Spring 1999

Hardware-software codesign in a high-level synthesis environment

Tamas L. Visegrady
University of New Hampshire, Durham

HARDWARE-SOFTWARE CODESIGN IN A HIGH-LEVEL SYNTHESIS ENVIRONMENT

BY

TAMÁS L. VISEGRÁDY

Master of Science, University of New Hampshire, 1998

DISSERTATION

Submitted to the University of New Hampshire
in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

in

Engineering

May 1999

This dissertation has been examined and approved.

[illegible]

Dissertation Co-Director, Dr. Péter Arató, Professor of Electrical Engineering

[illegible]

Dissertation Co-Director, Dr. Andrzej Rucinski, Professor of Electrical & Computer Engineering

[illegible]

Dr. Philip J. Hatcher
Professor of Computer Science

L G Kraft

Dr. L. Gordon Kraft
Professor of Electrical & Computer Engineering

[illegible]

Dr. John R. LaCourse
Professor of Electrical & Computer Engineering

[illegible]

Dr. John L. Pokoski
Professor of Electrical & Computer Engineering

May 12, 1999

Date

Dedication

Szüleimnek

Acknowledgments

The valuable contribution of others have made a large portion of this dissertation possible. I would like to express my thanks to (in alphabetical order) Erik Fischer, Amy Hatch, Pawel Nowakowski, Ferenc Tél, and Dr. Pilar de la Torre for their support. (Meaningful questions qualify as support.)

The work of countless volunteers has produced the tools used to create this dissertation. I am indebted to the members of the programming community who made it possible to finish this dissertation using solely free and open source software.

Last, but not least, the whole process would have been impossible without the continuous parental encouragement and support. Köszönöm.

Table of Contents

List of Tables

List of Figures

ABSTRACT

HARDWARE-SOFTWARE CODESIGN IN A HIGH-LEVEL SYNTHESIS ENVIRONMENT

by

Visegrády, Tamás L.
University of New Hampshire, May, 1999

Interfacing hardware-oriented *high-level synthesis* to *software development* is a computationally hard problem for which no general solution exists. Under special conditions, the hardware-software codesign (system-level synthesis) problem may be analyzed with traditional tools and efficient heuristics. This dissertation introduces a new alternative to the currently used heuristic methods. The new approach combines the results of top-down hardware development with existing basic hardware units (bottom-up libraries) and compiler generation tools. The optimization goal is to maximize operating frequency or minimize cost with reasonable tradeoffs in other properties.

The dissertation research provides a *unified approach* to hardware-software codesign. The improvements over previously existing design methodologies are presented in the framework of an academic CAD environment (*PIPE*). This CAD environment implements a sufficient subset of functions of commercial microelectronics CAD packages. The results may be generalized for other general-purpose algorithms or environments.

Reference benchmarks are used to validate the new approach. Most of the well-known benchmarks are based on discrete-time numerical simulations, digital filtering applications, and cryptography (an emerging field in benchmarking). As there is a need for high-performance applications, an additional requirement for this dissertation is to investigate *pipelined* hardware-software systems' performance and design methods. The results demonstrate that the quality of existing heuristics does not change in the enhanced, hardware-software environment.

Introduction

This dissertation covers the theoretical background and implementation of a ***system-level synthesis*** (SLS) process. The contributions of the dissertation (summarized under "Contribution", p. 5) are concentrated on a development process capable of synthesizing systems with communicating software and hardware submodules. In addition to the theoretical results, the improvements have been applied to an existing CAD tool.

The example used for demonstration is an engineering Computer-Aided Design (CAD) tool, *PIPE*. This CAD tool has been developed at BME, ***Budapesti Műszaki Egyetem*** (Technical University of Budapest, Hungary). The *PIPE* environment, a tool performing optimization for custom pipelined hardware systems, has been incapable of targeting mixed hardware-software structures in the past. Using the results of this dissertation, *PIPE* is capable of designing systems that operate as interconnected hardware and software-based modules. The extensions and *PIPE* together form the Hardware-Software *PIPE* (*HSPIPE*) framework, which is an excellent demonstration of the capabilities of the enhanced design process. The dissertation discusses the problems encountered in such extensions and feasible solutions to those problems.

Since the ***hardware-software codesign*** (HSCD) or ***system-level synthesis*** (SLS) process is applied to diverse areas of development, none of the currently used development methods is capable of optimizing every SLS design. Such is the case in this dissertation research as well; the results provide solutions to some of the most frequently encountered problem types. In addition, the results have been tested with a number of benchmarks, with remarkably different results based on benchmark type. The dissertation includes an analysis of both significant and negligible improvements as applications of the research, and draws reasonable conclusions.

The dissertation results have been applied to the design of finite impulse response (FIR) digital filters, a frequently encountered design example. The implementation produces significant potential improvements over traditional, purely hardware solutions. Indeed, describing FIR filters covers a much broader set of typical signal-processing problems [Kun82]. As an extreme negative example, the RC-5 encryption algorithm is presented, where no improvements have been achieved in the algorithmic design when compared with a heuristic, traditional design. Section 5.2 contains a detailed description of the failure and implications. The negative results in this case may be attributed to the design of the algorithm.

An overview of a mixed hardware-software development process is presented in Chapter 1 (" *"Basic Stages of Hardware-Software Codesign"*, p. 6). The design environment targeting multiple execution contests is described as an extension of the traditional hardware development cycle. Additional steps are required to describe the hardware-software target architecture. In most contemporary systems, the relative cost of inter-module communications is significantly higher than the cost of functional operations.

Chapter summarizes the steps of a systematic hardware-software codesign architecture in detail, and presents the problems encountered in each step. The primary purpose of the chapter is a general overview without exposing the reader to an unnecessary amount of detail. A more verbose, detailed description of problems and solutions is described in Chapter 2 (*"State-of-the-Art Techniques in Hardware-Software Codesign"*).

Since the current status of the *HSPIPE* project is mainly hardware-centric, the code generation tools of the hardware-software codesign framework are not as detailed as the hardware synthesis subsystem. Since the *HSPIPE* framework is designed to be completely modular, incorporating incremental changes in the software generation sections are expected to be relatively easy.

Literature survey forms most of Chapter 2 (*"State-of-the-Art Techniques in Hardware-Software Codesign"*, p. 43). Since each major step of the hardware-software codesign development cycle presents at least one NP-complete problem (clustering/partitioning, scheduling, allocation), effective hardware-software codesign methods must employ heuristic approximations to provide feasible, solutions. A summary of currently accepted approximations is presented in the chapter.

Chapter 2 contains a brief description of possibilities when selecting heuristics for each stage of hardware-software codesign. The chapter introduces some of the applicable definitions and properties common to state-of-the-art hardware-software codesign research. In addition to the immediately applicable results, the chapter contains an overview of the status of research at some of the most important institutions actively researching hardware-software codesign or related topics (University of California at Riverside and Berkeley, University of Washington (United States of America); ETH Zürich (Switzerland); Oxford University (Great Britain); TIMA (France)).

Chapter 3 is dedicated to the process model and the approximation algorithms that were found appropriate in the *HSPIPE* CAD system. The approximation algorithms are selected based on the layout of the *PIPE* design flow and were verified with practical applications. The chapter presents a high-level overview with justified choices.

Chapter 3 introduces the additional mathematical notations of the dissertation over the current state of hardware-software codesign research. The chapter also presents a formal description of the extended properties of *control-data-flow graphs* (CDFGs) in multiple-context environments. The concept of a CDFG is extended in a way which is compatible with state-of-the-art HLS design methodologies, yet incorporates the necessary extensions to handle the unique requirements of multiple-context environments.

A separate chapter is dedicated to implementation of the hardware-software codesign environment in the *PIPE* framework (*HSPIPE*) ("*Implementation*", Chapter 4, p. 110). The chapter contains a detailed description of *PIPE* internals required for understanding the data structures and the design process inside *PIPE*. The selected approximation algorithms have been implemented as modules in the *PIPE* design process, hence extending it to the full *HSPIPE* framework. Chapter 4 summarizes the choice of implementation language, describes the front end, the data transfers with internal representation, and the output formats.

The results of tests and performance analysis are presented as the conclusion chapter ("*Analysis*", Chapter 5, p. 119). This chapter contains three major benchmarks well-known from the high-level synthesis community. A digital filter application and a differential equation solver are used to demonstrate the results on classical benchmarks. In addition to the traditional benchmark applications, the performance of the *HSPIPE* algorithm has been evaluated on a cryptographic applications, RC-5 encryption. Encryption algorithms recently appeared as benchmarking applications for hardware, software, and mixed environment systems [Ele98].

Chapter 6 presents a brief summary of dissertation results. The chapter enumerates the experience gained from the benchmark applications, points out fields where the dissertation results contribute to applicable design flows, and describes the cases where no serious improvement has been observed. The analysis of the example applications covers the areas where performance is less than optimal.

The hardware-software codesign extension of the *PIPE* system offers several exciting possibilities of expansion. Some of the potential extensions are described in a dedicated chapter ("*Future Development*" Chapter 7, p. 131). Without investigating them in detail, the chapter enumerates some of the possible extensions, including, but not limited to, interfacing to an improved user interface (*Visual PIPE*), algorithmic enhancements to new architectures, support for multiprocessor systems, and reconfigurable structures.

Contribution

The contribution of the research to the field of hardware-software codesign may be summarized as follows:

1. The dissertation presents a model of a multiple-context environment as an extension of generally accepted system descriptions used in high-level synthesis. This extended model may be used to perform synthesis, optimization and simulation of such multiple-context environment systems using an improved set of notations.

   The extended notations of the multiple-context environment description are upwards compatible with currently used systems.

2. The dissertation prescribes a transformation for mapping multiple-context system specifications to system descriptions that may be processed by existing, purely hardware or software-based (*single-context*) optimization and synthesis tools without modification.

3. The dissertation demonstrates that the above transformation preserves the necessary information to properly simulate and optimize multiple-context designs in existing single-context tools, while retaining properties unique to multiple-context environments.

4. The dissertation demonstrates a sample application of the results on systems which may be considered typical target environments.

5. The dissertation describes a sample application that is a framework of heuristic blocks. The framework is capable of obtaining results comparable to that of state-of-the-art research, and it may be upgraded in a modular way.

Chapter 1

Basic Stages of Hardware-Software Codesign

This chapter presents an overview of a mixed hardware-software development process. The mixed environment process is described as an extension of the traditional hardware development cycle. In addition to an abstract description of high-level synthesis, the chapter summarizes the steps of a systematic hardware-software codesign process.

Hardware-software codesign (HSCD) is a set of structured, automated design methodologies that implement digital systems as communicating hardware and software modules.

System-level synthesis (SLS) is the generic name of engineering design processes where a system description is synthesized to final, production-ready implementation in an automated environment.

The combination of hardware performance and software flexibility offers increased freedom, possibly reduced development time, and an abstraction layer over purely hardware or entirely software applications. Since the dissertation contribution to hardware-software codesign creates fully functional systems with a generated (synthesized) set of software and hardware modules, the term *system-level synthesis* is applicable to our approach. The two names, hardware-software codesign and system-level synthesis, are used interchangeably in the dissertation. (The investigations do not discuss approaches that do not integrate of hardware and software.)

Property	Software (COTS)	FPGA	Semi-	Full-
			custom	ASIC
Speed	Low	Medium-high	High	Highest
Operation	Serial	Serial/Parallel	Parallel	Parallel
Flexibility	Variable	Medium	High	Highest
Design effort	Low	Medium	High	High
Update effort	Lowest	Low	High	High
Development cost	Low	Low-Medium	High	High
Development time	Lowest	Low-Medium	High	Highest
Reconfiguration time	Lowest	Low-Medium	High	Highest

Table 1.1: Trade-offs in software and hardware systems

The advantages and disadvantages of hardware and software environments are summarized in Table 1.1. Since hardware implementations of system-level synthesis designs are realized either in custom hardware or in programmable silicon, Table 1.1 describes both custom ICs (*ASIC*s) and configurable silicon (*FPGA*s). Software implementations are generally executed on off-the-shelf, general-purpose processors, and are treated as COTS (commercial, off-the-shelf) solutions for the dissertation investigations. The dissertation investigations do not attempt to cover the possibility of synthesizing processors matched to specific problems. (Our definition does not consider such designs as hardware-software codesign, since the instruction-specific processor solutions do not necessarily feature multiple execution contexts.)

1.1 Hardware

Understanding hardware-software codesign requires a description of the difference between hardware and software-based implementations of systems. This section presents a high-level overview of the properties of synthesized hardware solutions. The dissertation results are not tied to any particular hardware technology or fabrication. This section contains an overview of the hardware synthesis process in general.

While details of the hardware environment are important for efficient generation of hardware components, the dissertation approach hides unnecessary specifics from the designer by delegating them to lower levels of abstractions (as described in Chapter 3). In the test implementation used for experiments and benchmarking, an abstract model of hardware synthesis appears in a technology library file. The library describes implementation properties by extracting significant information from the target system. The parameters at the technology library level are silicon area, cost, and timing of elementary operations. Other details of the target technology are not used directly in the logic synthesis step.

The hardware component of multiple-context environment systems is usually entirely *application-specific hardware*, *semi-custom integrated circuits* or *FPGA-based*. Even if custom, semi-custom and programmable silicon may coexist in an embedded environment, the different behavior of FPGA and custom silicon devices makes it difficult to integrate them properly [HB95b]. The problem is caused by the FPGA internal timings being outside the control of the designer, as they are determined by the efficiency of the routing algorithm and supporting hardware [HBE94]. An important implication of the FPGA routing problems is that routing algorithms require knowledge of FPGA geometry and interconnect information, and this dependency reduces portability [BR96].

Some of the problems of non-deterministic FPGA behavior may be handled by very high-level descriptions, where the whole set of FPGA input-output ports may be treated as a design block. Practical results have been demonstrated at the University of California at Riverside by Dr. Frank Vahid, where a high-level partitioning approach (*functional partitioning*) is used to manipulate problem descriptions over the flow-graph level [Vah97b]. (High-level synthesis operates on a flow-graph system description, as described in Section 1.3 and in Section 1.5.) Functional partitioning approach has been successfully applied to several designs at the University of California. Functional partitioning requires an extensive knowledge base of previous designs. Lacking such an extensive design database, the dissertation results are limited to flow-graph level partitioning and optimization.

Since FPGA internal timings are usually functions of different physical and topological parameters instead of predetermined input specifications, they may not be tuned without additional rounds of iterative experiments. In fact, in some FPGA systems, even such performance tuning might be unavailable for designers. Practical FPGA applications indicate that most FPGAs available today also exhibit definitely non-deterministic pseudo-random routing behavior, and may produce extreme variations in measured system timings, even under controlled, identical design settings. Since the detailed investigation of FPGA timing phenomena is outside the range of this dissertation, unpredictable FPGA timings are treated as worst-case values for the purposes of timing. Such worst-case treatment is required for both scheduling and allocation, since both depend on exact timing information, and timing violations are not permitted.

Choosing between a custom and a programmable hardware subsystem is an important design choice at the start of the system-level synthesis process. Performance-critical applications are usually implemented in *full-custom hardware*, which offers the highest performance. *Semi-custom* implementations are more useful if the application requires either faster reconfiguration, or if the turnaround time offered by full-custom ASICs does not meet design criteria.

