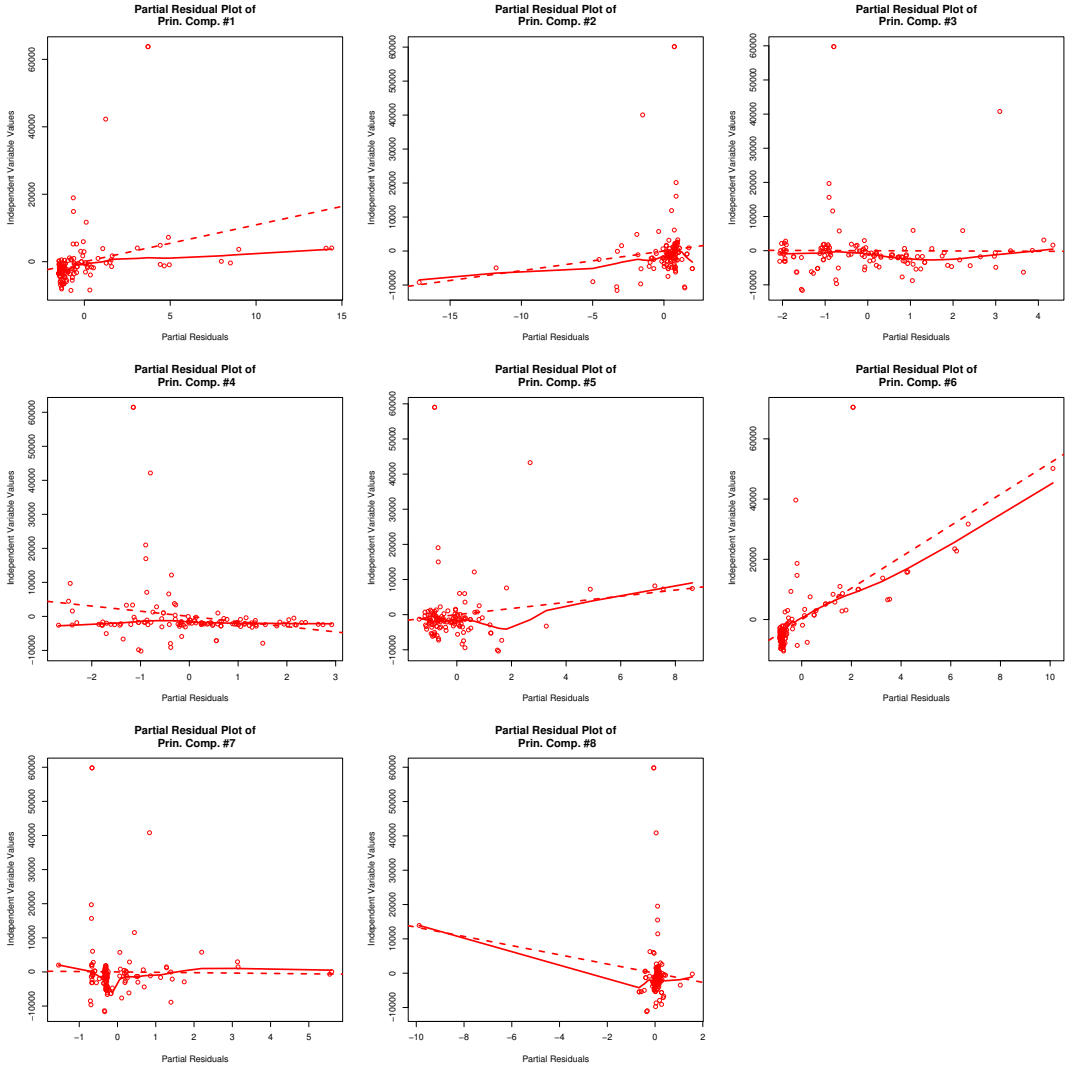


Figure A.3: Partial Residual Plot for Look-Up Tables: DWARV

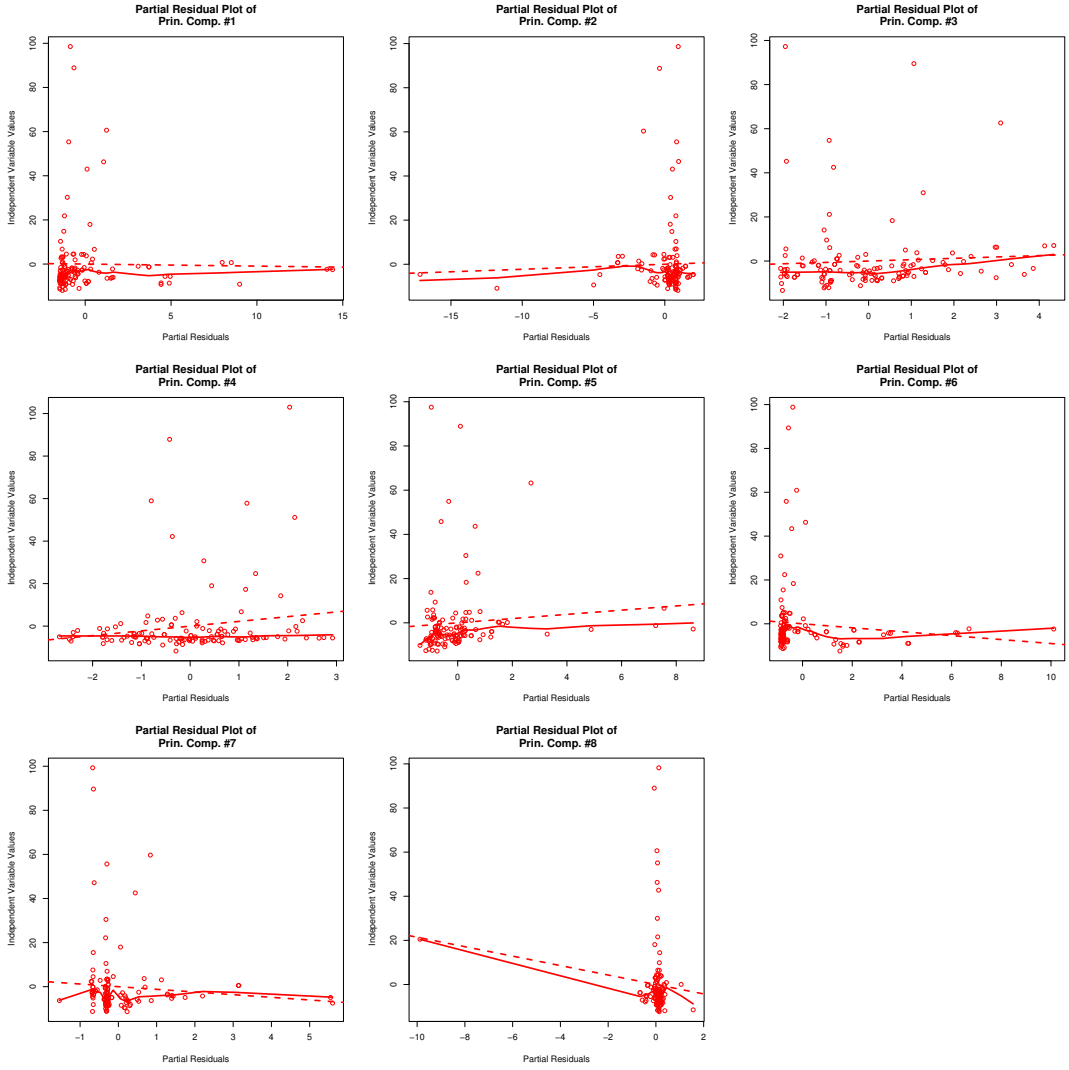


Figure A.4: Partial Residual Plot for Multipliers: DWARV

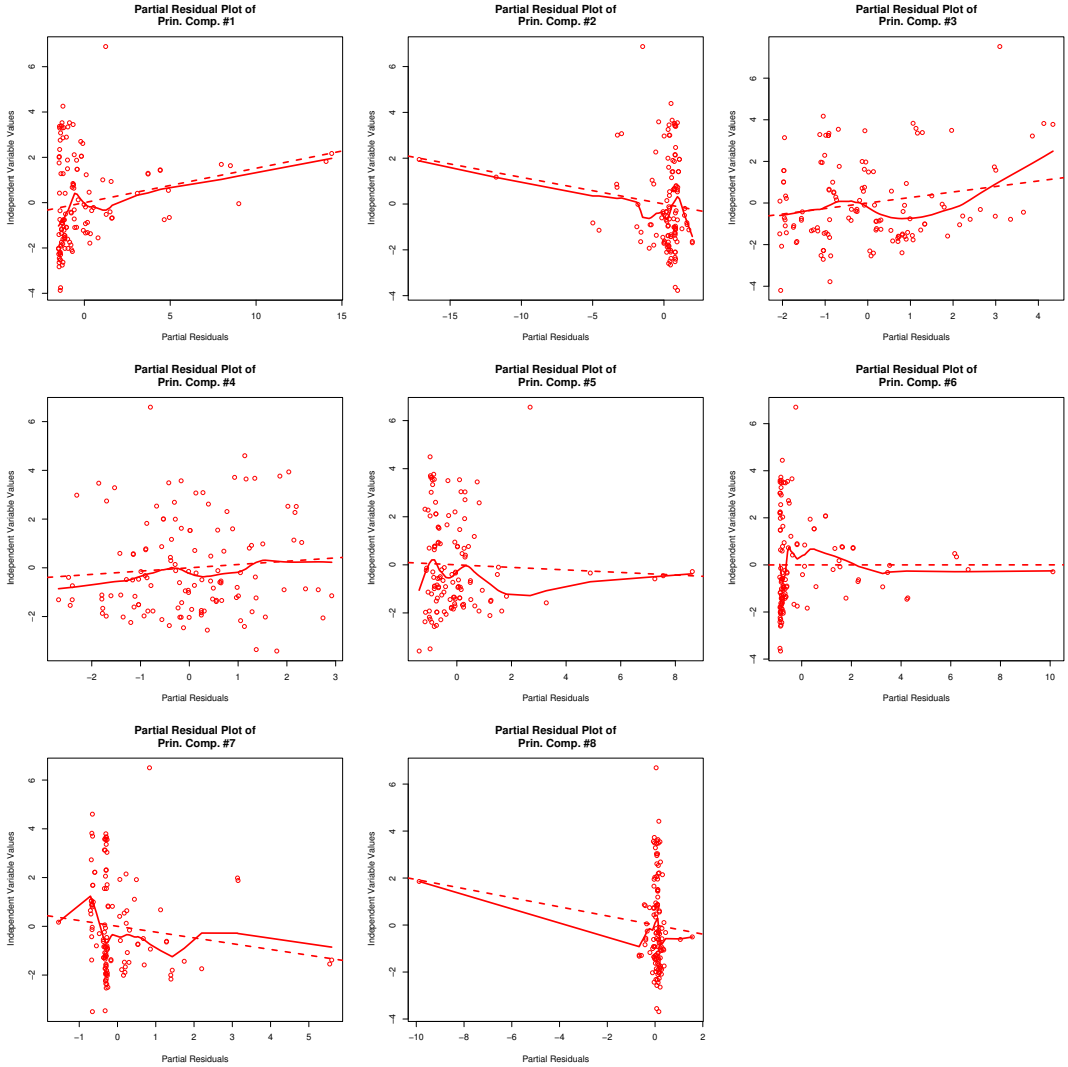


Figure A.5: Partial Residual Plot for the Period: DWARV

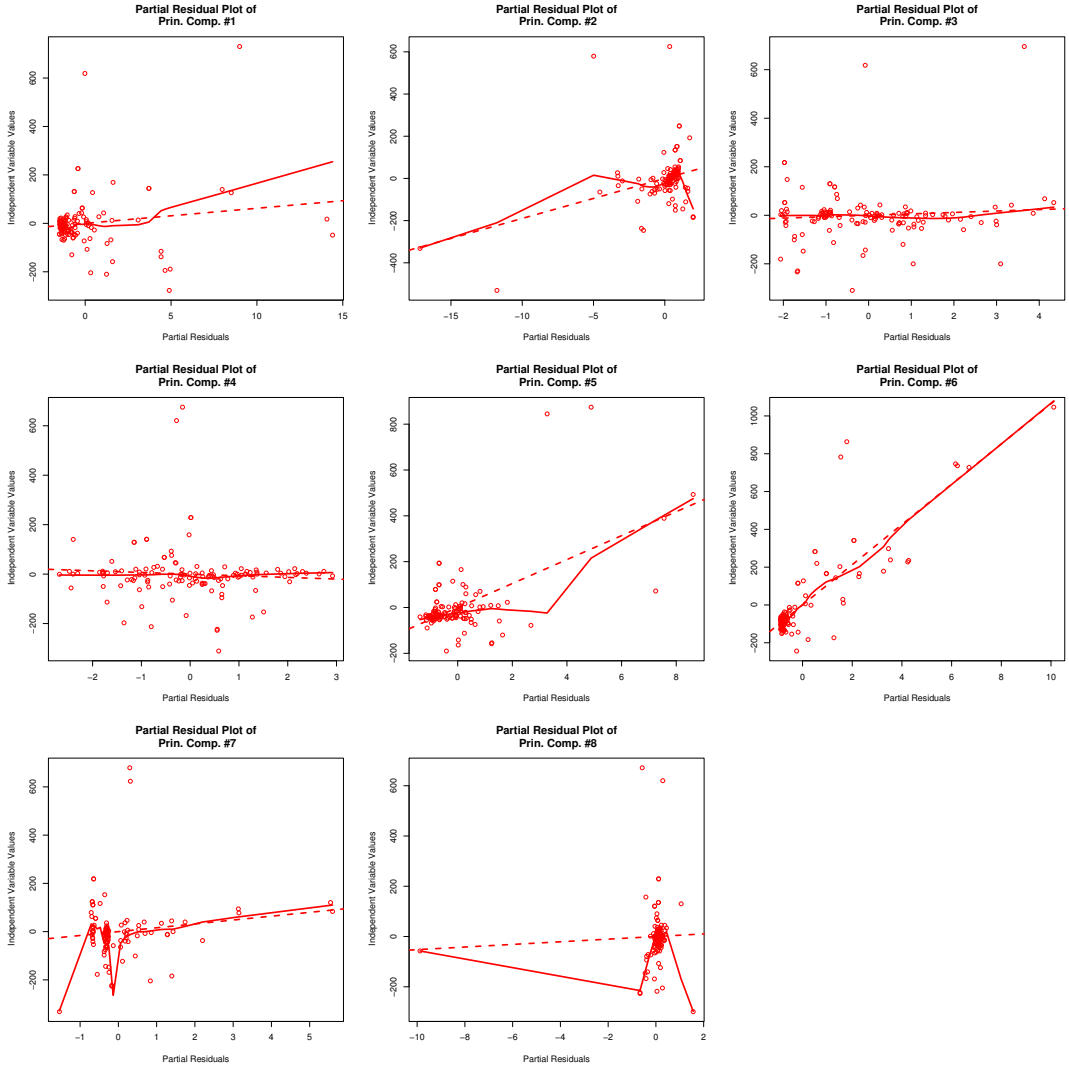


Figure A.6: Partial Residual Plot for States: DWARV

A.2 DWARV - ANOVA Tables

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	8.7843e+09	8	1.0980+09	37.45	< 2.2e-16
Error	3.4598e+09	118	2.9320e+07		
Total	1.2244e+10	126			

Table A.1: ANOVA Table for the Slices dependent variable in the DWARV linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	1.5456e+10	8	1.9320e+09	170.67	< 2.2e-16
Error	1.3358e+09	118	1.1321e+07		
Total	1.6792e+10	126			

Table A.2: ANOVA Table for the Flip-Flops dependent variable in the DWARV linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	1.8649e+10	8	2.3311e+09	24.852	< 2.2e-16
Error	1.1068e+10	118	9.3799e-07		
Total	2.9717e+10	126			

Table A.3: ANOVA Table for the LUTs dependent variable in the DWARV linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	1683	8	210	0.7203	0.6732
Error	34465	118	292		
Total	12244116192	126			

Table A.4: ANOVA Table for the Multipliers dependent variable in the DWARV linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	49.65	8	6.21	1.5551	0.1458
Error	470.99	118	3.99		
Total	520.64	126			

Table A.5: ANOVA Table for the Period dependent variable in the DWARV linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	6072174	8	759022	57.242	$< 2.2e\text{-}16$
Error	1564673	118	13260		
Total	7636847	126			

Table A.6: ANOVA Table for the States dependent variable in the DWARV linear model ($\alpha = 0.01$).

