Rcosy DES.6392
Reconfigurable Compiler System - Rcosy
Towards a Quantitative Model for Hardware/Software Partitioning.

Roeland J. Meeuws Yana Yankova Koen Bertels

Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail: {rmeeuws,yyankova,koen}@ce.et.tudelft.nl

Abstract

In the field of Reconfigurable computing the problem of partitioning a system in hardware and software parts has been tackled in many different ways. Our idea is to devise a quantitative model based on software metrics that are representative for different hardware characteristics. In this paper we research the different estimation and partitioning strategies currently employed in the field as a preparation for developing such a model. We determine that much research has been done on area and speed metrics, while power, memory, and communication aspects have not received much attention. Furthermore, we find that many different partitioning strategies have been developed. Many of which aim to be hill-climbing algorithms, i.e. algorithms that try to find a non-local optimum. Based on the literature we review, we conclude that estimation in a future quantitative model must have a clear notion of error and precision. Furthermore, the use of software metrics as a basis for hardware software partitioning has not been sufficiently explored. On the subject of partitioning strategies, we conclude the need for incorporating the dynamic aspect of reconfigurable computing. Furthermore, partitioning and estimation should be on the same level of granularity.

Keywords: Hardware/Software Partitioning, Hardware Estimation, Software Metrics, Reconfigurable Computing

Contents

Chapter 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 [1]. 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 [2] 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. These application specific systems (ASICs) 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 Reconfigurable Computing (RC). Reconfigurable Computing combines programmable software components with programmable hardware components [3], like FPGAs. Hence, Reconfigurable Computing advances the idea of heterogeneous systems by introducing 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. To clarify the concept of Reconfigurable Computing, let us look at [4], where three levels of programmability are identified for both control-flow models (software) and data-flow models (hardware) (Table 1.1). The programmability in Reconfigurable Computing 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 [4]

1.1 A case study: Software Radio

As an illustration of how Reconfigurable Computing can help alleviate processing requirements, while remaining flexible, we will now look into the Software Radio[5, 6, 7, 8, 9]. In mobile communications many different frequency bands are used for different applications, like GSM for voice, GPRS for Internet, and 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 beneficial.

However, the computing power required for a software radio on a conventional processor are quite high. As an illustration, look at the example in [6], 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. [6], argues for the use of a heterogeneous multiprocessor architecture, i.e. an architecture that comprises different ASIC and General Purpose processors, to tackle this problem. Nevertheless, such an approach may be expensive. [7, 8] suggests Reconfigurable Computing (especially FPGA technology) as a means to make radio signal processing possible, while remaining flexible enough to service different networks at different times. In [8] 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, [3] argues that while the Von Neumann Machine is supported by an extensive and mature base of tools, apis, and design methodologies, no such extensive support is available for the Reconfigurable Computing paradigm. In other words, for Reconfigurable Computing to be commercially applicable, it should have a comparable support base. In recent years some tools have been developed to attack this problem.

The problem doesn't end there, however. Because Reconfigurable Computing moves away from the von Neumann model, the extensive base of support should be adapted accordingly. In [10], for example, current hybrid programming models are pointed out to be immature, because FPGAs and 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 multi-threading 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 [11] 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 reconfigurable computing is presented in [12], 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 [13] a redefinition of

the term algorithm, as used in computability theory, is presented tailored to Reconfigurable Computing.

1.3 MOLEN and Delft Workbench

Our research group has been working on reconfigurable computing for some time and we have developed a reconfigurable programming paradigm, with an accompanying platform, called the MOLEN programming paradigm [14] and the MOLEN polymorphic processor [15] respectively. 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 these papers mention, 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 (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 don't 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.

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.

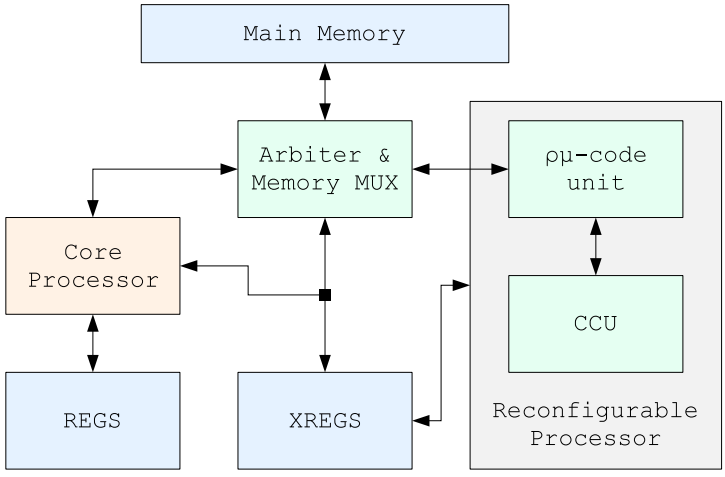


Figure 1.1: A basic overview of the general MOLEN platform.