1.1.1 Full-custom hardware

Full-custom *application-specific integrated circuits* (ASICs) generally offer the highest performance for all possible solutions. Since ASICs are custom-designed for each application, they are the most flexible. The only practical limitations to the capabilities of custom hardware components are those of the manufacturing process and the available design tools. (The development of microelectronic design tools does not follow the pace of manufacturing improvements, resulting in a widening performance gap between available and fully utilized silicon area [VG99].) An entirely different type of constraint is system cost and development time, which tend to be much higher than the cost of partially or fully off-the-shelf solutions.

Since hardware devices operate in a parallel fashion, full-custom hardware solutions may parallelize a problem in a variety of ways. ASIC implementations are flexible enough to explore additional hardware-specific tradeoffs in performance optimization. The possible speed-increasing techniques include, but are not limited to, combinational circuits instead of sequential solutions (at the price of increased silicon area), massive parallelization by physical replication, and geometry optimization. CAD tool support is available for some of these techniques, while others must be applied in an iterative, manually assisted trial-and-error process. The design cycle of full-custom systems is generally much longer than programmable implementations. (When compared to gate-array or sea-of-gates designs (Section 1.1.2), full-custom VLSI differs only in the additional step of transistor-level layout design, and these differences are usually minor for general-purpose applications.) Implementation times are generally also higher for than for programmable devices.

Since custom hardware components are manufactured using a large number of custom fabrication steps, overall system costs are generally much higher than either FPGA-based or software solutions. One must note that custom hardware fixed costs are decreasing with an increase in production volume, even more than semi-custom and FPGA-based implementations.

For practical systems, turnaround time is decreased by relying on a set of available low-level modules, using them as bottom-up blocks in the top-down design process. By using well-known and tested module libraries, the necessary simulation, testing, and verification time may be decreased, which in turn may reduce the number of manufacturing iterations. In fact, simulation at extremely low levels (such as register-transfer or gate level) without having such libraries is infeasible in most practical systems [LLSV98, VG99]. (The same idea of standardization and subsystem is present in an organized way in the design of standard cell structures.)

1.1.2 Semi-custom hardware

Representing a higher level of abstraction than application-specific hardware, *semi-custom hardware* solutions are based on low-level primitives to describe hardware systems with higher level constructs. Typical representatives of semi-custom systems are *gate-arrays* and *standard-cells.*

Gate-arrays are hardware systems that are created by programmed interconnects between general-purpose, pre-fabricated transistors [WE93, p. 409]. The transistors themselves are manufactured before customization; identical wafers are reused in a wide range of gate-array designs. Customization prescribes contacts to wafers and the layout of metallization layers. Gate-arrays produce application-specific integrated circuits, but the reduced number of application-specific manufacturing steps decreases cost and fabrication time compared with full-custom designs. Base wafers, containing large regular arrays of unconnected transistors, may be reused in a wide variety of applications without modification, which increases production volume, and decreases the number of application-specific manufacturing steps and masks. The contemporary variant of gate-array technology, *sea-of-gates* (SOG), uses transistors in a two-dimensional transistor array layout. Such transistor arrays may contain hundreds of thousands of usable transistors in a single device.

Since gate-array and SOG designs create logic gates by connecting predefined and existing transistors, these technologies are synthesized at the transistor level. For practical purposes, efficient generation of gate-array or SOG designs must rely on a set of higher-level primitives built in a bottom-up fashion. Without access to these basic blocks, traditional top-down designs must descend to the transistor level in synthesis, which increases problem sizes considerably over designs at higher levels of abstraction.

Standard-cell hardware systems standardize architecture at a logic or function level [WE93, p. 413]. Reusing existing logic building blocks, hardware design attempts to partition the problem into subsystems matching already existing components. Basic building blocks (standard cells) are generally available for logic functionality up to the level of basic arithmetic units, comparators, datapath manipulation, registers, and memories. At the level of schematic capture, a top-down design may be matched against the set of available standard cells. Identified modules may then be directly implemented by reusing standard cell layouts. The whole system is placed and routed automatically; standard cell designs tend to show regularity in their layout.

Standard cells at a much higher level of abstraction are widely used in modern embedded systems in the form of *Intellectual Property* (IP) modules. Such IP blocks contain complete synthesizable subsystems, for example communication protocols or complete complex functional modules. The complexity of such IP blocks enables designers to completely automate the generation of hardware-software interfaces [ET98]. Since IP blocks are "basic" in the sense of very high-level system descriptions, they are extremely easy to integrate in specification-level synthesis [GDZ98, VG98].

Even if semi-custom hardware is synthesized from using more complex building blocks than full-custom hardware, semi-custom implementations are still not reprogrammable without redesign. Once committed to a particular design, the realized functionality may not be changed in a device without replacing it physically. Because of the lack of reprogram-

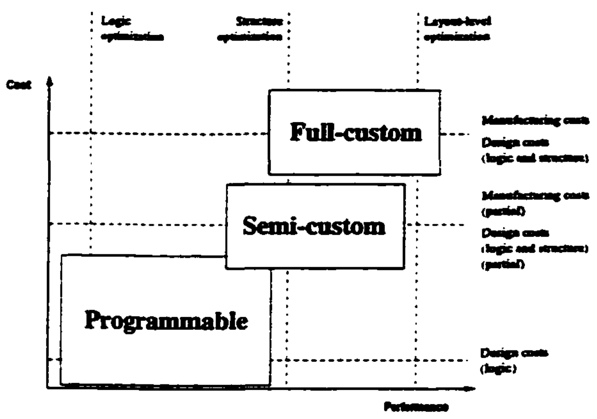
Logic optimization
Structure optimization
Layout-level optimization
Cost
Full-custom
Semi-custom
Programmable
Manufacturing costs
Design costs (logic and structure)
Manufacturing costs (partial)
Design costs (logic and structure) (partial)
Design costs (logic)
Performance

Figure 1-1: Relative performance and cost of hardware implementations

ming capability, semi-custom hardware implementations are considered to be similar to full-custom solutions for the purposes of this dissertation.

1.1.3 Programmable hardware (FPGA)

Considering systems which contain both software and hardware components, *programmable integrated circuits* are off-the-shelf hardware units that may be "reconfigured" at the structure level after fabrication. Programmable solutions generally select the desired functionality from a set of possible configurations rather than implementing functions from extremely low-level building blocks. Containing general-purpose, configurable logic blocks, reconfigurable hardware may be reprogrammed to perform different functions inside the same hardware device. Some programmable devices are *one-time programmable*, retaining one, immutable configuration indefinitely, while others may be repeatedly configured. Different FPGA types are used in different target environments, based on the relative importance of quick changes (fast reprogramming) or external device support (programming in the target environment).

As an example of one-time programmable devices, *antifuse FPGAs* contain their program in a set of switches (antifuses) which are turned into permanent short-circuits if the appropriate programming voltage is applied. *Altera* FPGAs are typical representatives of this programming method. Since such a permanently programmed device does not require

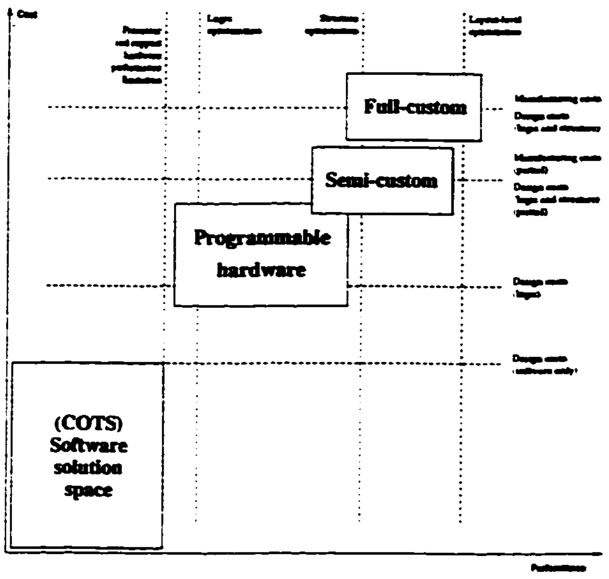
Cost
Logic optimization
Structure optimization
Layout-level optimization
Full-custom
Semi-custom
Programmable hardware
(COTS) Software solution space
Manufacturing costs
Design costs (logic and structure)
Manufacturing costs (partial)
Design costs (logic and structure (partial)
Design costs (logic)
Design costs (software only)
Performance

Figure 1-2: Comparison of software and hardware performance and cost

additional components for program storage, the amount of support logic is decreased. In addition to the self-contained nature of permanently programmed devices, they do not have programming overhead in usage (i.e., the initialization sequence of such systems is shorter than soft-programmable solutions, where configuration is supplied by an external device). Because of the permanent programming, replacing or upgrading such devices requires replacing the FPGA itself, which requires physical access to the system.

Static RAM-based (soft-programmable) FPGAs, which contain interconnect configuration in local memory, may be repeatedly programmed. FPGAs from *Xilinx* FPGA families are the most important soft-programmable devices today. Xilinx FPGAs are two-dimensional matrices of *configurable logic blocks*, CLBs, where CLB logic functions and interconnects between CLBs are set by RAM storage inside the FPGA. CLBs are individually programmable, basic FPGA functional units, capable of implementing small amounts of memory as several small look-up tables, combinatorial logic, small multiplexer blocks, or combinations thereof.

The properties of soft-programmable FPGAs are well suited to the requirements of system prototyping and are used extensively for rapid prototyping and development. Since the programs of soft FPGAs are stored in RAM, these devices have an initial overhead when the system is started. In addition to the programming overhead, static RAM-based FPGAs require external components to store the FPGA configuration , which must supplied from custom-programmed (E)EPROM memory or through a serial connection.

As programmable hardware components may be used for different purposes without modification, manufacturing volume of silicon in FPGA production may be higher than full-custom ASICs. The increase in volume numbers lowers the cost of programmable hardware, if compared with full-custom or semi-custom ASICs; FPGA solutions are available for single-unit (prototype) or extremely low volume systems at a cost of several hundred dollars per unit (as of May, 1999). For comparison, the manufacturing costs of full-custom ASICs are an order of magnitude higher *for every mask* in the manufacturing process; the first prototype of a full-custom hardware solution may cost several hundred times more than one in programmable hardware.

As programmable hardware is simply using predefined logic blocks, it requires significantly larger silicon area than equivalent functionality in full-custom circuitry. In addition, because of the inherent overhead in programmable devices, FPGAs are generally slower than their full-custom equivalents. Other important limitations of FPGAs include, but are not limited to, inadequate support for asynchronous operation, different performance metrics imposed by the fixed structure, and difficulties of estimating system performance.

A brief summary of FPGA performance metrics is presented in [VH98]. [Pag95] also provides an overview of current limitations of reconfigurable logic devices and development tool support. An overview of alternate problem description methods is available in [LSVS98]; some of the transformations in the article are used in FPGA designs to transform high-level problem descriptions to a representation which is more appropriate to the target FPGA architecture. To separate the dissertation results from quickly changing hardware parameters of FPGA implementations, the above changes are not modeled at the level of *multiple-context high-level synthesis* (MCHLS).

As mentioned before, reprogrammable hardware may be customized faster than the development of equivalent full-custom devices. Turnaround times are typically measured in hours instead of weeks or months (as for full-custom devices), or even minutes in case of minor changes. The reduced development time makes programmable devices more attractive for prototyping and small-volume systems. Since the available gate count is smaller in programmable devices than in ASICs of the same silicon area, FPGA-based designs may spread to multiple FPGAs if necessary. Timing problems of such an environment may be much more serious than single-package solutions, and due to the complexity of the topic they are not within the scope of this dissertation. Some of the multi-FPGA problems are summarized briefly in [HB95b]; similar problems are present in the routing issues and the related performance variance.

1.2 Software

Software modules of mixed hardware-software systems are traditionally compiled binaries executed on a separate, dedicated microprocessor. In addition to the functional code executing the subtask, software modules must contain interface code to synchronize program execution with hardware [JRV+98, TV97].

Software development is generally faster than most hardware design cycles. Compiled code (an executable *binary*) is typically executed on a hardware environment which is (preferably) standardized. Software development for an initially known, fully (or sufficiently) specified, ideal hardware environment is possible using a virtual environment that replicates the ideal working hardware. (In fact, the development of universal virtual environments is a field of active research [Knu99].)

Most embedded systems feature emulators and development tools for a virtual, ideal hardware environment, which enables software to be developed based on simulations. Since development in such a virtual environment does not have to rely on the status of potentially buggy or incomplete hardware, software may be developed at the same time [Bro95]. A similar approach is used in practical hardware-software codesign environments, where an idealized boundary is created between the hardware and software subsystems [LLSV98, Kal95, BS98, Ros98]. The interactions on the simulated boundary may be monitored during the design process, and the system response may be compared with system specifications.

While software development may be significantly faster than creating the equivalent hardware system, there are serious limitations of traditional embedded software environments. Similar to hardware systems, software development is possible at multiple layers of abstraction. High-level programming languages usually offer faster development than lower-level languages, while low-level languages are typically much more efficient. (As an exception, in *Reduced Instruction Set Computer* (RISC) systems, programming at a very low level is not always more effective.) Regardless of the development language, the functionality of the generated executables is bounded by the capabilities of both the programming language and the development environment.

Software, even if bounded by the development environment, is a useful tool for mixed hardware-software system designs. Since software is easier to modify than hardware, and more portable for different platforms, software may serve as a useful interface layer in embedded applications. By layering software and separating most of the application from the internals of the hardware environment, a reusable and portable application may be created. Because of the continuous improvements in the underlying hardware systems, even inefficient software may be used with a reasonable performance level if hardware support enables the system to reconfigure the software environment on-line [VH98].

Even if modern operating systems and programming languages offer support for multitasking and multithreaded execution, typical embedded systems may not be able to exploit such features. Because of this limitation, most hardware-software codesign systems are executing code in a traditional, entirely sequential fashion. Unlike hardware, such executed code may not automatically take advantage of parallelism. In a multiprocessor environment, where processors may execute different instruction streams (i.e., a *Multiple-instruction-multiple-data* (MIMD) architecture [HX98, Qui94]), software parallelism is possible and should not be inhibited by the applied MCHLS heuristics. By applying heuristics that do not imply an entirely sequential execution, without unnecessary reduction of degrees of freedom, the dissertation results may easily be extended to the synthesis of multiprocessor embedded systems (Chapter 7, "Future development").

This dissertation covers some implementation details and suitable models of such multiprocessor systems, but does not include detailed descriptions of an example in such a system. Utilizing the simulation results and building such a multiprocessor system based on these dissertation results is outside the range of the dissertation investigations. Most of the necessary extensions are covered in Chapter 7 ("Future developments", p. 131).

1.3 Partitioning

In a multiple-context environment system, selecting an efficient combination of hardware and software subsystems may be formulated as a special case of the ***partitioning problem.*** This step of the design process assigns design subsystems to hardware and software environments, creating partitions of the set of functional blocks (such as subsystems in the system block diagram).

Execution contexts are the largest possible subsystems in the target environment where direct communication is possible between components in the same context. Communication is considered to be direct if it may be realized using entirely combinatorial (i.e., stateless) logic.

All communication between operations in the different contexts must pass through a *context-switch.*

The execution context of elementary operation e_i is x_i. The set of all execution contexts is X.