A.3 SPARK - Omitted Plots

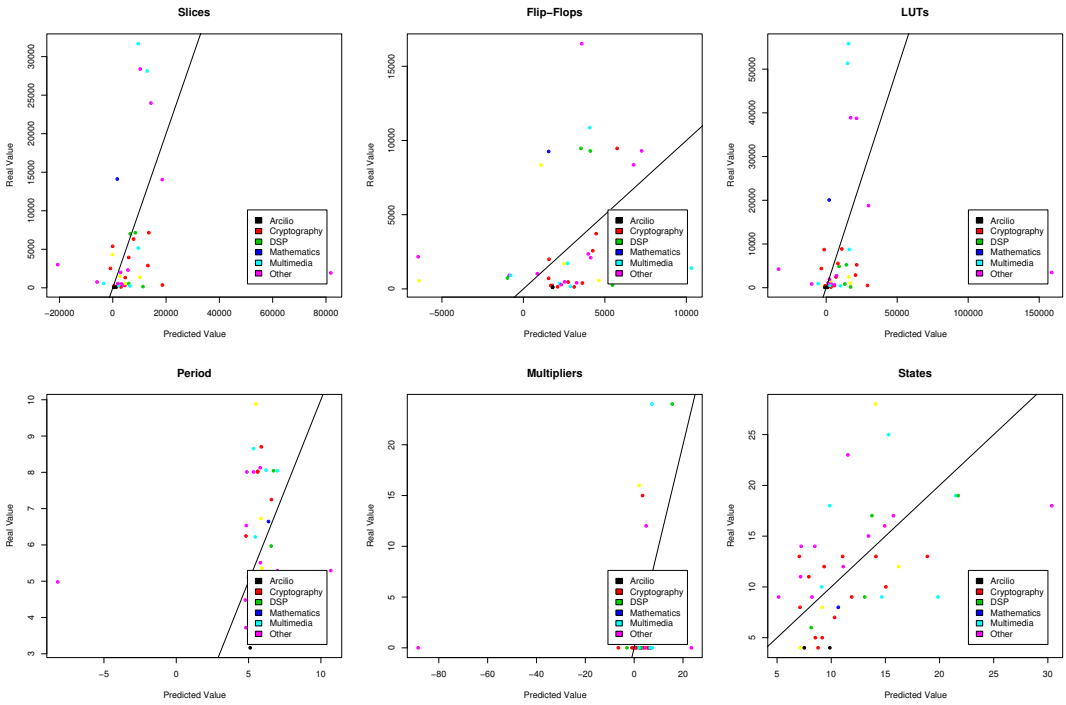


Figure A.7: Omitted Model Predictions for SPARK

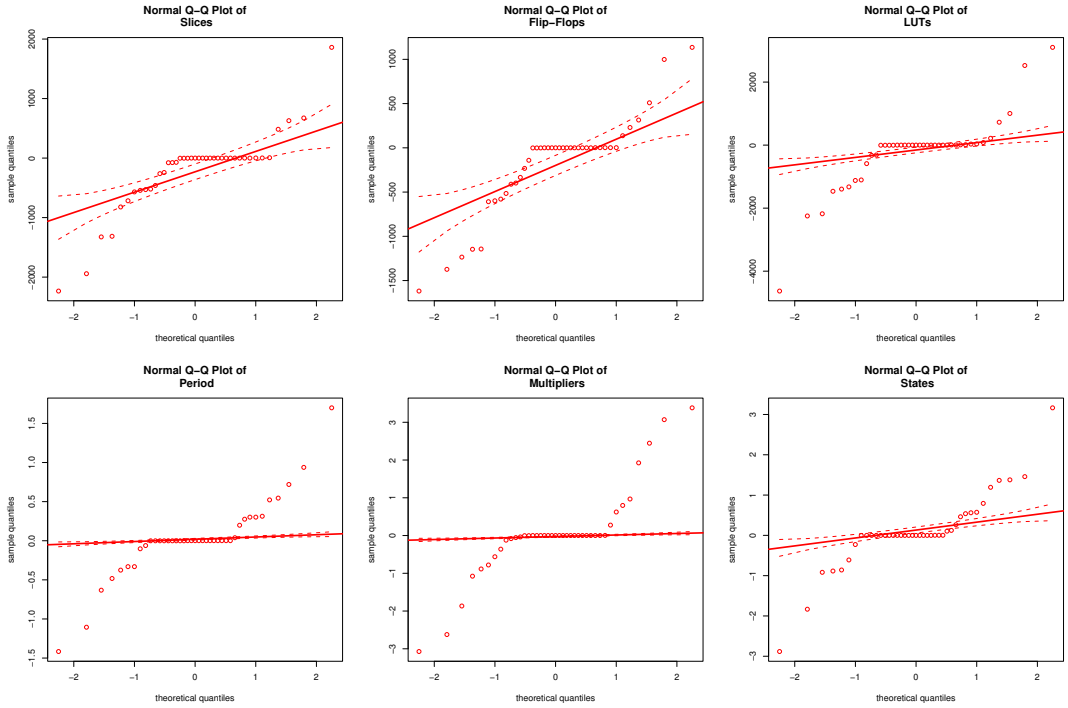


Figure A.8: Omitted Q-Q Plots for SPARK

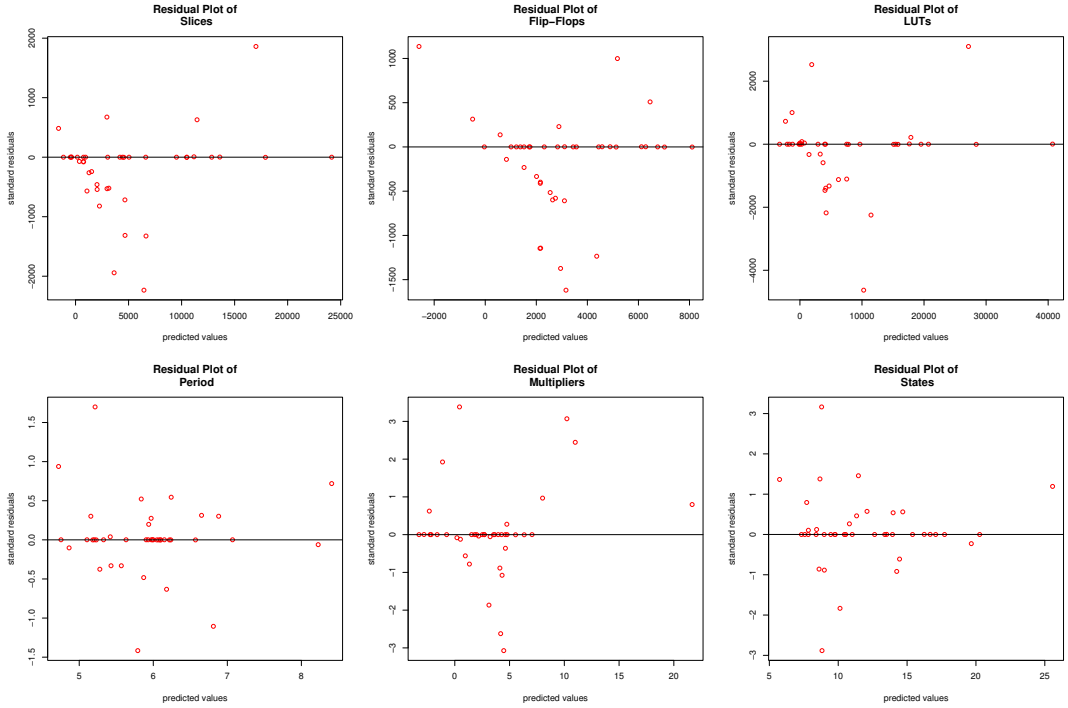


Figure A.9: Omitted Residual Plot for SPARK

A.4 SPARK - Partial Residual Plots

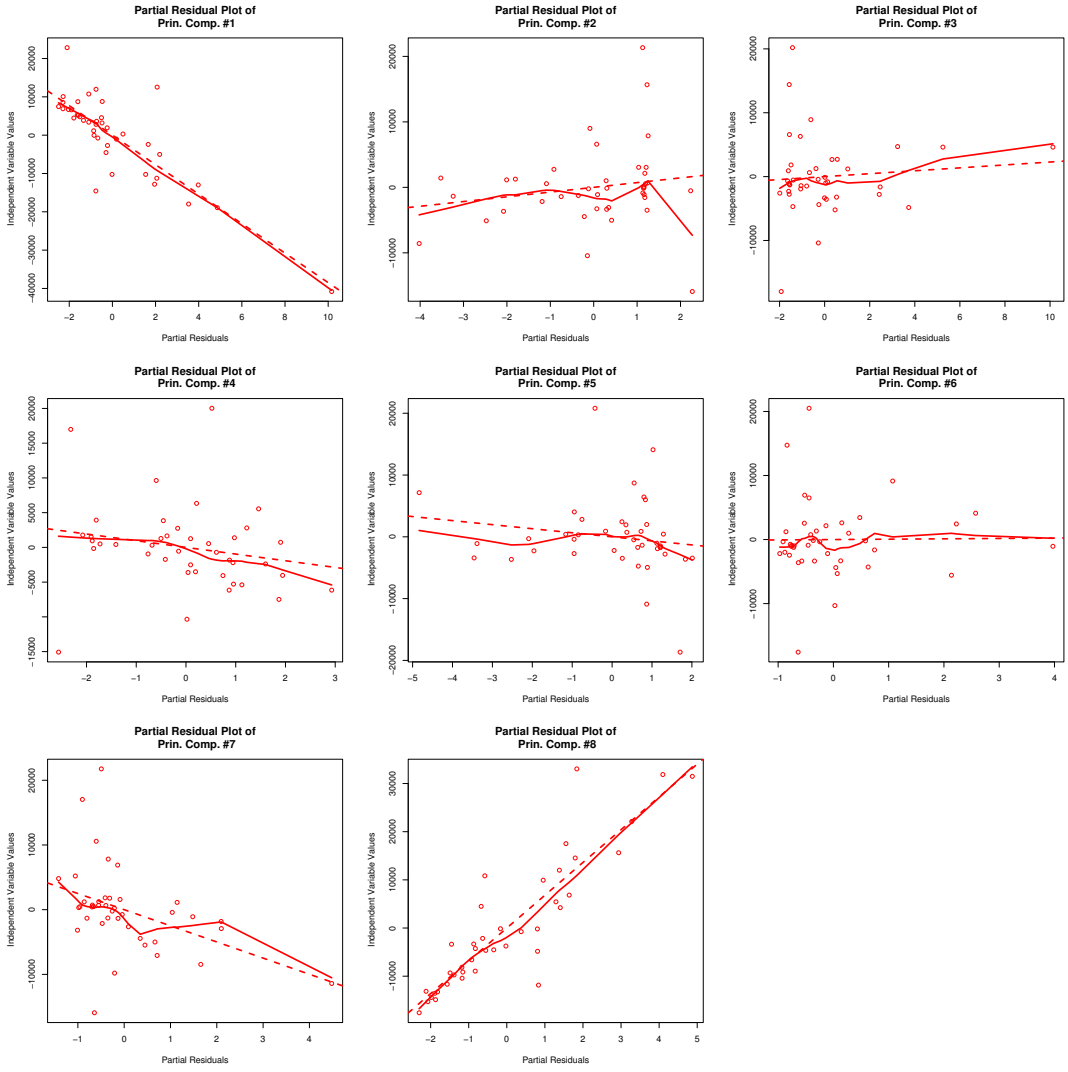


Figure A.10: Partial Residual Plot for Slices: SPARK

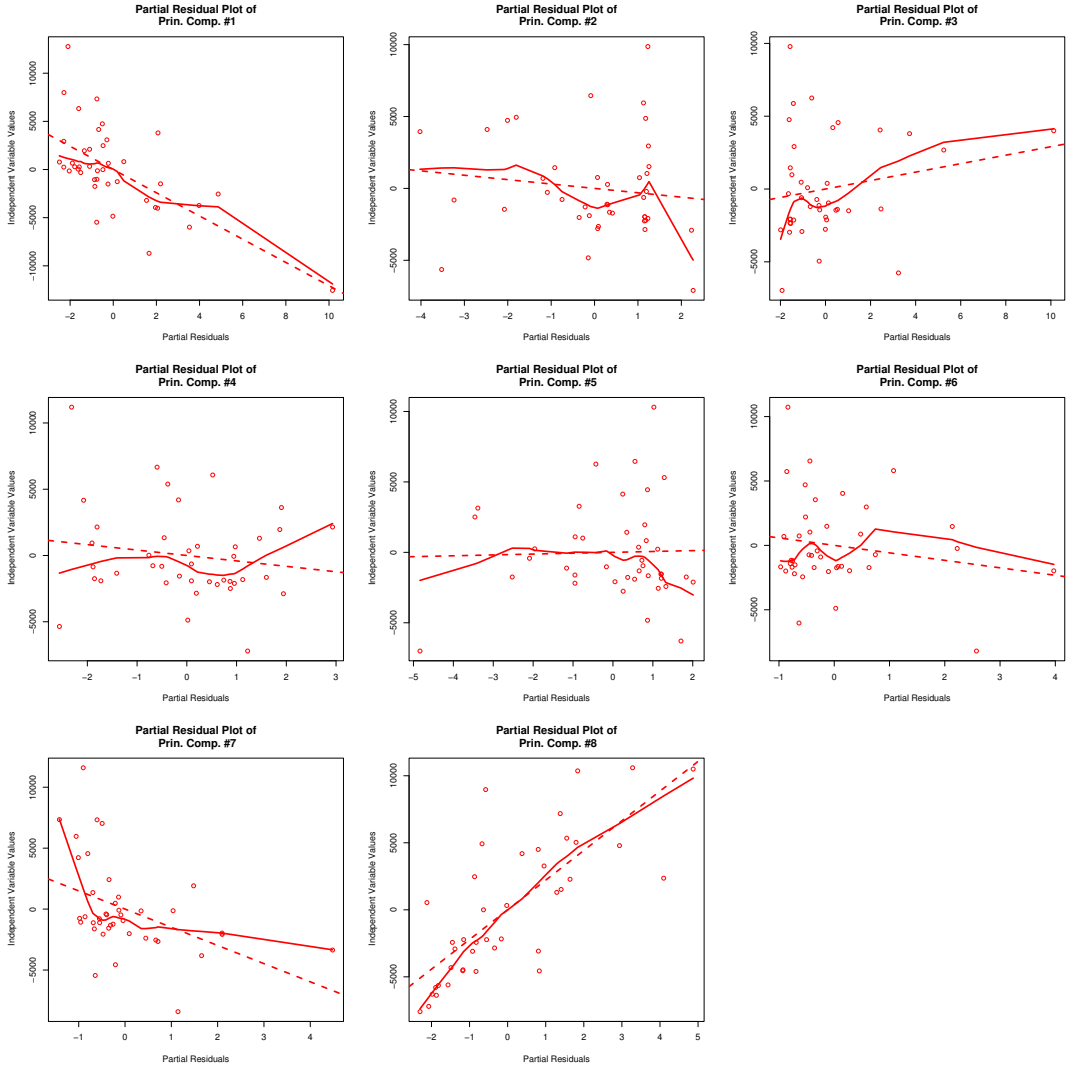


Figure A.11: Partial Residual Plot for Flip-Flops: SPARK

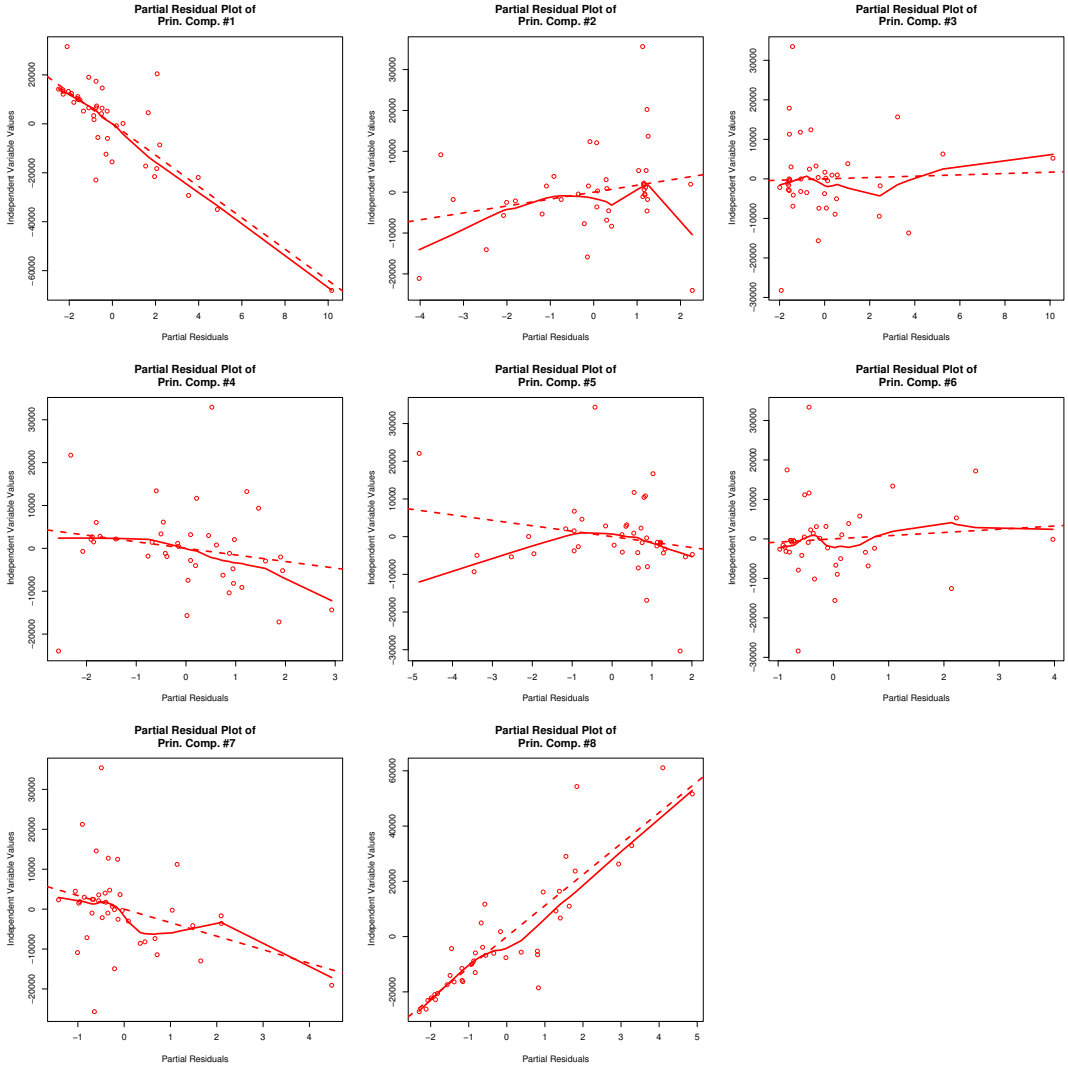


Figure A.12: Partial Residual Plot for Look-Up Tables: SPARK

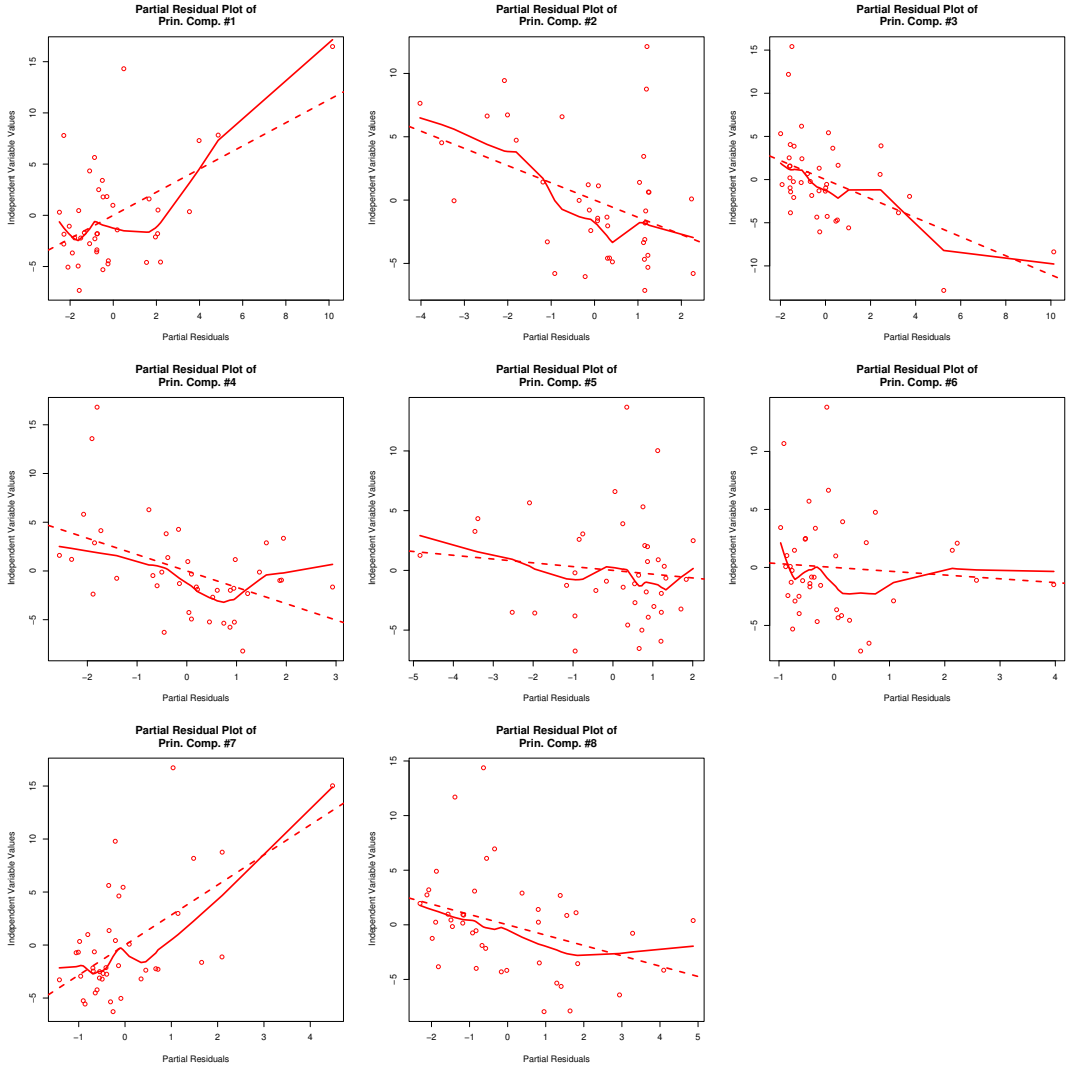


Figure A.13: Partial Residual Plot for Multipliers: SPARK

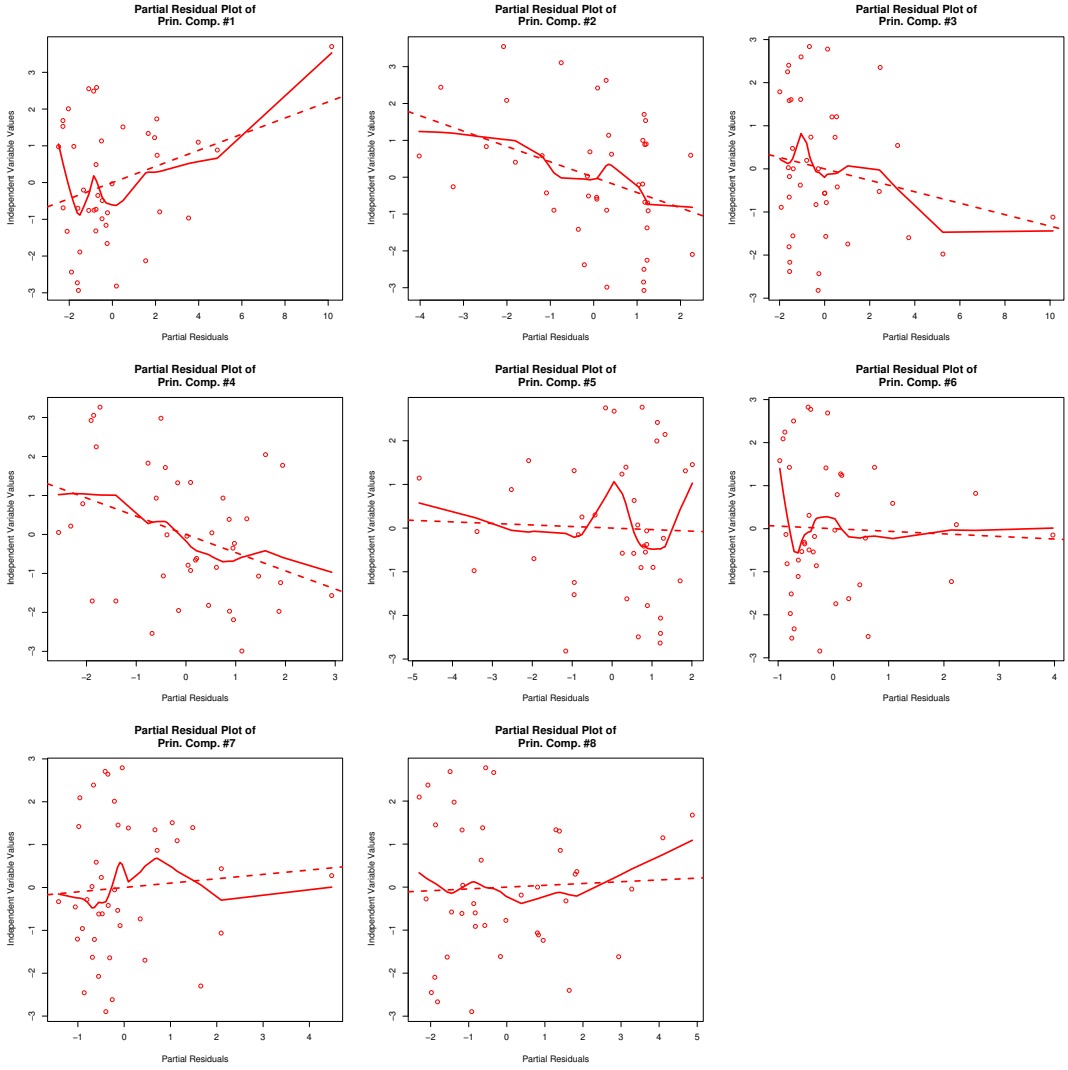


Figure A.14: Partial Residual Plot for the Period: SPARK

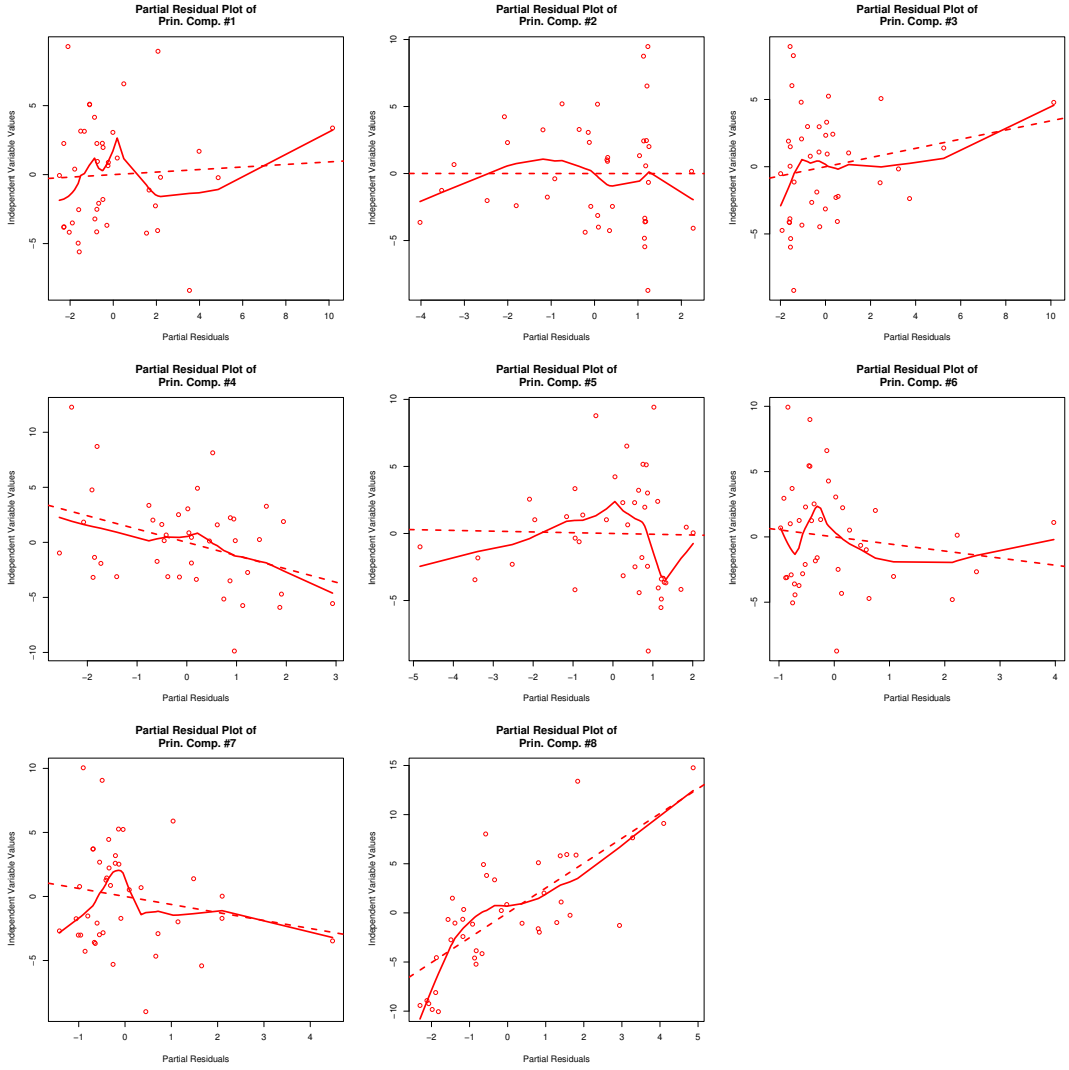


Figure A.15: Partial Residual Plot for States: SPARK

A.5 SPARK - ANOVA Tables

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	1.337659e+09	8	1.672073e+08	3.6444	4.041e-03
Error	1.468166e+09	32	4.588018e+07		
Total	2.805824e+09	40			