The basic structure of the MOLEN platform is depicted in Figure 1.1. 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 microcode unit or $\rho\mu$-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. a

FPGA.

We have already established the need for extensive tool support when considering the acceptance of reconfigurable computing in industry. Tool support and integration for reconfigurable computing is the main focus of the Delft Workbench project. Its research covers the entire design process from code profiling to compilation. The project has four main objectives [16]:

- 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.

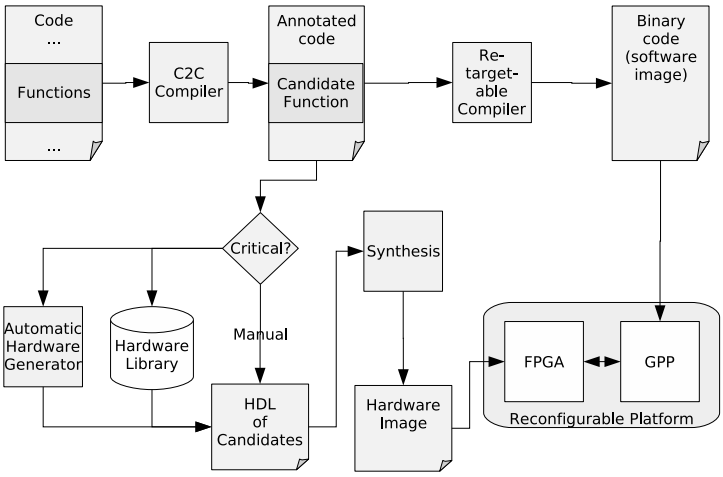


Figure 1.2: A basic overview of the general Delft Workbench tool-flow.

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 1.2. 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. Therefore, in order to develop such a model this paper investigates different metrics, estimation strategies, partitioning models, and partitioning strategies. The paper is organized as follows. In Section 2 I will discuss metrics that indicate hardware aspects of a (part of a) program and estimation techniques to determine them. Then, in Section 3 I will review hardware/software partitioning models and strategies. Finally, in Section 4 I will briefly discuss the results presented in this paper and indicate possible elements for the decision model to be used in the code profiler/partitioner in Delft Workbench.

Chapter 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 will discuss different metrics and techniques to estimate them. First I will briefly present the different papers I reviewed. Then, I will review different aspects of the estimation schemes by finding ways to classify these metrics. Furthermore, we will briefly go into synthesis and optimization and how we may measure its effects.

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.

2.1.1 Area

Area estimation has been tackled in various ways by different groups. In [24] a lower bound on area is estimated under certain performance constraints. Specifically, they estimated the number of modules of each type and the area

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

Table 2.1: 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 2.4.

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 busses required.

Because area estimation tries to predict the results of synthesis tools, [21] 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 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)