In an example taken from GSM speech compression in Figure 1-3 (replicated from Figure 3-11, p. 104), there are two execution contexts. The hardware context ($x_i = 1$) contains vertices 1, 2, 8, 9, 13, 14, 15, and 16. The software context ($x_i = 2$) contains vertices 3, 4, 5, 6, 7, 10, 11, 12, 17, and 18. Obviously, $X = \{1, 2\}$

Context switch data connections (CSDCs) are edges (e_i, e_j) in the system flow-graph where a context switch occurs between e_i and e_j. Using set notation, the CSDC set W is defined as:

$$W = \{e_i \rightarrow e_j : x_i \neq x_j\}$$

Multiple-Context Environments (MCEs) are design target systems where functional units are mapped to multiple *execution contexts.*

Figure 1-3 shows a sample MCE, where parts of the GSM speech compression algorithm are implemented in software, while others are in hardware. In this case, there are two execution contexts.

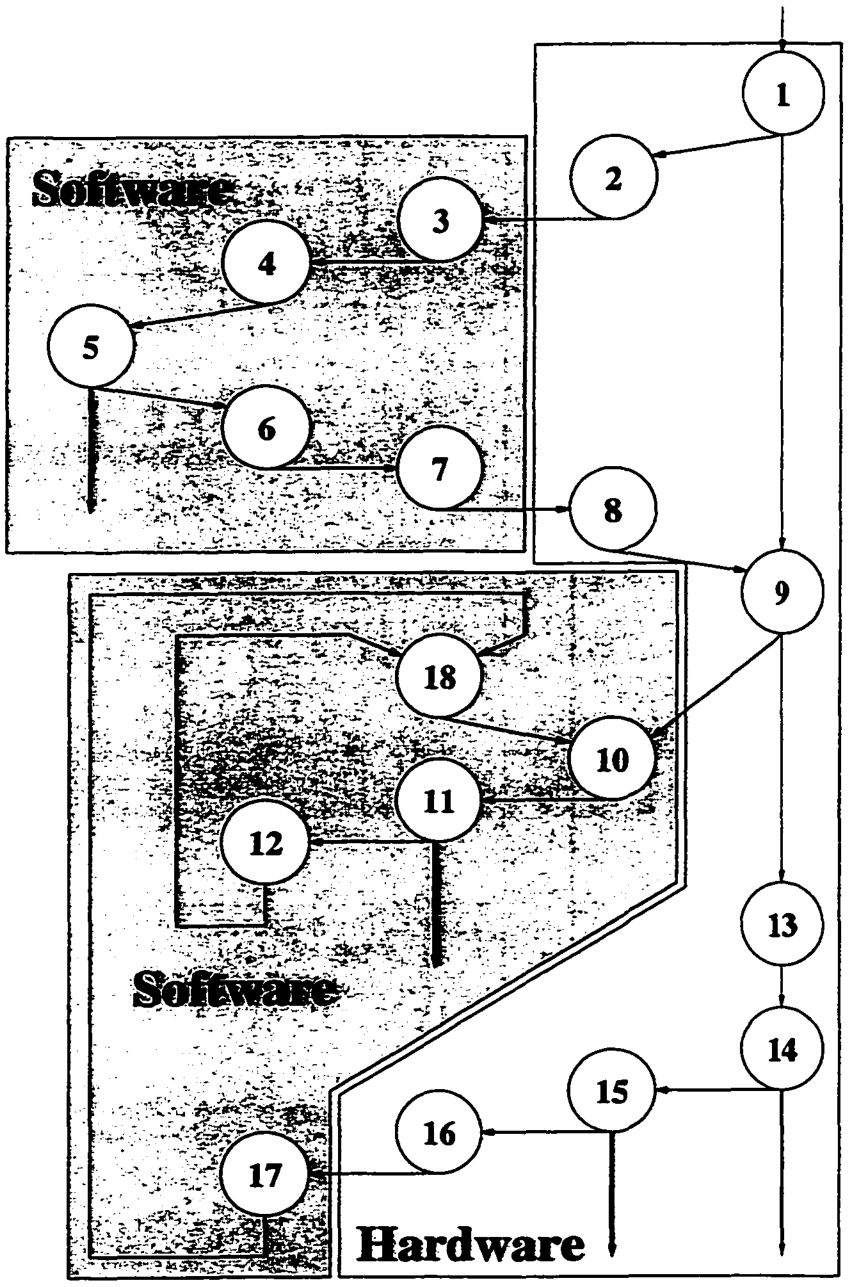

Figure 1-3: Extended elementary operation graph of partitioned GSM example

Flow-graphs are graph-based description of problems, formulated as a set of vertices (V) describing operations and a set of ***directed edges*** (E) describing direct dependencies between operations.

Control-data-flow graph (CDFG) is a special flow-graph that carries both control and data information. Attributes related to data transfers are contained in direct data dependencies (edges) while control information is present in global properties of the graph, such as sequences of edges. (The details of such control information are described in Chapter 3)

Elementary operations are functional operations that may be realized directly with one register-transfer level primitive (i.e., a single element of the underlying technology library).

The number of elementary operations is denoted with n.

Context switch occurs between elementary operations e_i and e_j if and only if they are in different execution contexts ($x_i \neq x_j$) and a direct data connection exists between e_i and e_j.

In the example system (Figure 3-11, p. 104) context switches occur during the following data transfers: $e_2 \rightarrow e_3$, $e_7 \rightarrow e_8$, $e_9 \rightarrow e_{10}$, and $e_{16} \rightarrow e_{17}$.

Context switch complexity ($n_{i,j}$) is the significant size of context switch $e_i \rightarrow e_j$ in the system cost function. In most systems, the complexity of the context switch is a monotonically increasing function of the number of bits in the data transfer.

Context switch weight ($w_{i,j}$) is the weight factor of context switches in the system cost function. The weight depends on the source and destination contexts, and the complexity of the context switch:

$$w_{i,j} = w_{i,j}(x_i, x_j, n_{i,j})$$

Context boundary (set) of an execution context C (B_C) is the set of context-switch edges such that one of the edges is in C and one is in $\overline{C} = V - C$. Obviously, every CSDC must be an element of at least one context boundary:

$$W = \bigcup_{C \in X} B_C$$

In graph terms, a context boundary is a cut of the system flow-graph. In the above example, since there are exactly two execution contexts, the context boundary contains all the context switch edges ($e_2 \rightarrow e_3$, $e_7 \rightarrow e_8$, $e_9 \rightarrow e_{10}$, and $e_{16} \rightarrow e_{17}$).

The partitioning problem in hardware-software codesign is finding *partitions* of the set of data-flow graph vertices such that the total cost of edges connecting vertices in different partitions, defined by a suitable cost metric, is minimal.

Note that the above formulation of the partitioning problem does not attempt to minimize the difference between the number of vertices in execution contexts [Las93, p. 3]. The definition does not inhibit non-binary partitioning, where the target environment contains more than two execution contexts. Because of this relaxed requirement, the dissertation results may easily be applied to systems beyond traditional single-processor hardware-software codesign (such as multiple microprocessors).

The *binary partitioning problem*, most often encountered in hardware-software codesign, attempts to find a set of edges in a graph that, when removed, separates the graph into *two* components. The cost of a solution is taken as a function of edge weights and component distribution in the result. Since the distribution of vertices does not implicitly influence the cost of a solution, cost function in MCHLS is based entirely on the cost of edges in the partition cut, as shown later.

Finding the partitions with the minimum cost in a system, even as a *binary (two-context) partitioning problem* is NP-complete [Hoc97, Chapter 5, p. 192] [GJ79]. As the the partitioning problem is not tractable, several heuristic approximations have emerged. Literature divides partitioning heuristics into two definitely distinct groups, differentiating between *global* or *construction algorithms* and *local* or *improvement algorithms* [Las93, p. 4]. Construction algorithms are used for generating a partition based on performance metrics, and are generally applied in a non-iterative fashion. Improvement algorithms use an already existing system partition as a starting point and attempt to enhance system properties by applying incremental changes to it. Most generally used partitioning approaches combine the two solutions by selecting construction and improvement heuristics that complement each other [HB95a, KL97, VNG97].

Given the extensive literature and research on partitioning, partitioning heuristics must be selected based on the target application, since some of the algorithms are limited to the problem category where they were developed. As an example, partitioning research in supercomputing differs from hardware-software codesign because of an additional requirement in supercomputing, *balanced load* [YW94]. (In addition to minimizing communication costs, as the case of hardware-software codesign partitioning, supercomputing also attempts to minimize overall calculation time, which is bounded below by the processing time of the slowest computing vertex.) The increase of partitioning computational requirements because of the need for balancing, makes supercomputing-derived heuristics inefficient in hardware-software codesign.

Similar to the inefficiency of heuristics targeting balanced partitioning, *geometry-based partitioning techniques* (such as *line bisection*, [Las93, p. 15]) are generally not useful for hardware-software codesign partitioning. In geometry-based heuristics, edge cuts are made based on geometry-related information, and solutions are investigated as functions of angles and graph layouts. Most geometry-based partitioning heuristics are derived from systems where communication costs are directly related to physical system layout. These geometry-based heuristics are applicable where data transfers are affected by physical placement, number of hops and other related parameters. There is very limited support, for cost functions based on abstract properties (such as penalties for wide data transfers) in geometry-derived heuristics. Since the partitioning process relies on abstract cost functions of partition cuts, and has practically no relation between geometry and system data-flow graph (DFG) layout, most of the geometry-oriented partitioning algorithms have to be excluded from the dissertation investigations.

Several of the most popular, practically used partition improvement techniques rely on the incremental *Kernighan-Lin* algorithm [KL70], or, more precisely, a particular extension of it, the *Fiduccia-Mattheyses* algorithm [FM82]. Both algorithms take an initial partition as input, and attempt to refine it by relocating vertices between partitions. Both algorithms are capable of converging to local optima [Vah97a] since they terminate when no local change improves the current cost value. Several extensions attempt to increase algorithm robustness around local extreme values, but no generally useful solution has been found. In fact, even some of the most popular practical hardware-software codesign design tools, such as VULCAN [Gup93] has been observed to terminate in oscillation because of convergence problems around local extreme values [Knu95, p. 31, "Previous approaches"].

1.4 Clustering

In some hardware-software codesign environments, graph transformations are applied to the system description before the partitioning step to reduce the size of the solution space during the partitioning step. Such reduction of the solution space is feasible if there are groups of elementary operations that should not be delegated to different execution contexts. By inhibiting the partitioning process from separating such elementary operations, the problem size of the partitioning problem may decrease considerably. Even if the problem formulation may be similar to partitioning this process is part of the design for a slightly different reason than the partitioning problem.

The primary effect of graph transformations before partitioning is to create groups of locally connected vertices. The best results are usually achieved when the groups are highly connected; finding highly connected subgraphs in a graph is a computationally expensive problem with extensive coverage in literature [Hoc97, Chapter 6, p. 236]. As opposed to partitioning, where global metrics are important, local grouping serves a different purpose, and this additional step has a separate name, *clustering*.

The clustering process prescribes vertex groups to be assigned to the same partition (execution context); such vertex groups are called *clusters*. Forming clusters reduces the number of possible context switches. and so reduces the runtime of the partitioning step.

Creating clusters of locally connected vertices improves the performance of partitioning in several ways. As well as reducing the number of vertices, clustering vertices that are "close" to each other may inhibit unnecessary steps and local minima of the cost function by inhibiting vertex movement inside groups.

Since the cost of a multiple-context environment implementation is related to partition cuts, elementary operations connected by wide data transfers are "close". Such elementary operations should be placed in the same execution context so that data transfers do not have to cross execution context boundaries.

As well as clustering "close" vertices before partitioning, a closeness heuristic may be used after partitioning to merge partitioned clusters to implementation blocks, i.e., FPGAs or separate subsystems [VG95a]. Since closeness metrics and the corresponding theory are most useful in functional specifications and the necessary high level of abstraction, the dissertation research does not consider applying closeness-based partitioning extensions to the design process. Should the algorithm be extended to handle functional specifications of the incoming high-level description, closeness-based cost functions could become valuable additions to the partitioning process.

For established hardware-software codesign environments, where a large database of standard modules is available, an efficient way of clustering before partitioning is the identification of vertex groups that could be easily implemented in an already existing module. Identifying such vertex groups is a computationally expensive problem with a number of alternative heuristic methods. The existence of available modules is generally tied to a certain hardware environment, and not easily portable without encapsulation at a higher level of abstraction. Because of the lack of useful algorithms for this encapsulation, the necessary hardware detail may not be represented in the design process at the level of the dissertation research.

1.5 High-level synthesis

High-level synthesis, the process of transforming abstract (high-level) hardware descriptions into silicon, is widely practiced by the CAD community.

High-Level Synthesis (HLS) The automated design of an advantageous register-transfer level (RTL) description of a system from an abstract high-level description.

Register-Transfer Level (RTL) An intermediate-level description of structures, defined in terms of storage (registers), elementary functional units, and interconnects between storage elements, with the necessary control logic.

Practical applications of high-level synthesis (HLS), do not exist for hardware-software environments as of today. The traditional HLS process transforms high-level (data-flow graph) descriptions to an intermediate-level, connection-oriented description of the design (register-transfer level description). An alternative to terminating at an intermediate description is called "silicon compilation" [Gaj88]. In silicon compilation, high-level system descriptions are directly transformed to transistor or layout-level. Such a direct, one-step synthesis process offers significant savings in time, but usually at the expense of silicon area. A direct transformation from a high-level description to silicon offers some unique advantages (such as testability and verification), but without substantial heuristic support, the silicon-related costs may be too high for practical usage (especially in large systems).

This dissertation relies on the underlying module generators and compilers and proceeds to an intermediate abstraction level, utilizing external, specialized tools to synthesize the register-transfer level description. The heuristics of RTL synthesis are different from those used in HLS and are not discussed in this dissertation. Since the locality of registers affects the performance of the partitioning process [Hea93, Chapter 7, p. 43, "Variable-Register Communications"], heuristics in MCHLS have to address the issue of storage optimization.

The dissertation research attempts to reduce designer degrees of freedom as late as possible. By keeping more design paths open, the chances of generating a suboptimal problem because of converging to a local optimum are reduced considerably. (One must note that this "safe" approach increases the time complexity of the design process considerably.) There are practical hardware-software codesign environments, where multiple-context environment systems are designed in a different way. Some practical codesign approaches, for example, are limited to certain (fixed) ratios of hardware and software, presenting a limited choice of useful designs to the designer without exploring large sections of the design space [AJ97].

While traditional hardware-related HLS is a well-known process, it is practically incapable of handling problem relocation between hard and soft computing environments. One of the primary reasons for the lack of multiple-context environment capabilities is the unpredictable timing of software systems, especially in RISC environments [Ker93, SW95, Cor96]. Since most practical HLS algorithms rely on fixed timings, they are unable to easily address the unique problems of multiple-context environment systems. For the same reasons, well-known HLS algorithms are incapable of designing for FPGAs, except for worst-case approximations. Worst-case FPGA designs are usually inefficient because of the high variance of routing-related FPGA parameters. This might change if the current trend continues in the accuracy of FPGA routers and simulators. (The predictability of some FPGA families has shown a long-term tendency of increasing accuracy.)