Table A.7: ANOVA Table for the Slices dependent variable in the SPARK linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	2.202121e+08	8	2.752651e+07	2.0079	7.752e-02
Error	4.386953e+08	32	1.370923e+07		
Total	6.589073e+08	40			

Table A.8: ANOVA Table for the Flip-Flops dependent variable in the SPARK linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	3.495949e+09	8	4.369936e+08	3.3605	6.615e-03
Error	4.161223e+09	32	1.300382e+08		
Total	7.657171e+09	40			

Table A.9: ANOVA Table for the LUTs dependent variable in the SPARK linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	26.661	8	3.333	1.1161	0.3789
Error	95.549	32	2.986		
Total	122.21	40			

Table A.10: ANOVA Table for the Period dependent variable in the SPARK linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	697.54	8	87.19	3.3687	6.521e-03
Error	828.27	32	25.88		
Total	1525.81	40			

Table A.11: ANOVA Table for the Multipliers dependent variable in the SPARK linear model ($\alpha = 0.01$).

Source of Variance	Sum of Squares	Degrees of Freedom	Mean Square	F-ratio	$P(> F_c)$
Regression	729.00	8	91.13	4.8527	5.585e-04
Error	600.90	32	18.78		
Total	1329.90	40			

Table A.12: ANOVA Table for the States dependent variable in the SPARK linear model ($\alpha = 0.01$).