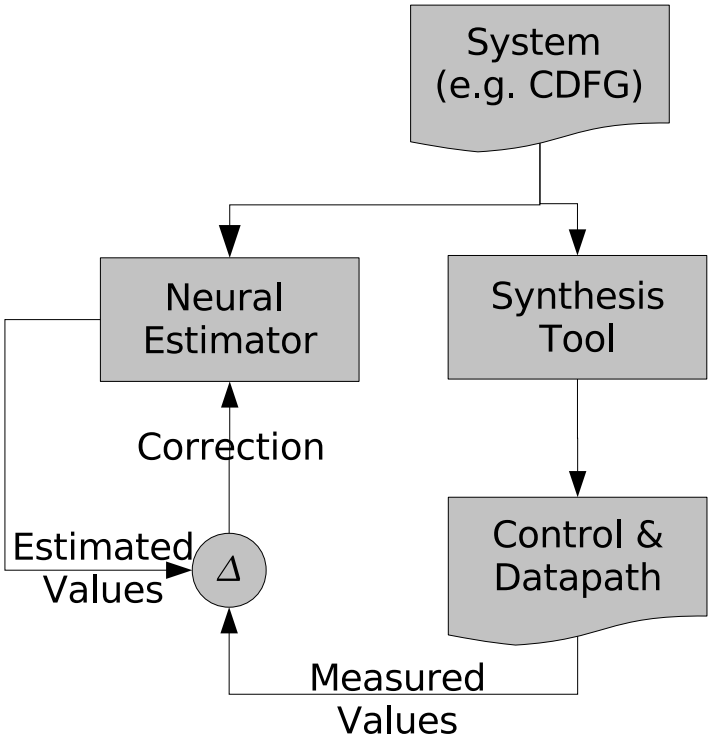


Figure 2.1: The NESTIMATOR neural estimator from [19] working together with a synthesis tool. The results of synthesis are compared with estimates and the neural network is adapted accordingly.

in [19] takes a more non-deterministic approach. The estimator in this paper is first "teached" 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 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)
- 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 [18]. This paper describes how a preprocessed information data structure holding basic design information can be maintained between succesive 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 [25]. 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 [23] 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 [28] 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.

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-linear2	multi-input AND or OR	$y = p_0 + (p_1 * x/2)(z/2 - 1)$

Table 2.2: Bit-based area models of the main operators in LUTs, as mentioned in [17].(y=area(LUTs), n=bit-width, m=inputs, p_i= experimentally determined constants)

In an effort to guide optimizations in an SA-C compiler [17] 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 2.2. 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 [31], 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 [22]. 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 [26] 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 [26] 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 [29] 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 [32] the authors of [20] 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 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 [27] 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.

2.1.2 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 [24] 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 [25] 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 [27] 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 [36] 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 [28] discussed earlier, also estimate latency by first determining the longest path. This preliminary value is then

Paper	Dynamic/ static	Level of Design	Application Domain	Data structure	Node model	Granularity	Strategy	Complexity	Error
[33]	Static	Instruction-level	n/a	Intermediate code	n/a	Application	Complexity	n/a	8%
[29]	Dynamic	System	Communicatio	(annotated) DFG	Bit-width based	Entire graph	Scheduling	$O(simulati$ nonlinear	10%
[20]	Static	System/ Behavioral	Multimedia, Mathemati-cal	VHDL Ab-stract Syn-tax Tree	Bit-width based	Entire Graph	Scheduling	linear	13%
[24]	Static	Behavioral	Multimedia	DFG	Simple	entire graph	Scheduling	$O(nc^2)$ (c: time steps)	n/a
[34]	Dynamic	Behavioral	Multimedia, Cryptogra-phy, Com-pression, General Purpose	n/a	n/a	Loops	Simulation	$O(simulati$ $O(1)$	n/a
[25]	Static	Behavioral (during par-titioning)	n/a	(annotated) DFG	Simple	entire graph	Scheduling	$O(n^2 \log n)$	n/a
[35]	Static	Behavioral	Communicatio	n/a	Bit-width based	Functions (=Occam processes)	Complexity	n/a	n/a
[30]	Static	Behavioral (during par-titioning)	n/a	Task graph	Simple	Entire graph	Incremental	linear	n/a
[36]	Static	Behavioral (during par-titioning)	Communicatio	Task graph	Simple	Entire graph	Scheduling	linear	n/a
[27]	Static	System/ Behavioral	Multimedia	H-CDFG	Simple	Entire Graph	Hierarchical, Allocation (bipartite matching)	$O(n \log n)$	n/a
[28]	Static	Behavioral (during par-titioning)	Multimedia, General Purpose	(C) DFG	Simple	Node clus-ters	Scheduling	n/a	n/a
[37]	Dynamic	Behavioral	Multimedia, Cryptog-raphy, General Purpose	Hierarchical Loop Graphs	n/a	Loops	Simulation	$O(simulati$ $O(1)$	n/a
[21]	Static	Behavioral	Multimedia, General Purpose	PACT HDL Abstract Syntax Tree	Simple	Entire Graph	Scheduling	nonlinear	n/a

Table 2.3: 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 2.4.

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.

Area estimation from Matlab code as presented in [29, 20] was already discussed in the previous section. These two papers also describe delay estimation techniques. In [29] 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 [20] 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 functional units 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).