To extend a HLS process for the broader HSCD environment, clustering and partitioning extensions must be appended to the HLS design process. The iterative flow of the HLS design process is then extended with the additional iterative rounds of clustering and partitioning (Figure 3-2, p. 77). To keep the optimization time complexity low, the clustering and partitioning steps must be controlled with fast feedback possibilities to discard a solution that does not meet performance criteria before executing the computationally expensive tasks of scheduling and allocation. The extended HLS design process, with the external partitioning, clustering, and feedback system, is referred to as *multiple-context high-level synthesis.*

Multiple-Context High-Level Synthesis (MCHLS) is an algorithmic extension of a traditional High-Level Synthesis design process, capable of synthesizing systems with multiple execution contexts.

Even if the MCHLS extends the target environment of a HLS design process to more complex architectures, it may rely on the established HLS heuristics by using a set of notations upwards compatible with those of HLS.

Since the HLS process requires a graph-based problem description as input (either a *data-flow graph*, DFG, or a *control-flow graph*, CFG), a suitable description should be able to embed partitioning information in the graph description. There are several well-known ways of transforming high-level descriptions from natural and programming languages to DFGs, and our implementation does not deal with the details of generating the system DFG. (In fact, in most established HLS processes, designer freedom is present only in the generation of the system DFG, since further steps are automated and proceed without designer interaction.) In the implemented MCHLS system, the front end supplying a high-level description such as *SpecCharts* [VNG95] or VHDL [AV98] is *not* considered to be an internal part of the MCHLS design environment.

The two primary stages of HLS (*scheduling* and *register allocation*) require initial knowledge of placement (partition information), so execution context information must be at least partially included in one of the graphs. Both data-flow graphs and control-flow graphs are capable of conveying context information, the former with vertex attributes, the latter with control information for context switches. Chapter 3 presents a solution for describing execution context information in a system CDFG.

By applying a suitable transformation and embedding context switches as protocol delays in the CDFG, one retains the accumulated heuristic knowledge of scheduling and allocation methods, since the data model of elementary operations is practically unchanged compared with HLS (the negligible differences are discussed in Section 3.2). The chosen transformation preserves partitioning information without loss of significant information. As a disadvantage, the algorithm increases the number of vertices in the system, since context switches are represented as individual vertices instead of properties of edges.

The HLS process developed at BME (Technical University of Budapest, Hungary) is based on a *control-data-flow graph* or CDFG. This flow-graph combines the information content of traditional DFGs and control-flow graphs by embedding control information in graph vertices. Such a combined solution is useful since a single graph is used during synthesis, as the necessary control information is extracted only during the last stages of the design process. Since the available BME module library consists of data-oriented modules and source code, a natural approach is to design the data-flow in the system, and supply the necessary control information only later. The data-oriented usage of the CDFG enables designers to generate control information after completing all initial steps on data operations. A number of control-oriented boundary conditions are applicable to data-oriented synthesis; these conditions are covered in greater detail in Chapter 3. Control-oriented boundary conditions represent physical and organizational constraints of hardware components in the control circuitry.

This dissertation assumes that the HLS process terminates at the *register-transfer* level (RTL). In other words, dissertation investigations do not target the complete design process (silicon compilation [Gaj88]), rather they concentrate on the abstract representation of lower-level steps. This way the system may use optimized *low-level* hardware primitives and reusable software modules while retaining the design freedom of a complete design process. In performance-critical problems, the RTL description may further be optimized to improve performance; time-to-market and lower personnel costs are becoming more and more important than resource utilization or subsystem performance [Wir98]. BME (Technical University of Budapest, Hungary) already has an extensive library of proven hardware primitives, and the *PIPE* CAD system relies on these libraries. The code generator and the external (auxiliary) software module library of *PIPE* are currently (as of May 11, 1999) incapable of producing industrial-quality results. (The requirements of industrial and academic CAD tools are definitely different, as outlined in [Fuh91].)

1.6 Register-transfer level synthesis

Following the steps of allocation and scheduling (Figure 4-1), a hardware system is often expressed as a connection network of registers, multiplexers, primitive arithmetic units (ALUs), basic logic functions, and the necessary control logic (Figure 1-8). All control information, including timing and direct data dependencies, is included in this register-transfer level (RTL) description. RTL descriptions offer much lower flexibility than the different flow-graph representations since design alternatives or different schedules are not represented in the transfer-level description at all. The step of translating RTL descriptions to fabrication input is referred to as "RTL synthesis".

Since the level of abstraction at the RTL level is very low, efficient RTL synthesis algorithms have been devised based on advances in hardware characterization and compiler research. Most practical approaches rely on an extensive set of standard cells and perform the RTL synthesis step by selecting the standard cell presenting the closest match to the desired submodule. For further discussion of RTL synthesis, the following references are recommended: [Jha95, DK91].

1.7 Code generation and compilation

After covering hardware synthesis steps in the previous section, this section presents an overview of the code generation and compilation steps. The dissertation assumes that compilation is handled by compilers optimized for the target architecture, and such compilers are treated as a black box system. The only requirement for the compiler is that it should generate executables (or object files, if linking is a separate step) that may be downloaded to the off-the-shelf microprocessor and executed there. The investigations do not model synthesis for application-specific processors. By delegating the task of compilation to an external application, the portability of dissertation results is increased considerably. Note that, even if the compiler is treated as a black box, certain optimizations may be performed before submitting code to the compiler. In the case of the dissertation results, as shown later, the efficiency of the generated software subsystems is approximated without actual compilation.

Similarly to HLS problems, most existing *software compilers* are unable to handle relocation from software to hardware in a flexible way. Pure software-based high-level synthesis attempts to optimize software for a given environment. Optimizing RISC compilers are the most important representatives of this approach. Quite similar to RISC compilation, *very long instruction width* (VLIW) computers, with their extreme dependence on the target environment, present a very good example of software tied to a given architecture. For our investigations, it is assumed that our system takes a high-level description in the form of flow graphs programs and finishes at an intermediate level, by producing optimized assembly source code as an output. An additional front-end step is assumed to generate data-flow descriptions from higher-level descriptions, such as C or Java source code.

Partitioning of the compilation process to two passes (to register-transfer and then binary level) makes it possible to have complete control over the efficiency of the code, yet perform the optimizations at a relatively high-level, leaving the mundane tasks of final compilation and linking to the (system-dependent) assembler. A disadvantage of this solution is that a processor model of the target environment is required at higher levels of abstraction. Describing processor internals becomes more and more difficult as designers merge solutions from different environments to increase performance. Practical examples show that processor emulation for profiling purposes may become a serious complexity issue with modern processors, since the software behavior may be tracked only with increasingly complex finite state automata [BR95].

Target environments may be treated as pure transfer-cost systems for a quick model of execution profiling. High-level features, such as *multiprocessor interprocess communication* or *shared memory handling* become difficult to capture at this level (at least without the necessary bottom-up libraries) so this dissertation investigation targets a minimalist execution environment. An example of target architecture is an FPGA-based or microprocessor environment interfacing to local memory and peripherals in addition to the custom or semi-custom hardware components. It is assumed that compilers perform effective optimizations for the target architecture. RISC compilation *of fixed DFGs* is an NP-complete problem where the optimal solution may be known beforehand, or at least an accurate estimation may be given [OKD97]. Since there are available, well-known approximation algorithms for both scheduling and allocation [Hoc97, p. 1, p. 94], bounds may be set limiting the optimal solution in polynomial time without actually finding it. Finding a solution sufficiently close to the optimum makes it possible to exit initial iterations without useless rounds of optimization.

Software compilation differs from pure hardware HLS in the placement of scheduling and allocation. Software instruction scheduling and register allocation are usually performed in the same phase, as opposed to hardware designs, where allocation generally happens after scheduling. A possible reason is that inserting an additional register to compensate for over-utilization is easier in hardware and may be performed without further iterations. In software over-using the register set may be solved by code transformations (namely, using *spill code* [ASU88, p. 542]) and requires feedback in the design process.

Merging instruction scheduling and register allocation provides immediate feedback for problematic scheduling decisions and may reduce the cost of further iterations. The disadvantage of combined scheduling and allocation is the increased solution space of several simultaneous NP-complete problems. A major advantage is the fact that the conflicts between scheduling and allocation are evaluated in single phase of optimization [NP98]. Practical implementations use separate, specialized algorithms for scheduling and allocation, and select algorithms that may use status information from the other stage [NP95b].

Another approach to integrate scheduling and allocation is presented in [NP98]. By splitting scheduling in two and performing allocation between the two schedulers, the second scheduler pass may compensate for the effect of any spill code introduced by the allocator.

Since the different design processes must be matched somewhat in the mixed hardware-software solution space, a simplification must be made to make the extreme (i.e., purely hardware or software) cases compatible. For this reason, the hardware section shall be based on mixed scheduling-allocation algorithms (which are also present in HLS). A summary of combined algorithms is presented in [PK89]. Combined scheduling-allocation methods in HLS generally perform scheduling with initial monitoring of allocation so that the stages are done in the usual order (scheduling first, allocation second) but the allocation process is guaranteed to give the expected results and so no further iteration is necessary. This approach is identical to the pure software solution.

1.8 System-level synthesis process and classification

Initially performed manually, the HSCD development cycle of today's systems may not be finished manually in feasible time. For the purposes of this dissertation, HSCD is used solely as a term for ***structured, automated*** designs. The dissertation extends the theoretical background and a sample implementation of a high-level synthesis environment, and this extended design process is called Multiple-Context High-Level Synthesis.

HSCD methodologies are inherently more complicated than single-environment (i.e., purely hardware or software) designs. The problems include, but are not limited to, the additional communication requirements on the boundary of hardware and software as well as the more difficult testing in a mixed environment [BS98]. Even before testing, the simulation environment of a hardware-software codesign environment must be adapted to the special requirements of the multiple-context environment [LLSV98, Ros98].

Several of the design steps unique to HSCD are NP-complete. These computationally hard optimization problems must be solved in addition to the NP-complete problems already present in high-performance software development or high-level logic synthesis. The most important computationally expensive problems in HLS and software performance op-

```
A = A + S[0];
B = B + S[1];
for (i=1; i<=r; i++) {
        A = ((A ^ B) << B) + S[2*i];
        B = ((B ^ A) << A) + S[2*i+1];
}
```

Figure 1-4: Code of RC-5 main loop

timization are efficient *register allocation* and *instruction scheduling*). (For a brief summary of useful approximation algorithms for scheduling and graph coloring problems, see Chapter 1 and Chapter 5 of [Hoc97], and Chapter 23 of [CLR90]. An overview of popular approximation algorithms is given in Chapter 2.)

The design of connected hardware and software subsystems requires at least two additional design steps before scheduling and allocation are performed. *Clustering*, the selection of elementary operations, creates groups of operations which are to be executed in the same environment, regardless of the hardware-software boundaries. *Partitioning*, setting the boundaries of hardware and software, follows clustering and attempts to generate an efficient division of execution contexts.

Starting from a high-level description of the problem, the system must be formulated as a *flow-graph*. There is a possibility of using several flow-graphs during the initial phases of hardware-software codesign (control-flow (Figure 1-6), data-flow, or control-data-flow). There are slightly different heuristics to be used based on which one is chosen as an input. (Note that in most environments, assuming the necessary information about the target environment is available, all of these three descriptions may be converted to the other two representations.)

In most HLS systems the input flow-graphs are assumed to be immutable and they are not changed except minor details, such as inserting buffers (delay). Even if practical systems exist where the flow-graph description may be modified by the synthesis process (such as functional design processes [VG92]), the dissertation investigations do not attempt to perform functional optimization. The task of generating a suitable flow-graph description from an even higher level description is delegated to the system front end (Section 4.2,