Contents of CD-ROM

B

This appendix contains a general specification of the contents of the CD-ROM accompanying my thesis. The following sections represent the different directories on the CD-ROM. All the third party sources included on the CD-ROM were published under open source licences.

B.1 `codebase/`

This directory contains the set of C kernels that were used for this thesis. Furthermore, several scripts for acquiring the metrics, compiling the kernels, and synthesizing the kernels are present here.

- `utils/`
  This directory contains a C library for generating the required data for performing simulation from the C kernels.
- `SPARK/`
  This directory contains the C kernels used in the SPARK model.
- `Instrumented/`
  This directory contains all sources used in the DWARV model, but are parially instrumented for generating the required data for performing simulation.
- `DWARV/`
  This directory contains all sources used in the DWARV model. Furthermore, all generated VHDL sources and synthesis log files can be found here.
- `Results/`
  In this directory one can find the simulation and synthesis results for SPARK and earlier experiments.
- `scripts/`
  This directory contains the synthesis scripts for the SPARK framework.
- `Original/`
  Here one can find the original unmodified sources for all kernels.

B.2 `Metrics/`

This directory contains the Elsa/Elkhound sources from UC Berkeley extended with several classes that collect the software metrics from the C kernels. The added classes can be found in the following files:

- `metrics.cc`
  Base class of all the metrics.
- `halstead.cc`
  The Halstead metrics can be found here.
- `simple.cc`
  The number of statements, number of variables, Cyclomatic complexity, Basili-Hutchens complexity, and Prather's μ metrics can be found here.
- `nesting.cc`
  The Maximum, Average, and Cumulative Nesting Depth, Piwowarski complexity, and Gong-Schmidt complexity can be found here.
- `aicc.cc`
  This file contains the AICC metric implementation.
- `npath.cc`
  This file contains the code for determining the NPATH metrics.
- `scopenumber.cc`
  This file contains the implementation of the Scope Number.
- `dataflow.h`
  In this file one can find the templates used in the Forward Data Flow Analysis used in the pathlen.cc file.
- `oviedo.cc`
  This file implements the Oviedo DU metric.
- `taidu.cc`
  This file implements the Tai DU metric.
- `pathlen.cc`
  This file implements the Average and Maximum Path Length metrics.
- `memaccess.cc`
  This file implements the metrics: Loads and Stores.
- `looparea.cc`
  This file implements the Loop Complexity metric.

B.3 `statisics/`

This directory contains the datasets and scripts used for performing the statistical analysis.

B.4 `thesis/`

This directory contains the original source for this thesis document.

Curriculum Vitae

Roel Meeuws was born in Rotterdam, the Netherlands on the 24^{th} of May 1982. He graduated in 2000 at the secondary school Blaise Pascal in Spijkenisse. He began his study of Computer Science in 2000 at Delft University of Technology. After he received his B.Sc. degree in Computer Science in 2004, he joined the Computer Engineering laboratory, led by professor Stamatis Vassiliadis. He performed his thesis work at the Delft Workbench project under the supervision of Koen Bertels, Ph.D. His thesis is titled "A Quantitative Model for Hardware/Software Partitioning. His research interests include: reconfigurable computing, computer architectures, programming languages, operating systems, compilers and embedded systems.