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

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 [34, 37] 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, [34] 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. [37], on the other hand, puts the former comment in perspective by pointing at the 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, [35] describes using the 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 2.4.

The algorithm in [21] 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.

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 2.4: Bit-based timing models of the main operators as mentioned in [35].

The μ–profiler as described in [33] uses the number of cycles gained by hardware implementation to drive the discovery and selection of custom instructions for 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 ILP, pipelining, and other software optimizations.

One such optimization, i.e. loop unrolling, is specifically studied in [31]. 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.

2.1.3 Power

Paper	Dynamic/ static	Level of Design	Application Domain	Data structure	Node model	Granularity	Strategy	Complexity	Error
[38]	Static (dynamically determined input probabilities)	Behavioral	Communicatio 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
[27]	Static	System/Behav	Multimedia	H-CDFG	Simple	Entire Graph	Hierarchical, Allocation (bipartite matching)	$O(n \log n)$	n/a

Table 2.5: 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 2.4.

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 [40], 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 2.6. 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.

The system-level partitioning algorithm in the TOSCA co-design environment [41] 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. 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.

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

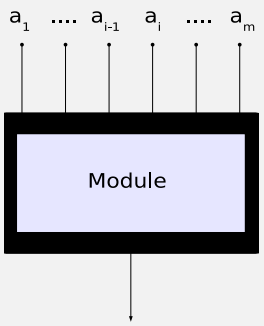
Table 2.6: Taxonomy of power estimation techniques as presented in [40].

- *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 busses 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 [40], is presented in [27], along with area and speed estimation techniques, discussed earlier. The paper estimates trade-off curves for power



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).

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

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 [38].

consumption against varying timing constraints. The power estimation differs from the area and time estimation by taking into account the execution frequencies of functional units and the clock frequency.

The authors of [22] 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 [38]. This particular approach utilizes both 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 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 Other metrics

In Section 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.

2.2.1 Communication

Estimating the amount of communication is also present in some area, speed, and time estimation techniques (e.g. [24, 21, 20, 30, 39, 28]), 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 [23]. 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 [42]. 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 transfered along the edges (in bits).

2.2.2 Memory Usage

Few hardware-oriented memory usage estimation strategies have been proposed, but in e.g. [34], 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, 43, 44, 45] and some hardware/software partitioning algorithms [28, 36] 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 [46], 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 [47] 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 [45] 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 [44]. 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.

2.3 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 ([45]) or may indicate code most susceptible to optimizations ([37]), but software measures also make comparisons of hardware and software implementations possible. In [48], for example, we find software energy models that enable the authors to compare different hardware / software partitions. There are some problems with such comparisons, though:

- *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 cyclometric complexity; it's not at all clear if software (C) and hardware (VHDL) cyclometric complexity are equivalent or comparable.

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 will discuss different aspects of the presented metrics and estimation strategies, and explore possible classifications.

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 will 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 [40], however, we do find some cases of dynamic techniques like dynamic behavioral activity-based estimation (Table 2.6). 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 [34, 37, 29, 45, 40] all concentrate on speed and memory usage. These characteristics are often

dependent on the input and change during run-time. Still, hardware estimation mainly focusses 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.

2.4.2 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 [40].

It is not the aim of this paper to present an exhaustive study of system design steps, therefore we will 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 2.1, 2.3, and 2.5, 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. While, in this phase accuracy is not very important yet, estimates should be 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.

2.4.3 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 *Data Flow Graph* (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.

- *CFG*
  In the traditional von-Neumann computing paradigm, an often used representation of programs during e.g.compilation or analysis is the *Control Flow Graph* (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 *Control- and Data Flow Graph* (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 [49, 27] 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 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 [38] a *State Transition Graph* (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 Trees among others (AST), among others. However, these models, like CFGs, do not always directly represent data dependencies, which makes determining parallelism inaccurate.

2.4.4 Strategies

Different strategies have been proposed for different data structures and models, but some approaches are more alike then 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 functional units 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 [50]. 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 functional units), and the edges representing possible mappings. An advantage of using synthesis-like allocation in estimation is that separate estimates for registers, functional units, 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, [22], 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 reevaluate 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 reestimating the measures can be beneficial.

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 2.1, 2.3, and 2.5 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, images, 3d, etc. 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 how a high degree of parallelism, these performance requirements can be 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 algorithm, 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.

2.4.6 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 grain 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.