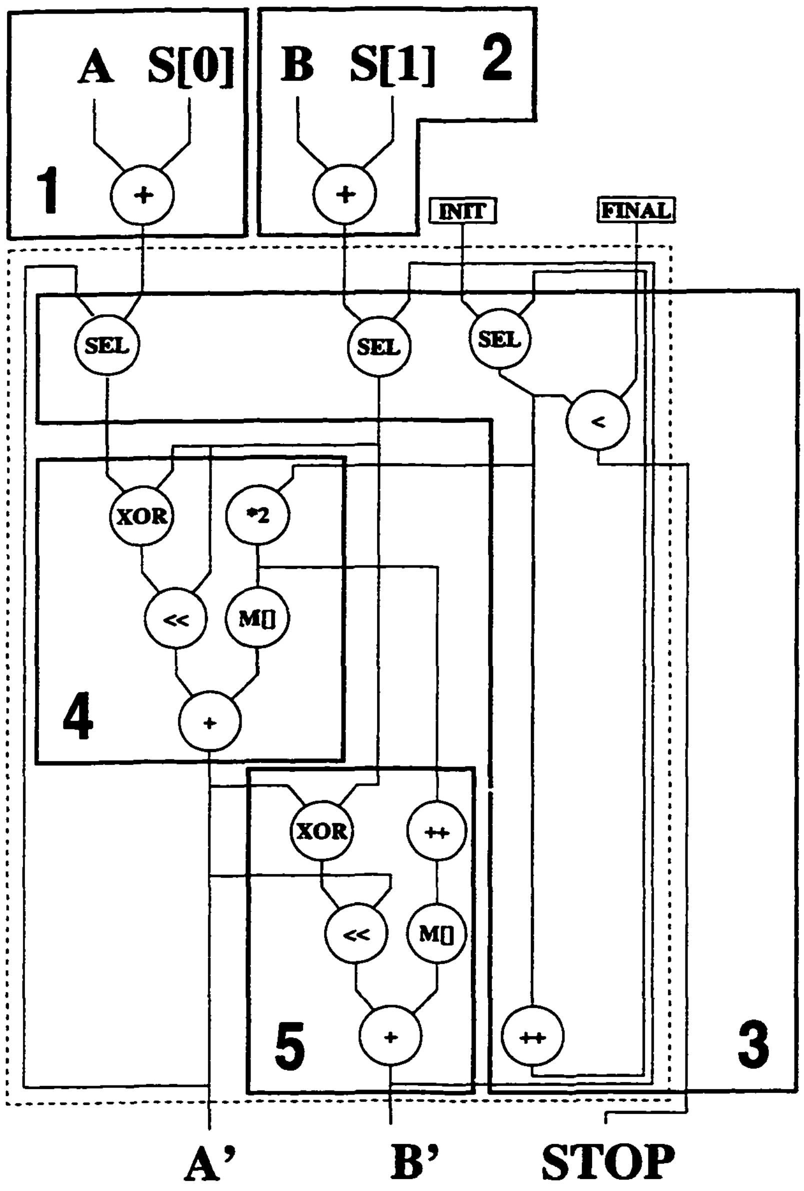

Figure 1-5: Generated data-flow graph of RC-5 main loop with source line numbers

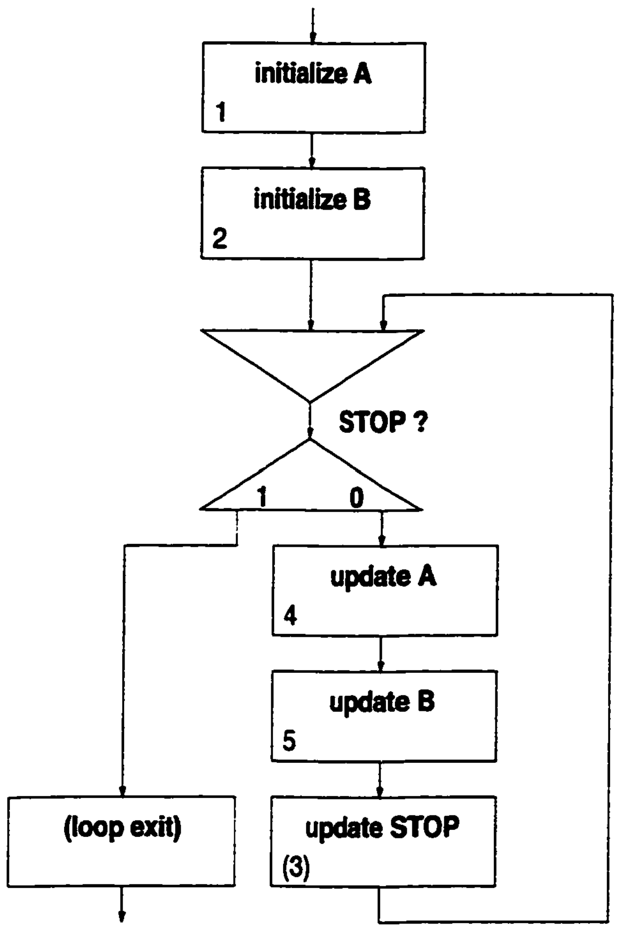

Figure 1-6: Coarse control-flow graph of RC-5 main loop

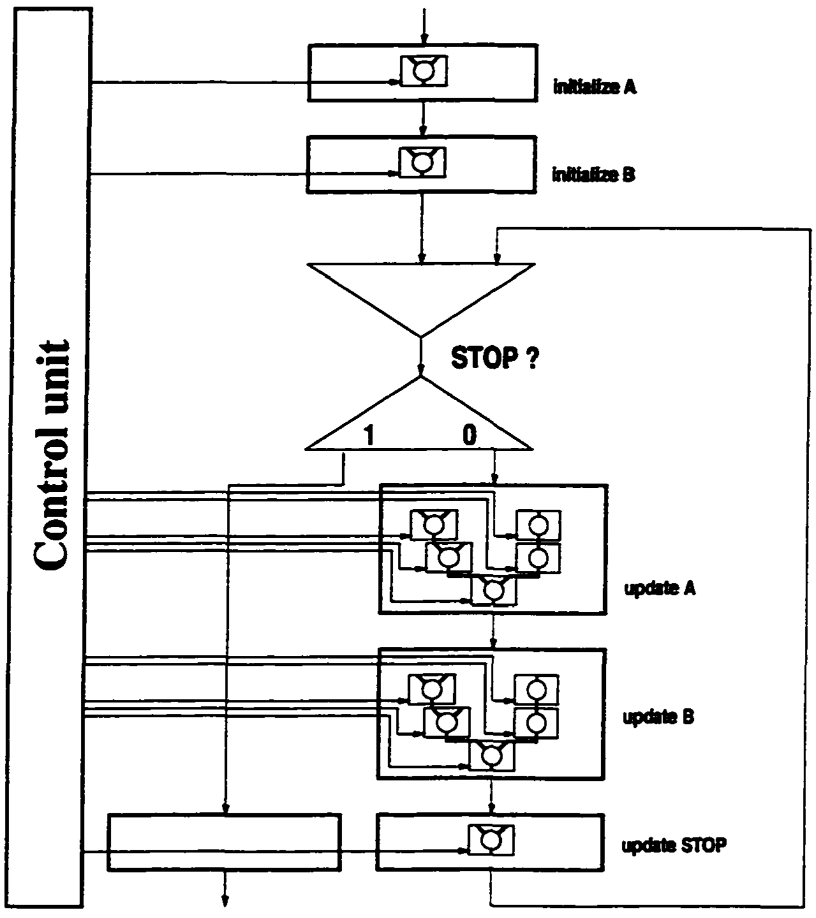

Figure 1-7: Control information of RC-5 main loop

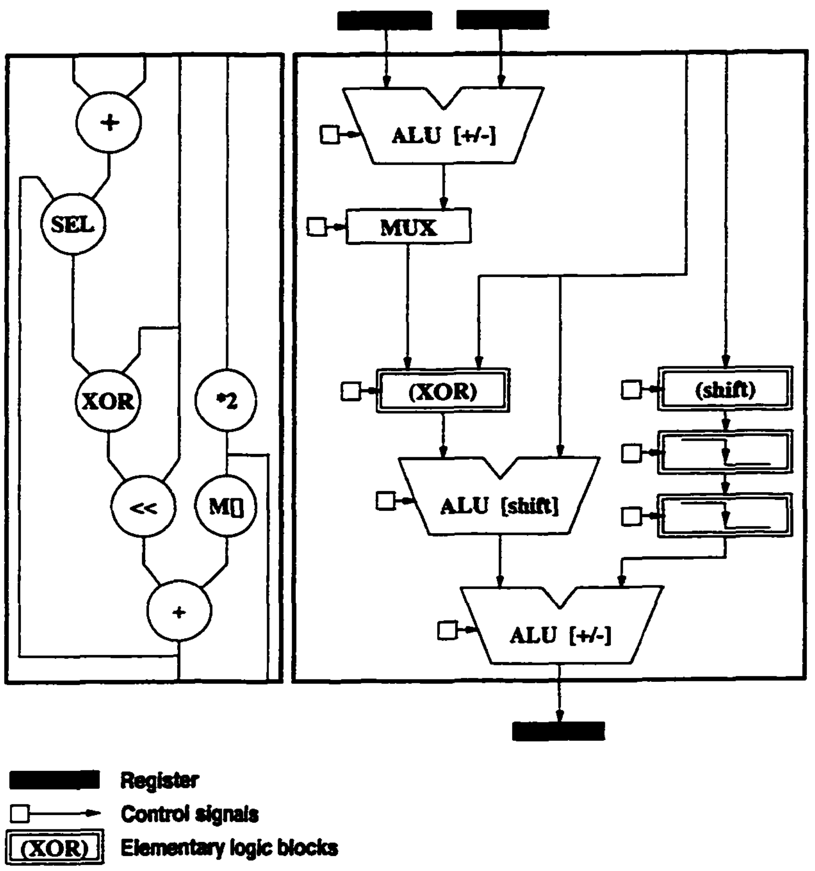

Figure 1-8: Section of a register-transfer level description and the equivalent data-flow

p. 115). As an example algorithm of generating graphs for regular FIR filter structures, see Appendix A, p. 154. An example code loop is presented in Figure 1-4 with the corresponding annotated ***data-flow graph*** in Figure 1-5 showing source lines ("`M[]`" denotes a memory access). Note that in the data-flow graph, loop control is maintained in a STOP signal generated by a comparison operation.

Flow-graph system descriptions may follow the ***control flow*** inside the system (***control-flow graph***, CFG) (Figure 1-6, Figure 1-7), or concentrate on the data-flow (***data-flow graph***, DFG) (Figure 1-5). Control-flow-based system descriptions use control transfers to model data propagation, showing the relations between the operating times execution units.

A combined version of the data-flow graphs called ***control-data flow graphs*** (CDFGs), is frequently used in high-level synthesis. CDFGs describe the system with an improved data-flow graph. CDFGs contain additional information over DFGs so that the necessary control information may be extracted from the data description.

To stay compatible with the currently available *PIPE* knowledge base, this dissertation uses a CDFG to describe the input problems, and generates the control structures once the register-transfer level (RTL) description is available (such as in Figure 1-8).

Currently existing industrial design systems do not offer satisfactory support for mixed hardware and software environments. The most widely used approaches fall into one of the ***bottom-up*** or ***minimum solution*** approximation methods. While both approximations are based on practical systems and have found useful applications, no successful integrated environment has implemented a completely flexible design tool with manageable computational requirements. One of the goals of this dissertation is to present a solution capable of considering a much wider set of possible solutions than restricted approaches. The different approaches to hardware-software codesign may be summarized in a way similar to that presented in Figure 1-9.

Using the notation of Figure 1-9, one may classify different system-level synthesis approaches based on their progress in the design triangle. Starting from a unique system description at the top of the design space, the design process proceeds to the bottom of the triangle as the level of abstraction decreases. The two extreme solutions are software and hardware, on two ends of the solution space. Note that at the level of data-flow graph no

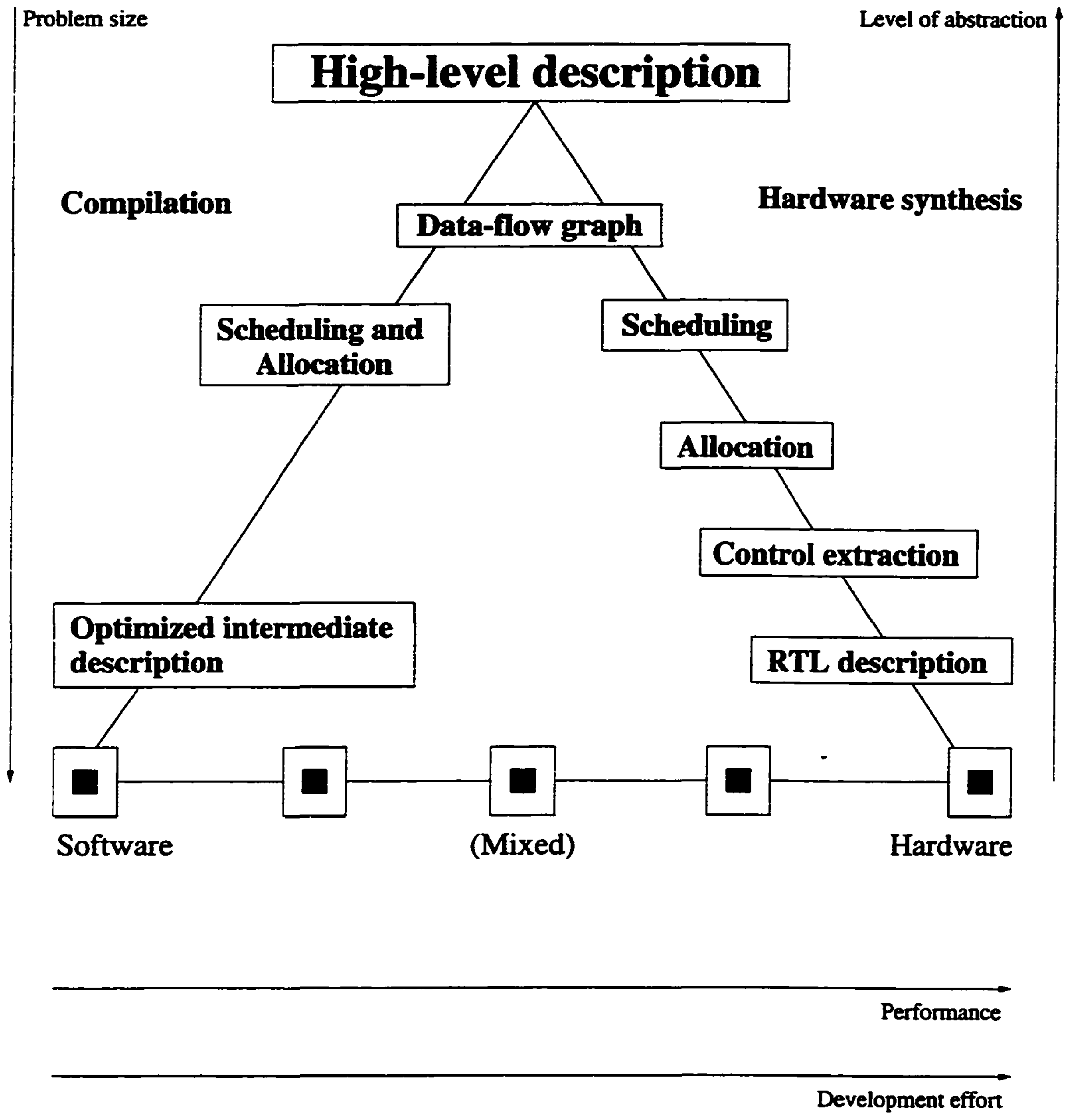

Figure 1-9: Hardware and software development with codesign solution space

decision has been made of implementation context, and therefore the data-flow graph as a starting point of the design is considered to be a immutable. The design process then passes through the system search space (the triangle in Figure 1-9) and terminates in a completely specified solution (i.e., in a point on the base of the triangle). In such a specified solution, there is no more designer freedom left, apart from the choices in implementation.

Entirely software solutions proceed through instruction scheduling and register allocation to a source code level, reaching the base of the triangle in Figure 1-9 while staying on the extreme left of the triangle (i.e., maintaining an entirely software system). Software synthesis therefore traverses trajectory "a" in Figure 1-10.

Similarly to software synthesis, hardware solutions from the same data-flow graph proceed through the stages of scheduling and allocation, arriving to a RTL description (entirely in hardware). This solution is represented on the right of the solution space in Figure 1-9, since none of the intermediate steps have software components, therefore the design process passes through trajectory "b" in Figure 1-10.

Some of the currently existing hardware-software codesign design methodologies attempt to reuse optimized structures during the design of multiple-context environment systems [ABIC+98, AJ97]. Depending on the available design database, the number of possible solutions is limited, which narrows down design search space to a number of discrete points. Using low-level, optimized structures, the design process is practically evaluating the performance of each possible implementation and selecting one with a reasonable cost, as illustrated in Figure 1-10.c. Further optimization is usually not necessary, since design reuse already presents submodules (such as the case of *hard IP (Intellectual Property)* macros). This approach is definitely very fast, since the steps of optimization are omitted, and no NP-complete problems have to be solved.

A possible disadvantage of the reuse of already existing designs is the limited number of solutions. Even if the selection process evaluates several possible solutions on the lowest level (i.e., implementation), more efficient systems between existing solutions are not found. As the cost functions of complex hardware-software codesign systems tend to be far from smooth, local extremes may appear in unexpected points in the solution space (i.e., the base of the triangle in Figure 1-9). Evaluating a set of discrete solutions samples the system cost function at the implementation level, and may lead to suboptimal results if the sampling is inefficient.

Another possible approach to the design of hardware-software codesign systems is the *minimum solution* solution. This approach, a variant of *greedy approximation algorithms*, starts with an extreme partition and refines it through local extremes until it meets design targets or it has exhausted improvement possibilities. The optimization terminates correctly when the predefined performance criterion is met, otherwise, the greedy solution does not converge.

Two possible solutions are especially popular initial configurations in state-of-the-art hardware-software codesign research. Since a large number of multiple-context environment systems are constrained primarily by implementation cost or execution time (while other performance metrics are of secondary importance), following a greedy heuristic may produce good results.

Starting from a purely software implementation, and transforming the critical path to hardware, results in a system that fulfills timing constraints (if possible). Relocating the DFG critical path to hardware (fully or partially) decreases system latency.

Critical path is a set of vertices and edges in a DFG which form the execution path with the highest total latency.

There may be multiple critical paths in the system, if more than one route has the same (maximal) total latency.

Since the critical path may change because of moving vertices to hardware, the new critical path (or paths) must be found after every iteration, finishing as soon as latency constraints are met. Also, since relocating functionality between execution contexts introduces context-switch vertices to the system DFG, additional iterative rounds are needed to verify the system meets design constraints. Each round traverses a trajectory similar to that depicted in Figure 1-11 (trajectory "b"), selecting the software solution in most steps, with a limited number of subsystems where a hardware solution is desired.

Similarly to initially software solutions, starting from a pure hardware solution and relocating vertices to a software context, hardware resource requirements may be reduced to the cost limit set by external constraints. The problems of this approach are identical to those of the hardware-based refinement algorithm.

A different classification of hardware-software codesign is possible if the system specification (the data-flow graph) is assumed to be flexible. Most lower-level optimization approaches (including HLS and software optimization) treat the system specification at the DFG level to be immutable. Should the design process explore different data-flow graphs implementing the same functionality, a different solution space could be obtained, with probably different results. This higher-level approach, manipulating problems at a very high level of abstraction, is behind functional-level designs [VG95a, GVNG98, VNG95]. (As an example of different data-flow graphs derived from the same higher-level description, see Appendix A). The very high level steps of functional-level design explore different data-flow graphs generated from the same functional description, which is demonstrated in trajectories denoted with "a" in Figure 1-11.

Similarly to traditional hardware and software development, top-down design is popular in hardware-software codesign. Unlike most of the previously described methods, bottom-up solutions rely on available building blocks (reusable submodules) when making decisions, but the building blocks are not tied to specific architectures and may be optimized to the target system. Since the currently used submodules are usually complex systems themselves, the system is usually modeled at a coarse resolution, which reduces the size of search space considerably. As the submodules are not optimized at an implementation level (such as soft *IP (Intellectual Property) blocks*), the system may be still optimized after selecting an initial solution. This process is illustrated in Figure 1-11.c., where a higher-level solution is selected first and then it's optimized to match the requirements of the target architecture.

One of the problems of the coarse bottom-up solution is that it may quantize the solution space very early, i.e., represent it as a small set of discrete solutions. As problem sizes are increasing, a discrete solution space may be inappropriate for a truly effective solution, which might be described by a finer resolution model only. Since the selection may exclude the global optimum, the partitioning phase of HSCD should try to evaluate as much of the solution space as possible. Another related problem is that hardware-related systems may not have smooth cost functions. These problems are common with that of discretizing the solution space at a lower-level solution (see above).

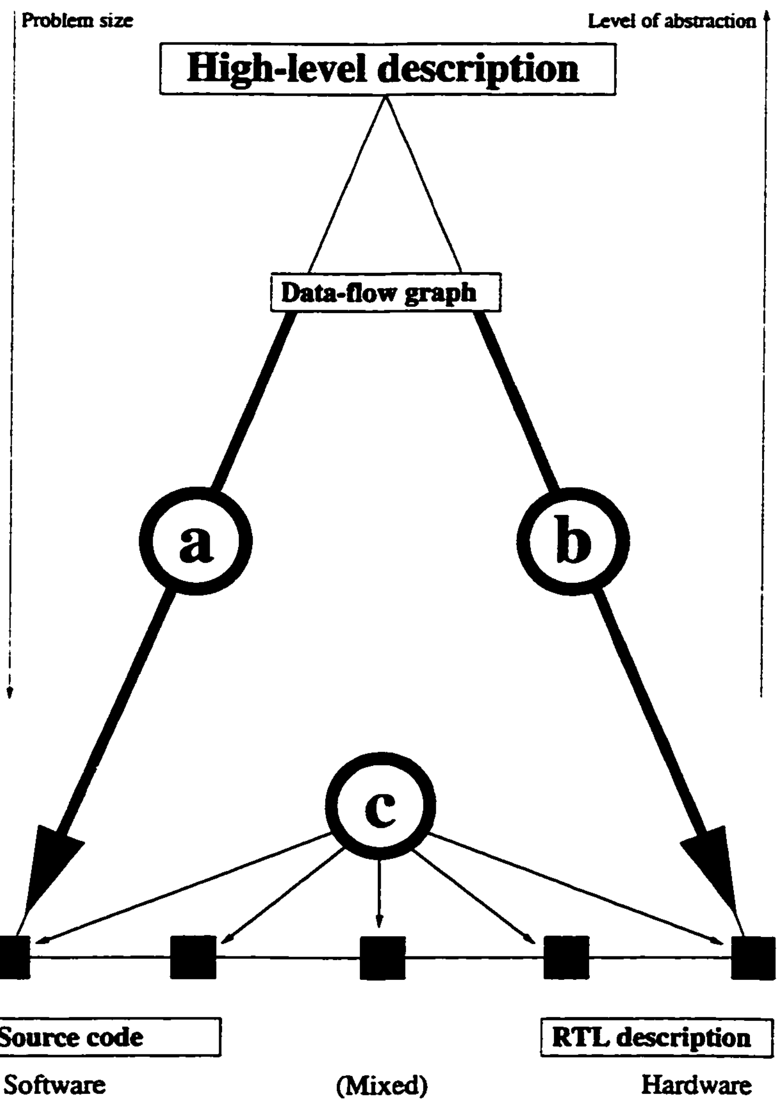

Figure 1-10: (a) Software development, (b) hardware synthesis, (c) synthesis of fixed-ratio systems

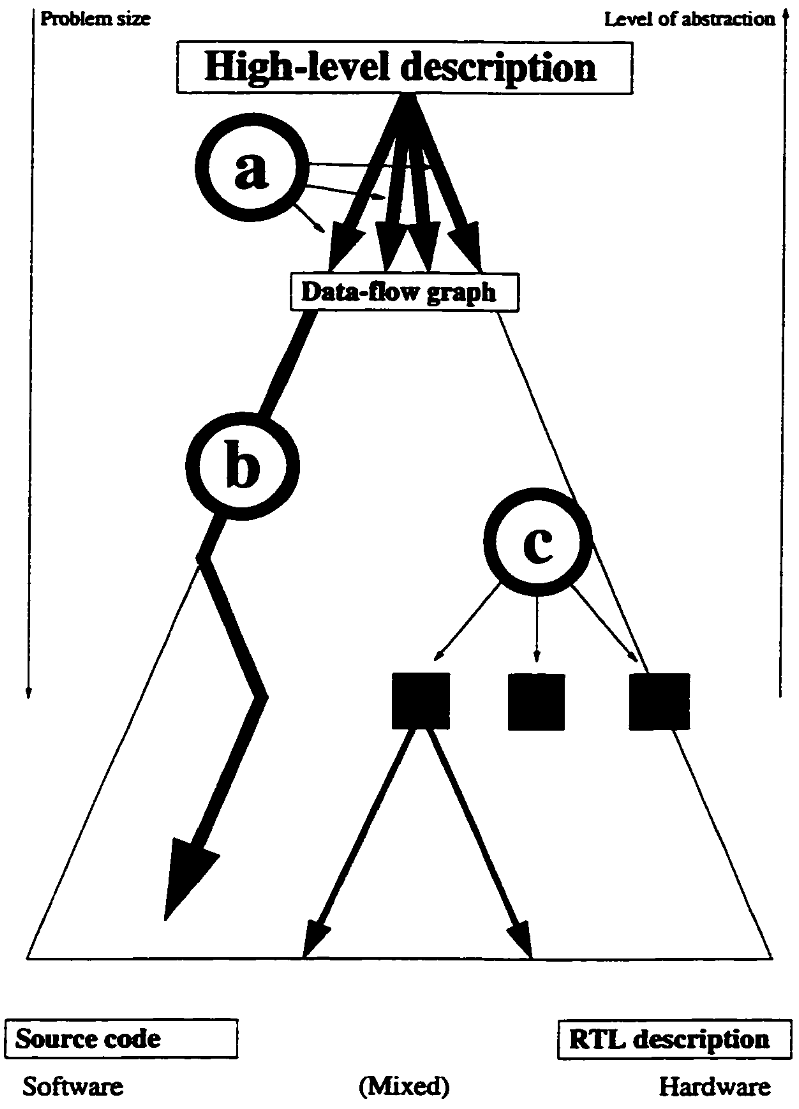

Figure 1-11: (a) Functional-level designs, (b) iterative refinement, (c) hardware synthesis with bottom-up submodules

Chapter 2

State-of-the-Art Techniques in Hardware-Software Codesign

Since each major step of the hardware-software codesign development cycle presents at least one NP-complete problem, effective hardware-software codesign methods must employ heuristic approximations or approximation algorithms to solve them. A summary of currently accepted approximation algorithms is presented in this chapter, including the heuristic background. The approximation algorithms are also classified based on the results of researchers working on hardware-software codesign (in addition to the purely mathematical coverage). Because of the unique requirements of the hardware-software codesign design environment, some of the frequently used approximations are not directly applicable to multiple-context environments. This chapter contains information about methods of limited usefulness as well. Including heuristics that have been proven to be less efficient in optimizing benchmarks is reasonable, since the same heuristics may be much more useful in different types of applications.

Initial research in heuristic research tried to interface software systems with full-custom or semi-custom microelectronics (standard cells or full-custom VLSI). The high cost and development time of these VLSI systems made it difficult to use hardware-software codesign as an efficient solution to practical problems. Initial attempts at codesign were also based on design steps that reduced the available designer freedom too early, in the partitioning stage. Attempting to partition without feedback from later stages often results in a very inefficient solution [Knu95, "1.3.1. The traditional approach to codesign", p. 7]. Also since the synthesis steps following the initial partitioning steps are computationally expensive heuristics for NP-complete problems, increasing the number of full iterations may increase the total synthesis time to an unacceptable level.

Advances in Field-Programmable Gate-Arrays (FPGAs) have made it possible to use reprogrammable hardware systems with external software support as reconfigurable building blocks [Pag95]. In addition to FPGAs, the increasing number of embedded systems has made it feasible to design heavily optimized, application-specific embedded systems using hardware-software codesign methods [Pin96, VH98, ETT98]. Since embedded systems are produced in large (and increasing) numbers, a small decrease of price or a slight increase of the price/performance ratio vs. increased development time and cost is feasible in such devices. Research of the *Programming Research Group* at the Computing Laboratory of Oxford University, among others, even targets creating semi-custom processors. Semi-custom microprocessors are compiled from a software-level description of a given program and are optimized for the instruction set required to execute that program [Pag94]. Creating a set of microprocessors optimized for a given task is a feasible and promising way of combining hardware performance with software flexibility. This solution does not strictly belong under "hardware-software codesign", since the execution context does not change during the execution of programs.

To accommodate the different environment of hardware-software systems, purely hardware and software optimization techniques must be used to generate subsystems with the required performance and cost properties. Before optimizing disjoint hardware and software modules, the designer must partition the system, i.e., decide which submodule to implement in hardware and which in software. An inefficient partition may result in an increase in communication overhead that eliminates the potential advantages of a mixed hardware-software solution.

Most hardware-software codesign systems are too large to handle with currently available algorithms when modeled at the elementary operation level. For the purposes of elementary operation models, each vertex in the data-flow graph is a basic, atomic operation, performed either in hardware or software. State-of-the-art high-performance heuristics for data-flow schedule optimizations have a time complexity of up to $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$ with linear space complexity. Using even a quadratic algorithm with a practical system would be too expensive since a typical telecommunications application, for example, may be translated to thousands of basic blocks if every elementary operation is treated as a basic block.

To reduce the number of basic blocks, most hardware-software codesign systems collapse nearby elementary operations to blocks of higher complexity, thus reducing the size of the problem. This process, *clustering*, is based on available heuristics for finding submodules that may be optimized.

After the clustering process, there is a clear distinction between *global* and *local* optimization, as the inside of basic blocks is left intact in the system-level (global) optimization process. Basic blocks are later optimized independently of each other (i.e., locally). Most optimization steps after clustering do not extend optimization to span multiple basic blocks. For this reason, selecting the proper basic block boundaries is a critical decision at this step, and the applied heuristic should definitely be matched to the target application or environment.

Selecting clusters sets the number of basic blocks partitioning must deal with. Since partitioning has a nonlinearly increasing complexity, clustering must find a balance between system granularity and design time. Also, since the clusters are used to predict the design decisions before actual scheduling and allocation, the choice of clusters should make it possible to give a useful estimation of the outcome. As the optimization steps differ depending on the selection of optimization algorithms, clustering cost functions must take this into consideration.

Optimizing the distribution of software and hardware basic blocks is called the *partitioning problem* as it attempts to find an efficient partition of the basic block set. Selecting the optimal partition requires a set of boundary conditions which is based on design criteria as well as technology parameters. Partitioning is an NP-complete problem, where polynomial-order heuristics are available for well-known architectures [OR94, Las93, Knu95].

In addition to the problem of separating software subsystems from hardware modules, the hardware-software codesign design process must model the effects of interconnections. Since context switching may require additional resources, or have strict timing requirements, the impact of connection overhead is non-negligible. Defining a proper model for data transmissions is essential to create an efficient optimizer for hardware-software codesign systems as most partitioning algorithms require cost information for edges [Las93].

Since our system is implemented as a wrapper around an existing CAD environment (see Figure 3-2), the interconnect model was chosen to complement the data model of the *PIPE* CAD system. Since the *PIPE* environment uses a control-data-flow graph to model data connections, our model treats data control as a timing constraint between functional vertices. For practical problems, simplifying connections to timing is sufficient for acceptable performance.

2.1 Clustering

Identifying the basic blocks in a hardware-software codesign system may be done using several approaches. The following listing, based on [Knu95], enumerates some of the possible clustering methods.

Trivial clustering, assuming every operation to be a basic block in itself is impractical in most systems, but may be useful in smaller embedded HSCD devices.

Obviously, trivial clustering is equivalent to no clustering at all, and is considered a clustering strategy only in theory.

Clustering to a certain level, assuming there is a critical submodule size which is the practical limit for local optimization. The reasonable cluster size depends on the technology of implementation and the local optimization algorithms.

Even-sized clusters with maximal size limit, resulting in a cluster layout which may be optimal for size-constrained target architectures.

Even-sized clusters with maximum element limit, similar to the above, more useful for target architectures that are constrained by I/O capability or similar implied bottleneck.

All of the above clustering heuristics may be performed by using only the system data-flow graph, without any knowledge of the high-level description it was generated from. Such problems may then be solved by any of the graph coloring heuristics, and are not discussed in this dissertation. (Note that partitioning heuristics, described in more detail in Section 2.2, are generally useful for clustering since the problem formulation of clustering is identical to partitioning, as discussed in [AHK96].) Experiments performed with the

Kernighan-Lin algorithm and its extended version, the Kernighan-Lin-Fiduccia-Mattheyses algorithm (see next section) show that these popular partitioning heuristics may be used for clustering with satisfactory results [HB95a].

Some clustering heuristics are based on the algorithms of finding highly connected subgraphs of the system data-flow graph, a well-known NP-complete problem [Hoc97, Chapter 6, p. 234]. These heuristics generally find distances between sets of vertices and attempt to place vertices within very small distances in the same cluster. Several versions of this basic idea exist; a thorough description and comparison of such algorithms is given in [HB95a]. The primary disadvantage of these heuristics is that threshold numbers and distance expressions generally depend on the target technology. Also, since most of the connectivity-related algorithms must be extended to some of the auxiliary features (such as high-connectivity signals, i.e., global controls and shared resources), automated clustering using connectivity-related heuristics is not feasible. Since this high-level design process is intentionally separated from the implementation details of the underlying hardware, connectivity-based clustering heuristics did not provide satisfactory results during benchmarking without extensive customization. For this reason, they are considered to be inefficient for the target system, and are not recommended in a general-purpose MCHLS environment.