2.4.7 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 [18, 35] 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 [28].

- 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.

2.4.8 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 [38] 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.

2.5 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 [31].

The optimization-aware estimation techniques discussed in this paper have mainly focused on loop optimizations, like loop strip-mining, loop unrolling, loop pipelining, etc. [17, 31, 38]. 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

live analysis [51], procedure exlining [52], partial evaluation [53], etc. Accurate estimation tools should somehow account for such optimizations, however, not many such tools have been proposed in literature.

Chapter 3

Hardware/Software Partitioning

In the previous section we have discussed 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. In the past partitioning was often performed by hand, but in more recently 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 will discuss several partitioning strategies and how they connect with estimation and profiling.

3.1 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 will discuss some of these algorithms.

3.1.1 Greedy

An early approach to the partitioning problem, presented in [55], used a greedy algorithm. Greedy algorithms use locally optimal decisions to approximate the globally optimal partitioning. In case of [55], 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 fulfils the constraints to hardware is used in many other

Paper	Dynamic/ Static	Strategy	Criteria	Model/ Data Structure	Granularity of Partitioning	Time Complexity
[54]	Static	Simulated Annealing	n/a	n/a	n/a	n/a
[55]	Static	Greedy	Minimal area, data-rate constraints	System Graph Model (like H-CDFG)	operations	linear
[49]	Static	Greedy (see [55])	Minimal area, data-rate constraints	Hierarchical Sequence Graph	operations	n/a
[56]	Static	Simulated Annealing	Minimal communication cost	Petri-nets, (annotated) CDFG	operations	$O(tn)$ t=temperature steps
[57]	Static	Simulated Annealing	Hardware suitability (compare local phase [58])	(extended) C^x syntax graph	basic blocks	n/a
[58]	Static	GCLP	GC objective function (e.g. Area combined with speed)	n/a	Tasks (instruction level subgraphs)	$O(ne)$, e=edges
[59]	Static	Binary Constraint Search	Constraints of encapsulated partitioning algorithm	n/a	n/a	$O(part(S))$ $part(S)$ = encaps. part. alg.
[34]	Static	Dynamic Programming	Temporal size of loops / leaf functions	n/a	loops, leaf functions	n/a
[60]	Static	GCLP (MIBS)	See [58]	CDFG	Tasks	$O(n^3 + n^2B)$, B=bins
[61]	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
[36]	Dynamic	Greedy, Clustering	Minimize area, timing constraints	Task Graph	task clusters	n/a
[62]	Dynamic	Clustering	Area constraints	CDFG	loop clusters	linear
[28]	Static	Evolutionary (Genetic)	maximize fitness (minimize area and interconnect)	DFG	fine:operations coarse:DFGs	n/a
[63]	Dynamic	Evolutionary	Maximum rank (Pareto ranking in power and price)	Task Graph	Tasks	n/a
[37]	Static	Greedy	Temporal size of loops / leaf functions	n/a	loops	n/a
[64]	Static	Dynamic Programming	Minimum latency, resource constraints	DFG	Tasks	polynomial
[65]	Static	Simulated Annealing, Kernighan-Lin	Minimize latency, area constraints	Call graph	functions	n/a

Table 3.1: Inventarization of several papers on hardware software partitioning with corresponding partitioning schemes, criteria, and data structures

```
function GreedyPart(system: graph): graph
  // Initialize partition and cost
  partition := InitPartition(system);
  cost := Cost(partition);

  // partition each node
  for each node in system do
    partition' := partition;
    // move node in partition' from/to HW
    swap(node, partition');
    if feasible(partition') then
      if Cost(partition') < cost then
        partition = partition';
        cost = Cost(partition');
      end if
    end if
  end for
  return partition;
end function
```

Algorithm 3.1: Pseudo code of an example greedy algorithm for solving the partitioning problem.

partitioning schemes in literature. In [37], for example, the most frequently executed loops are migrated to hardware. The CRUSADE algorithm [36] allocates task clusters with the highest priority in hardware first.