Even if connectivity-related clustering has produced inefficient implementations, clustering may still be performed in a MCHLS environment with satisfactory results. The main difference is that clustering is performed at an earlier stage of the top-down design process, when information is still available about system architecture. Starting from this higher-level description of the design enables more efficient clustering than lower-level (i.e., connectivity-based) algorithms. The dissertation recommendation for clustering assumes the high-level description is known, and clustering is performed before the complete elementary operation graph is generated.

If the clustering process is integrated with data-flow graph generation, it is possible to reuse the structure of the high-level description in the clusters. Practical systems may simply stop system refinement at a sufficiently high level, and treat functional blocks as clusters for partitioning. In the *HSPIPE* environment, such a solution is feasible since the design process is definitely top-down. An example of this approach to clustering is demonstrated on a practical example (GSM speech encoding) in Section 3.2.

Even if the initial high-level description is not available (i.e., the problem is initially formulated as a data-flow graph), the clustering process may try to match subsystems with known structures. Having a large knowledge base of previous designs and reusable modules, one may try to search the data-flow graph for known structures that are available in a module library or software repository. Identifying submodules in an early stage may decrease design time considerably as the available module database may be highly optimized. The identification process, on the other hand, increases the time requirement of clustering and requires an efficient associative-search algorithm. Recreating higher-level descriptions of the problem from the data or control-flow graph is a complex pattern-matching problem and is not part of the dissertation investigations.

For the dissertation benchmarks, clustering was implemented inside the top-down design process. It is assumed that there is no requirement in *HSPIPE* to implement a clustering algorithm capable of matching low-level structures to existing primitives. Incorporating such a system is definitely possible (since the design process is modular), but left as an exercise to users wishing to extend the system to architectures not covered in this dissertation.

2.2 Partitioning

Searching for an efficient way of separating hardware and software sections is an NP-complete task. The hardware-software *partitioning process* is an instance of the *optimal-cut problem* [Hoc97, Chapter 5]. The optimal-cut problem, and its specialized variants, minimizes a cost function for a graph which has its vertices assigned to different partitions. The cost function is based on the distribution of vertices and edges connecting vertices in different partitions. The weight structure of the cost function depends on the actual requirements of the target architecture.

The available heuristics are classified based on the results of researchers working on hardware-software codesign. Because of the unique requirements of the hardware-software codesign design environment, some of the frequently used partitioning, scheduling, and allocation heuristics are not directly applicable to hardware-software codesign. As an example, partitioning heuristics coming from supercomputing research often target uniform execution times across different computations [LH94], since the longest calculation time is a lower bound for the execution time of the whole computation [KK97]. Minimizing the longest calculation time puts additional constraints on the partitioning process, namely, partition sizes must be represented in the system cost function. Since the extension of cost functions makes the optimization process more difficult without improving the properties of the design, heuristics optimizing these extended cost functions are of limited usefulness for our target environment. In some cases, popular representations of parallel computations may be inaccurate, since cost functions are applied to improper measures of communication costs [HKed].

Similarly, in hardware-software codesign, partition costs are not directly related to system (or system flow-graph) geometry, and geometry-based partitioning techniques [AL98] [Las93, p. 15] are generally not applicable for hardware-software codesign purposes. As well-known VLSI circuit partitioning algorithms also use geometry-related information, some of these heuristics are also inefficient when used to partition hardware-software codesign systems [AHK96].

The partitioning problem has been thoroughly studied in the literature and is covered by both approximation algorithms and heuristics based on empirical results. Partitioning problems have to be solved in some of the most important stages of hardware and software design:

- *Register allocation*, both in software and hardware, is a process which partitions flow-graphs trying to optimize register assignment. The allocation problem, present both in compilation and high-level synthesis, minimizes the number of concurrently active elementary operations. For software systems, efficient register allocation method minimizes the impact of temporary storage access (*spill code*). In hardware, allocation attempts to minimize implementation costs by identifying possible resource sharing among elementary operations.

- Multiprocessor applications must be properly distributed among the processors so that load is balanced and distributed calculations are finished as soon as possible.

  The solutions are generally portable between different partitioning problem instances, with several of exceptions. Most of the exceptions are related to different cost metrics for different formulations of the same problem.

Since some of the available heuristics have been successfully applied to structures resembling typical *PIPE* designs, changes to the *PIPE* CAD system attempt to reuse some of this knowledge. As the hardware-software codesign research community has thoroughly analyzed most of the well-known heuristics, most results indicate that the combination of a single global pass and a sequence of local refinements results in feasible partitions [PD96, Las93].

Global (or construction) partitioning algorithms *create* an initial partition of the input graph. In hardware-software codesign, unlike partitioning in parallel computation applications, balancing the partitions is not necessary.

Local (or improvement) partitioning steps operate on an initial partitioning scheme, and try to introduce small changes that improve quality. Changes in the cost function are evaluated for small changes, and the one with the best effect on the cost function is taken. Improvement steps are generally repeated as long as the quality of the partition improves.

As the dissertation research treats the problem flow-graph as an immutable description, the dissertation investigations exclude *procedure cloning* [Vah99, MW96] and related partitioning techniques, which do modify the system graph. Cloning and replication techniques, in general, reduce the cost of context switches by replicating functionality in both execution contexts (as a tradeoff between implementation cost and communication cost). Should the *HSPIPE* system be extended with optimizations above the system graph level (such as the algorithmic extensions described in Section 7.2), implementation of such cloning/replication algorithm would become possible.

2.2.1 Construction algorithms

The most important global algorithms were developed in the field of parallel computation, where excessive inter-vertex communication is to be avoided because of prohibitive cost (increased data transfer times). This limitation (balancing) effectively disqualifies the majority of the heuristics developed in the supercomputing literature.

Providing a ***random initial partition*** is sometimes used as a very quick way of generating initial conditions [Knu95, p. 77] [DD96b]. The disadvantage of this straightforward approach, not surprisingly, is its random nature. Implementing such an "algorithm" is trivial, but it is not recommended for MCHLS. (As discussed later in this section, information about the design's higher-level description may be reused during the generation of the initial partition. Since such partitions may be matched to the structure of the system, they may provide better initial conditions than random initialization, with a small increase in execution time.)

A family of popular construction algorithms, *Greedy Graph Growing Partitioning* [KK95] has been demonstrated to provide good results with a high probability. The algorithm starts from a trivial cluster containing one vertex in one partition and grows this partition through local maxima. Vertices are moved between partitions one at a time, selecting the next one that provides the highest change in the cost function, terminating if there is no further improvement. Alternate versions of this greedy algorithm differ mainly in the cost function of the selection step.

As the quality of this greedy algorithm depends on the choice of the initial vertex, repeated attempts are recommended with randomized perturbations of the initial partition [AHK96]. Even if the Greedy Graph Growing heuristic has shown good results on practical partitioning problems, there are better choices for partitioning heuristics for the dissertation hardware-software codesign approach. The two most important disadvantages are the inherent randomness of the solution (necessitating further iterative rounds to improve the chances of finding a suitable partition), and the fact that extending the algorithm to non-binary partitioning (i.e., systems with more than two execution contexts) is not trivial. Since the dissertation research targets portability to more than two execution contexts, Greedy Graph Growing is not recommended for generating initial partitions.

Significant literature discusses the details of the numerous variants of ***multilevel partitioning algorithms*** [AHK96, KK97, KK95]. Multilevel heuristics, popular especially in VLSI circuit optimization, attempt to find an efficient partition by performing the following steps:

1. Subject the graph to *coarsening*, reducing vertex count based on clustering heuristics.

2. Perform graph coarsening recursively until vertex count is considered satisfactory.

3. Partition the coarse graph using an efficient, slow algorithm.

   Since the recursive coarsening is assumed to supply a greatly reduced vertex count, even computationally expensive heuristics are applicable to this initial partition. Because of the efficiency of the slower algorithms, initial conditions do not have a major influence on the result of this partitioning step.

4. Perform *uncoarsening* on the course graph. The uncoarsening step breaks the clusters created in the corresponding round of coarsening. After each uncoarsening step, a local improvement algorithm is applied to possibly improve the quality of the coarse cluster (by using neighborhood information gained during uncoarsening). The improvement algorithm may be applied iteratively.

5. Perform the uncoarsening step until the granularity of the original graph is reached.

Practical systems have been demonstrated for partitioning graphs from finite-element computational problems (featuring sparse matrices, or equivalently, mainly local connections) at the order of several tens of thousands of vertices and hundreds of thousands of edges under minutes on workstation-class computers [AHK96]. In practical systems of similar size, multilevel partitioning has been implemented as a very quick, sufficiently efficient initial partitioner. Note that the above results were obtained in a system with immediate feedback, i.e., no evaluation/estimation step was required between steps of the partitioning process. (In hardware-software codesign the evaluation step may dominate the runtime of the feedback loop.)

For the purposes of this dissertation, multilevel partitioning algorithms are not recommended, unless the problem size is very large. The primary reason to recommend against using multilevel partitioning is the relatively large execution time of the scheduling and allocation estimator steps. Since the execution time of the estimation iterations is dominated by the time of scheduling and allocation, slower partitioning heuristics (such as the Kernighan-Lin algorithm or one of its extensions) still provide reasonable runtime with much lower implementation complexity. At problem sizes exceeding thousands of elementary operations, a multilevel partitioning process is definitely worth investigating. Fortunately, because of the modular structure of the MCHLS framework, replacing the partitioning process with faster heuristics is possible without disrupting other modules.

For the purposes of this dissertation, especially for smaller graphs, initial partitions provided by a straightforward, greedy algorithm were sufficient. Starting from one of the extreme system configurations (i.e., purely software or hardware), the system is iteratively refined until it meets performance metrics. (The *VULCAN II* design environment successfully used a similar greedy algorithm at an elementary operation level. VULCAN II starts from an initial, all-hardware configuration, and iteratively moves operation branches to software to reduce cost [DMG92]. The complementary greedy heuristic, as shown in [EH92], moves selected vertices from an initially software solution to hardware to meet timing constraints. This software-based solution is also implemented at an instruction level.)

The steps to get a good initial partition depend on the primary constraint of the system, i.e., the dominant member of the cost function. In time-constrained systems, a solution with more hardware is generally better; similarly, in a system with space or cost constraints, an entirely software solution may be a better starting point. Because of the difference between these conditions, the initial partition is generated using slightly different methods:

- In systems under dominant cost constraints,
  1. Generate an initial, extreme system configuration: assign all operations to software.
  2. Terminate if system meets timing requirements, or comes sufficiently close. (This is an application-dependent measure of system cost, and varies depending on the target environment.)
  3. Transform the critical path or critical paths to hardware. Find the new critical path.
  4. Repeat from 2.
- In systems under dominant time constraints,
  1. Generate an initial, extreme system configuration: assign all operations to hardware.
  2. Terminate if system meets cost requirements.
  3. Transform operations outside the critical path to software (in time cycles with the highest hardware load).

4. Repeat from 2.

In practical terms, the applied heuristic is an expanded version of the greedy graph growing heuristic, where the choice of operation to be moved is better suited to the MCHLS design process. Also, the system may be expanded to handle more than two execution contexts, an advantage not present in the original form of greedy graph growing.

2.2.2 Improvement algorithms

Local partitioning algorithms, working on refinements of already existing global partitions, try to select small local changes that improve the performance of the partition. A number of frequently efficient local algorithms are derivatives of the *Kernighan-Lin algorithm* [KL70]. Since the efficiency of the Kernighan-Lin algorithm has been demonstrated in different configurations [Vah97a, BFS98, KL97, VNG97, HB95a], two versions of the algorithm are chosen as a basic module of *HSPIPE*. The analysis of relevant other algorithms summarizes potential alternate solutions at the end of this section.

In the time and space requirements, unless noted otherwise, n denotes the number of vertices and m the number of direct data connections in the graph.

The Kernighan-Lin algorithm starts with a partition and swaps pairs of vertices in an iterative way. The algorithm selects the vertex pairs based on a local cost function and tries to optimize the overall change in the global cost function. With $\mathcal{O}(n^2)$ time complexity (and a convenient, small constant factor), the Kernighan-Lin algorithm is reasonable to use in small to medium graphs (up to several thousands of vertices), but becomes infeasible for larger systems. An important extension to the Kernighan-Lin algorithm has been created by Fiduccia and Mattheyses in 1982 [FM82]. This modified version, based on the idea of [KL70], executes in $\mathcal{O}(n)$ or $\mathcal{O}(m \log n)$ time (depending on system type, see below) and considers moving individual vertices without attempting the swapping of vertex pairs.

The original version of the *Kernighan-Lin algorithm* [KL70] investigates vertex pairs of the system CDFG and attempts swapping *vertex pairs* between execution contexts to decrease total system cost. System cost function is defined as a non-decreasing function of the *cut size* of the partition, i.e., the number of edges connecting vertices in different execution contexts. The cost change caused by moving a vertex across an execution context boundary is described by a *difference function* in each partition. The Kernighan-Lin algorithm selects vertices as candidates for swapping based on their difference function, attempting to move vertices with maximal differences across execution context boundaries. Vertices that had been moved in a given improvement pass may not be moved any more in the same pass (they are said to be *locked*) to keep the Kernighan-Lin algorithm from oscillating in a pair of astable local minima. Each pass of the Kernighan-Lin algorithm continues as long as there are available unlocked vertices. After the pass is over, the total cost change is evaluated, and the next pass is attempted only if the previous one improved system cost. The Kernighan-Lin algorithm tends to present a non-increasing system cost improvement as the number of passes increases, and sometimes it's possible to terminate the algorithm well before it would actually stop with a relatively small impact on performance [HB95a].

For a better estimation of system cost in hardware-software codesign, the Kernighan-Lin algorithm must be extended with weights and performance attributes to properly model the effect of hardware-software communications. In the dissertation research, an efficient, but computationally cheap modeling method is used to transform execution context switches to variable-length delays. Since execution context switches should not affect data values, just representation, delay is a suitable model of the context switch. The actual length of the delay is a function of bit width, source context and destination context. Such a cost function may be constructed based on the timings of the hardware-software connections, communication protocols, and the properties of processor interfaces.

An extension of the Kernighan-Lin algorithm, the *Kernighan-Lin-Fiduccia-Mattheyses algorithm* [FM82] moves individual vertices across execution boundaries instead of attempting vertex swaps. Graph vertices are scanned sequentially, evaluating system cost after attempting to relocate each of the vertices to a different partition. The vertex and the partition it is moved into will be the move which reduces system cost the most. The iteration terminates if no move decreases the system cost any more.

The Kernighan-Lin-Fiduccia-Mattheyses algorithm in hardware-software codesign proceeds along a discrete gradient of the system cost function

$$D = D_t + D_r + \sum_{(e_i,e_j)\in W} w_{i,j}(x_i, x_j, n_{i,j})$$

where D_t is an indicator function for timing violations. Similarly, D_r is an indicator function for resource constraint violations. The definition of these functions depends on the target environment and primary design constraints (see Section 3.3).