3.1.2 Simulated Annealing

```
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 3.2: Pseudo code of a generic simulated annealing algorithm for solving the partitioning problem as found in [56].

In order to remedy the flaw of greedy algorithms to stick 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 [54], that is often used in hardware software partitioning [57, 56], 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 first.

3.1.3 Kernighan-Lin/Fiduccia-Mattheyses

```
function FMPart(system: graph): graph
  partition = InitPartition(system);
  repeat
    for each node in partition
      unlock(node);
    gain = 0;
    while unlocked nodes remain
      node = MaxGainUnlockedNode(partition);
      swap(node, partition);
      gain = node.gain;
      lock(node);
      for each node in partition
        update node.gain;
    end while
  until gain <= 0;
  return partition;
end function
```

Algorithm 3.3: Pseudo code of a Fiduccia-Mattheyses algorithm for solving the partitioning problem.

Classical graph bipartitioning 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, [65] shows that Simulated Annealing can deliver better quality partitions in less time.

3.1.4 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 [66] and since then several different types of evolutionary strategies have been developed, each with their own niche. For combinatoric problems

```
(* 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 3.4: Pseudo code of the Genetic Algorithm for solving the partitioning problem as found in [61].

like graph partitioning, the so-called genetic algorithm is most suitable. In hardware/software partitioning genetic algorithms have been used in [61, 28, 63] 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 [61], 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 fulfil the constraints. While genetic algorithms can give good solutions to problems with many constraints and multi-dimensional cost 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.

3.1.5 Global Criticality/Local Phase Driven Algorithms

An extension to simple serial greedy partitioning is the global criticality/local phase driven algorithm [58]. 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, 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 [60] a more elaborate scheme based on GCLP is presented. This paper combines GCLP and implementation-bin selection into an iterative algorithm called MIBS (Mapping and Implementation-Bin Selection). 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. When the implementation is determined the tagged node becomes a fixed node, signaling a new iteration. The implementation-bin selection

```
var U: unscheduled nodes;
var S: scheduled nodes;
(* ready nodes are nodes whose
   predecessors have all been
   scheduled *)
var R: ready nodes;
function GCLPPart(system: graph): graph
  computeExtremities(system);
  computeRepellers(system);
  while not empty(U)
    ComputeGlobalCriticality(system);
    Determine R;
    Compute effective exec. time t_exec(i)
(* if i ∈ U: t_exec(i) = GC.t_(HW,i) + (1 − GC)t_(SW,i)
   else if i ∈ S: t_exec(i) = t_(part,i) *)
   for each node in R
     compute LongestPath(root, node);
   pick node from R with Max(longestPath);
   if isExtremity(node) then
     delta = node.bias_HWorSW;
   else if isRepeller(node) then
     delta = node.repelValue;
   else
     delta = 0
   end if
   if node.gc >= .5 + delta
     objective=speedobjective
   else
     objective=resourceobjective
   end if
   partition node so Min(objective)
   U = U − node;
   S = S + node;
  end while
  return system;
end function
```

Algorithm 3.5: Pseudo code of a GCLP algorithm for solving the partitioning problem, as found in [58]. Some functions have not been listed here for space reasons.

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

3.1.6 Dynamic Programming

```
function DPPart(system: graph): graph
  if hasMultipleOutputs(system)
  then
    add dummy output node;
    connect each ouput to dummy;
  end if
  nodes = reverseTopologicalSort(system);
  while not empty(nodes)
    node = pop(nodes);
    merge(node, system);
  end while
  node = output node;
  bestSolution = minimum latency solution for node;
  for each node in bestSolution
    put node in node.partition;
end function

function merge(node, DAG)
  predecessors = fan-in nodes of node
  for part in [HW,SW] do
    node.partition = part;
    for each pred in predecessors do
out:  for each solution in pred.solutions do
        Delay = solution.delay + delay_{pred->node};
        for each other predecessor do
          (* choose solution that has smaller
             delay than Delay and minimum
             area.*)
          if no solution found then
            continue out;
          other[i] = chosen solution;
        end for
        if sum(other[*].area) > maxarea then
          continue out;
        node.solutions.add(combine(other[*], node));
      end for
    end for
  end for
end function
```

Algorithm 3.6: Pseudo code of a Dynamic Programming algorithm for solving the partitioning problem, as found in [64].

When a problem has recursive characteristics and overlapping subproblems, dynamic programming can help find a solution. Several papers use dynamic programming in their partitioning [34, 64] strategies. For example, [64] describes how possible mappings of a node in a 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.

3.1.7 Binary Constraint Search

```
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 3.7: Pseudo code of a Binary Constraint Search algorithm for solving the partitioning problem, as found in [59].

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 [59] 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 [59] 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.

3.1.8 Clustering Algorithms

```
const maxsize;
var loophierarchy: graph;
function Cluster(root: node): clusters
  clusters = empty;
  if hasChildren(root) then
    for each child in children(root) do
      cluster = subtree(child);
      if cluster.size <= maxsize
        clusters.add(cluster);
      else
        clusters.add(Cluster(child));
    end for
  else
    return {root};
  end  if
  return clusters;
end function
```

Algorithm 3.8: Pseudo code of a Clustering algorithm for solving part of the partitioning problem, as can be found in [62].

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 [36]. 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, 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 [62], 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 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.

3.2 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 the estimation strategies discussed in Section 2 are essential to hardware software partitioning. Some papers on estimation even presented schemes tailored for partitioning schemes [28, 30, 25, 23, 18]. 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 3.1, it corresponds to the granularity levels mentioned in Section 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.

3.3 Dynamic versus Static Solutions

Most papers on hardware/software partitioning have focused on finding static partitions between hardware and software. With the advent of reconfigurable computing, 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 [62, 36, 67, 63] among others. In [62] 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 [36] 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 [67] 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 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 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 [63], 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 [63] 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 Reutilization
   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.

3.4 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 [68], 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.

Chapter 4

Conclusions

In this paper, I have reviewed high level estimation techniques and hardware software partitioning strategies, that might help create a model for hardware candidate selection in the Delft Workbench project. From literature, I found most high level estimation research has focused on area and speed metrics. Because the Delft Workbench tool-flow aims to optimize for speed under area constraints, these metrics may be usable in this project. After classifying the metrics in literature, several conclusions can be made:

- Dynamic versus Static
  When looking at high level estimation of hardware characteristics in literature, static estimation proved to be most common. Dynamic estimation is almost only used in speed estimation. When looking at software metrics, however, dynamic estimation is very common.

- Granularity of estimation
  Estimation dominantly uses entire graphs as the objects of estimation. However, for dynamic speed estimation functions and loops are also used.

- Datastructures
  From the papers discussed in this paper, it is clear that the most commonly used datastructure is the CDFG.

- Lack of error characterization
  Almost all metrics reported in this paper, have no quantitative notion of error.

- Memory impact
  Almost no paper I found dealed with the impact of memory behavior on hardware performance of candidate functions.

- Software metrics
  Papers on hardware estimation mainly focus on directly estimating

speed-up, area, or power. Almost no paper investigates different software metrics, like cyclometric complexity or maintainability.

Apart from estimation this paper has investigated hardware/software partitioning strategies. Concerning partitioning I found that many different aproaches exist (see Figure 3.1). The following conclusions can be made:

- Dynamic versus Static
  From the reviewed papers it is clear not much research has been performed in dynamic partitioning, i.e. partitioning different candidate regions on the same area at different intervals.

- Strategies
  Three main strategies seem to be used often in the research area: simple greedy approaches, Simulated Annealing, and evolutionary algorithms. The simple approaches mostly aim at generating results fast. The other two approaches try to prevent finding a local optimum by introducing randomness in a controlled fashion.

- Granularity of partitioning
  In contrast with estimation, the granularity of the different partitioning strategies is more diverse. In order to partition a program in a software and a hardware part, estimation strategies should work on the same level of granularity.

While this paper is based on a broad number of papers on the subjects of estimation and partitioning, it does have some deficiencies. First, the most recent works in this area have not been included. Because there are several initiatives currently working on similar projects, the knowledge is constantly changing and therefore this paper has limited itself to the research before 2005. In following research, more recent works could be considered as well. Second, because this paper was written within the Delft Workbench project, the focus was on speed and area. However, power, communication, and memory costs, become ever more important and further investigations could be made into these areas.

Bibliography

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

[2] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[3] 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(12):38–43, 1997.

[4] Aravind Dasu and Sethuraman Panchanathan. Reconfigurable media processing. *Parallel Comput.*, 28(7-8):1111–1139, 2002.

[5] Enrico Buracchini. The software radio concept. *IEEE Communications Magazine*, 38(9):138–143, September 2000.

[6] Wayne Wolf. Building the software radio. *Computer*, 38(3):87–89, 2005.

[7] M. Cummings and S. Haruyama. Fpga in the software radio. *IEEE Communications Magazine*, 37(2):108–112, February 1999.

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

[9] 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(7):134–141, 2003.

[10] David Andrews, Douglas Niehaus, and Peter Ashenden. Programming models for hybrid cpu/fpga chips. *Computer*, 37(1):118–120, 2004.

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

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

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

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

[15] Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, 53(11):1363–1375, 2004. Fellow-Stamatis Vassiliadis and Member-Stephan Wong and Member-Georgi Gaydadjiev and Member-Koen Bertels and Student Member-Georgi Kuzmanov.

[16] Delft workbench.

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

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

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

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

[21] Yang Qu and J.-P. Soininen. Estimating the utilization of embedded fpga co-processor. In *DSD '03: Proceedings of the Euromicro Symposium on Digital System Design*, pages 214–221, VTT Electron., Oulu, Finland, September 2003. IEEE Computer Society.

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

[23] Rolf L. Ernst Jörg Henkel. High-level estimation techniques for usage in hardware/software co-design. In *ASPDAC '98: Proceedings of the ASP-DAC'98.*

Asia and South Pacific Design Automation Conference, pages 353–360, Princeton, NJ, USA, February 1998. IEEE.

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

[25] 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. In *DATE '98: Proceedings of the Design, Automation and Test in Europe Conference*, pages 218–225, Madrid, Spain, February 1998. IEEE.

[26] Tae-Woo Kim and Hyunchul Shin. Hardware cost estimation techniques for c-level description. In *ICVC '99: Proceedings of the 6th International Conference on VLSI and CAD*, pages 85–88, Ansan-City, Kyunggi-Do, South Korea, 1999. IEEE.

[27] 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.

[28] 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(1):140–158, February 2001.

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

[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(1):92–104, 1999.

[31] Jo ao M. P. Cardoso and Pedro C. Diniz. *Modeling Loop Unrolling: Approaches and Open Issues*, volume 3133/2004 of *Lecture Notes in Computer Science*, page 224. Springer, July 2004.

[32] 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. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[33] 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.

[34] Axel Jantsch, Peeter Ellervee, Johny Öberg, and Ahmed Hemani. A case study on hardware/software partitioning. In *FCCM'94, Proceedings of the Workshop*

on FPGAs for Custom Computing Machines, pages 111–118, Royal Institute of Technology, Stockholm, Sweden, 1994. IEEE Computer Society Press.

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

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

[37] 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(7):189–198, 2003.

[38] Srivaths Ravi, Ganesh Lakshminarayana, and Niraj K. Jha. Removal of memory access bottlenecks for scheduling control-flow intensive behavioral descriptions. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 577–584, New York, NY, USA, 1998. ACM Press.

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

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

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

[42] Adam Kaplan, Philip Brisk, and Ryan Kastner. Data communication estimation and reduction for reconfigurable systems. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 616–621, New York, NY, USA, 2003. ACM Press.

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

[44] 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.

[45] Trishul M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 191–202, New York, NY, USA, 2001. ACM Press.

[46] 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(7):908–921, 2003.

[47] Jianwen Zhu and S. Calman. Context sensitive symbolic pointer analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(4):516–531, April 2005.

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

[49] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. pages 5–17, 2002.

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

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

[52] Frank Vahid. Procedure exlining: a transformation for improved system and behavioral synthesis. In *ISSS '95: Proceedings of the 8th international symposium on System synthesis*, pages 84–89, New York, NY, USA, 1995. ACM Press.

[53] 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. In *FCCM '01: Proceedings of the 9th Annual IEEE Symposium on field-Programmable Custom Computing Machines*, pages 209–218, Colorado State University, CO, USA, May 2001. IEEE Computer Society.

[54] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

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

[56] Zebu Peng and Krzysztof Kuchcinski. An algorithm for partitioning of application specific systems. Technical 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.

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

[58] Asawaree Kalavade and Edward A. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 42–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

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

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

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

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

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

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

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

[66] T. Bäck, U. Hammel, and H.-P. Schwefel. Evolutionary computation: comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17, April 1997.

[67] U. Nagaraj Shenoy, Alok Choudhary, and Prithviraj Banerjee. Symphany: A tool for automatic synthesis of parallel heterogeneous adaptive systems. Technical Report CPDC-TR-9903-002, Center for Parallel and Distributed Computing, Northwestern University, Evanston, IL, USA, March 1999.

[68] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, 1993.