Note that in balanced applications of the Kernighan-Lin-Fiduccia-Mattheyses algorithm (i.e., where balanced partitions are desirable), an additional indicator term is present, penalizing large differences between execution context vertex counts. For reasons discussed earlier, MCHLS makes no use of such requirement, and balance terms are not used. Note that in the *HSPIPE* implementation, context switch information is merged into the graph, therefore the last term of the cost function is implicitly evaluated in D_t and D_r, and no longer needs to be calculated.

Potential cost improvement is evaluated for each vertex movement by evaluating the cost function before and after the movement. The move which provides the decrease in system cost is taken after each pass. The time complexity of the Kernighan-Lin-Fiduccia-Mattheyses algorithm in its original form is $O(n)$, but the extensions for non-uniform weights (a feature the original form lacks) increases that to $O(m \log n)$. In the example implementation of *HSPIPE*, the higher runtime is used, with reasonable runtimes for small and middle-sized problems.

Because of the greedy nature of the Kernighan-Lin-Fiduccia-Mattheyses algorithm, the algorithm may converge to inefficient local extreme values, as it explores only a very small subset of solution space. Several heuristics have attempted to improve the behavior of the Kernighan-Lin-Fiduccia-Mattheyses algorithm. The primary concern is the lack of lookahead, since the Kernighan-Lin-Fiduccia-Mattheyses algorithm in its basic form is capable of converging to local minima of the cost function. Extending the Kernighan-Lin-Fiduccia-Mattheyses algorithm with first-order look-ahead extensions, i.e., [Bal84], increases performance at the cost of runtime. Higher-order look-ahead did not show the expected increase in solution quality, while it increased runtime considerably, and is generally not used.

Various iterative improvement algorithms have been developed to overcome the shortcomings of the Kernighan-Lin algorithm and inspired improvement methods. Significant research has been conducted in VLSI circuit partitioning, where netlist optimization presents similar problems to execution context distribution in hardware-software codesign.

Based on an idea similar to force-directed scheduling (see below in Section 2.3), *probabilistic partitioning approaches* may be used to increase the performance of the Kernighan-Lin-Fiduccia-Mattheyses algorithm [DD96a]. Under the probabilistic heuristic described in [DD96a] (an $O(m \log n)$ time heuristic), a movement probability is assigned to each vertex. This probability denotes the likelihood of the vertex being moved in the current pass of partitioning; it is practically a measure of potential cost improvement for the given vertex. The probability function is initially uniform for all vertices, and later updated based on the real cost improvement obtained by moving the vertex.

Since probability functions depend on the values of the real cost functions, the probability functions are implicit functions of the system configuration, and therefore each pass of the probabilistic partitioning process is iterative. In production environments, where resource utilization is the most important design criterion, implementing such an iterative improvement algorithm may be desirable. During the testing of the *HSPIPE* implementation, the Kernighan-Lin-Fiduccia-Mattheyses algorithm was used because of the predictable runtime.

Utilizing the ideas similar to that of multilevel partitioning, *clustering-based iterative improvement* algorithms [DD96b] attempt to improve partitioning heuristics by running cluster-partition-uncluster iterative loops. The partitioning heuristics are performed on problems of significantly smaller size, potentially reducing execution time. Also, by calculating aggregated system cost functions, the movement of vertex groups may be evaluated instead of individual nodes, reducing search space considerably.

While practical results show that clustering-based improvement algorithms may provide better solutions than the Kernighan-Lin-Fiduccia-Mattheyses algorithm, such an advantage is lost if the Kernighan-Lin-Fiduccia-Mattheyses algorithm is aware of an efficient clustering of the initial graph. (Such is the case in MCHLS, where the snapshot of a higher level of the top-down system description presents good initial clusters.) In such systems, the computational overhead of a clustering-based improvement heuristic might be too high.

Such is the case of *HSPIPE*, where for moderate problem sizes (including the benchmark applications), computational overhead consumed most of the performance gains obtained by implementing a more efficient solution. (In this respect, the straightforward Kernighan-Lin-Fiduccia-Mattheyses algorithm has a definite advantage over more efficient algorithms.)

2.3 Scheduling

One of the computationally expensive problems of compiler optimization and high-level synthesis is scheduling, the arrangement of elementary operations according to cost calculations. Scheduling attempts to generate a *schedule* (timetable) for a set of operations such that the system performs prescribed calculations without violating time and resource usage constraints. Time constraints in the *HSPIPE* design environment may mean both system latency and restart time. (The two optimization goals are conveniently represented with the same kind of constraint.)

System latency (L) is the time difference between sampling the first system input and producing the final value of the last system output.

Restart time (R) is the time difference between subsequent data on the system inputs.

In non-pipelined systems, restart time is equal to system latency. In pipelined systems, such as generated by *HSPIPE*, restart time is lower than system latency.

This section presents an overview of the scheduling problem in general, redirecting the reader to references, where necessary. Since the *HSPIPE* system is modular, scheduling (and allocation) may be replaced with a more efficient algorithm, if so desired. Because of this modularity, the analysis in this section (and in Chapter 4, "Implementation") presents the scheduling heuristics considered in the example implementation of the *HSPIPE* environment. Since the choice of scheduling algorithm may depend on the target environment, users of *HSPIPE* are free to extend *HSPIPE* scheduling during iterative rounds with algorithms more efficient under design-specific circumstances.

In hardware systems, an execution schedule is a set of timing values (v_i). The timing values contain controller information for starting cycles of elementary operations. The system controller is synthesized after the register-transfer level description of the system is available. The controller generates a start pulse for the processor containing the given elementary operation in its start time cycle v_i. Since hardware components are theoretically capable of starting processing at any time, such control information is necessary to guarantee stability of signals.

In software, scheduling prescribes a permutation of instructions of the original code stream such that the system outputs are not changed by the permutation (*instruction scheduling*). Since instruction fetch accesses memory in a monotonous way, increasing fetch addresses if no branches occur (even in architectures with out-of-order execution), the order of instructions in the compiled binary influences execution time. (Since generated instructions are constructed from software sections of the elementary operation graph, the words "operation" and "instruction" are used interchangeably in this section.)

The goal of software instruction scheduling is increasing (potential) instruction-level parallelism. Because of the limited number of processing units in microprocessors (*execution units*), any attempt to use a busy unit is *stalled*, blocked from execution until the requested unit becomes available. Such a disruption in the input code execution is referred to as a *pipeline stall*. Instructions must be ordered so that the effect of pipeline stalls is minimal.

It must be noted that the dissertation assumes pipeline stalls as the only concern in evaluating software performance. Other performance issues, such as improper *branch prediction* (and the related pipeline flushes) are not covered. Unlike the mechanism of pipeline stalls, branch prediction internals may change considerably between even close revisions of the same microprocessors. Since a reasonable attempt to optimize code for efficient branch prediction would be difficult without an exact description of the branch prediction mechanism, it is infeasible to do in a way that is separated from hardware internals, and is not attempted in this dissertation. As most popular branch prediction heuristics depend on the distribution of branches taken and not taken, most of prediction problems may be addressed at the source code level, without regard to scheduling. The effect of these optimizations passes through instruction scheduling without changes, since they affect the execution flow of the code in a data-driven way, which is easier to control by the programmer.

Some scheduling methods are theoretically capable of moving instructions to arbitrary positions in the elementary operation graph. Such scheduling algorithms are referred to as *global scheduling*. Heuristics to be used as global schedulers should have a very low asymptotic time complexity to be effective, since global scheduling in an MCHLS deals with the lowest-level description of the system (which, obviously, has the highest vertex count).

Other scheduling heuristics are incapable of moving operations beyond the borders of the *basic block* they are in.

Basic blocks are sections of the instruction stream which are fetched from memory in an entirely sequential way.

Note that this definition, modifying the one in [ASU88], is valid even for out-of-order execution RISC microprocessors.

Scheduling heuristics that limit the movement of instructions to their basic blocks are called local schedulers. Because of the smaller number of operations to be considered with local schedulers, slower heuristics may be feasible schedulers if the performance gain justifies the slightly higher runtime.

2.3.1 List scheduling

More a collection of related heuristics than a single algorithm, *list scheduling* is a set of relatively fast methods of instruction scheduling. List scheduling algorithms generally maintain a list of instructions ready for execution and select the one to schedule next based on list position. List scheduling may be performed both locally and globally, although local problems are generally smaller, and would provide better solutions without significant increases in execution time by using better heuristics [NP95a]. For this reason, the dissertation recommendations concentrate on global list scheduling.

List scheduling is usually executed in the following way:

1. Construct operation ASAP and ALAP times (movement limits) as shown in Section 3.2. (Find the earliest and latest start cycles that do not violate constraints.)

2. Set the current time cycle to 0. (The algorithm fills up time cycles with instructions starting from this cycle.)
3. Gather all operations that may be started in the current time cycle (i.e., all their immediate predecessors have finished executing). These are the *available* operations.
4. Create a priority list of the available operations.

   The priority function is usually a function of operation mobility (i.e., number of cycles left before the operations ALAP cycle).
5. Assign as many operations from the top of the priority list to the current time cycle as possible (i.e., prescribe them to be started in this cycle).
6. Delay any non-assigned available operations to the next time cycle.
7. Delay operations affected by assignments to the current cycle, if necessary. (This applies to successors of currently assigned operations.)
8. Delay operations affected by delaying non-assigned available operations. (This affects the successors of such operations.)
9. Advance current time cycle to next one.
10. Repeat from (3) if there are still unassigned operations.

In the target system of MCHLS, the original idea of list scheduling [Gra66] must be extended with instruction dependencies and multiple execution unit types. Different extensions of list scheduling exist for slightly different problem instances and different conditions. The dissertation investigations had support for non-uniform execution times, dependencies between operations, as a design goal. Exhaustive description and analysis of approximation algorithms related to list scheduling is presented in [Hoc97, Chapter 1, p. 1].

It is worth remembering that the original list scheduling heuristic, as described in [Gra66], is an *1-approximation algorithm*, i.e., it generates a solution having a latency of at most twice that of the optimal. While this performance may not be impressive, such a very fast algorithm may produce an estimation on the optimal solution. Such an estimation, combined with information extracted from other graph-related details [OKD97], may be useful for limiting the number of iterative rounds of the estimation process.

Since the example MCHLS design process implementation uses list scheduling in the iterative step of partitioning and evaluation, runtime of scheduling heuristics was a very important algorithm parameter. As the iterative step of the MCHLS design cycle wishes to approximate solution efficiency without providing an actual solution, the performance of the scheduling heuristic is of secondary importance. By selecting a list scheduling heuristic which is an approximation algorithm at the same time, the results of the scheduling step may be used to give an accurate bound on the properties of the optimal solution.

Since iteration time is more important than algorithm performance in the iterative stage, very efficient but complex approximation algorithms have been excluded from the dissertation investigations. Similarly, algorithms where resource usage would be prohibitive are not considered for an MCHLS development process. Such algorithms include, but are not limited to, algorithms where solutions are formulated as solutions of linear programming problems [Hoc97, "Unrelated parallel machines", 1.7.3, p. 41], or heuristics where efficiency requires execution times to be uniform [Hoc97, "The general job shop: Unit-time operations"].

2.3.2 Balanced scheduling

An extension of well-known list scheduling methods, ***balanced scheduling*** [Ker93] provides the ability to efficiently distribute ***blocking instructions*** in the instruction stream. Blocking instructions are instructions that may temporarily suspend instruction processing by attempting to utilize a busy processing unit of the microprocessor. Such blocking instructions include instructions causing pipeline stalls and memory loads. (RISC architectures, in general, have to slow down considerably for memory loads.)

It must be noted that balanced scheduling is useful mainly in software environments. Even if cache effects may be observable in hardware systems, such as communication interfaces with their own caching schemes (such as the Corollary C-Bus-II) or subsystems with non-deterministic latencies (such as serial interfaces with internal compression), balanced scheduling may be still ineffective for hardware. The reason hardware scheduling may be performed without such extensions is that modeling non-determinism at this hardware level would require extremely complex extensions to list scheduling. Exhaustive modeling of such timing would quickly overtake the time complexity of the list scheduler. (As a matter of fact, the model would be practically impossible to port between different environments.)

A very important characteristic of the balanced scheduling algorithm is that it provides the capability to model variable execution time during scheduling. (Most scheduling heuristics treat execution time as a constant, which must be set before scheduling is started.) The idea of variable execution times may be safely extracted in optimizing software execution.

Combining balanced scheduling with other well-known software performance-enhancing steps, such as *loop unrolling* and *trace scheduling* may increase the performance, if the target system is known in detail [LE95]. Such extensions have not been implemented with the demonstration implementation of *HSPIPE*.

2.3.3 Force-directed scheduling

A popular, $O(n^3)$ scheduling algorithm, *force-directed scheduling* [PK89] is a possible heuristic for scheduling in a MCHLS environment. Given the cubic time complexity, care must be taken to apply the algorithm to smaller subgraphs to limit execution time. In practical design systems implementing force-directed scheduling, applying the algorithm within basic blocks, without interaction between basic blocks, produces good results in such local optimization. The total runtime is considerably reduced by applying force-directed scheduling locally. (As the runtime function is nonlinear, a third-order polynomial, total runtime decreases considerably if individual problem sizes are smaller). The force-directed scheduling algorithm usually produces very good results, with a number of exceptions noted in [AJV].

The basic steps of force-directed scheduling are the following:

- Construct operation ASAP and ALAP times (movement limits).
- Label every elementary operation as unassigned (i.e., subject to scheduling).
- Calculate expected system load for each time cycle.

  To calculate the expected load, a probabilistic approach is taken to estimate the number of execution units needed in each time cycle. For each elementary operation, a uniform probability of starting is assumed to each time cycle in its time frame. By calculating the sum of probabilities for each time cycle and for each execution unit type, an expected resource utilization is obtained. (Obviously, assigned operations have their resource usage described with the probability of 1 in the time cycle they are assigned to.)

Note that this generation step is slightly different for variable-length execution, but the necessary details are not replicated here. The example in Chapter 4 and the original source [PK89] discuss variable-length execution.

- Select an unassigned elementary operation.
- For each time cycle the selected elementary operation may be started, perform the following:
    1. Fix the operation to the currently investigated time cycle.
    2. Update time frames of operations that are affected by fixing the starting time of the current one.
    3. Calculate expected system load functions based on the modified time frames.
    4. Calculate the cost change function. This function is defined as a function of type $F = -\sum C \cdot \Delta C$, i.e., similar to a spring force function, where C is resource utilization. (The similarity with spring force equation is the reason for the name "force-directed scheduling.")
- Start the current operation in the time cycle where the cost change function was maximal. (This process attempts to make system resource requirements uniform as a function of time.)
- Repeat from system load calculation if there are still unassigned operations.

After implementing the modules of experimental MCHLS design environment, force-directed scheduling has been successfully applied to benchmark problems. The increase in execution time, especially when compared with faster list scheduling algorithms, is not justified unless the accuracy of the approximation is extremely important or the calculations may be parallelized [PB96].

2.4 Allocation

Allocation maps elementary operations to execution units. It is an instance of the *graph coloring* problem. An allocator colors the data-flow graph in such a way that no operations of the same color are executing in the same time cycle. In this case, each color corresponds to a unique execution unit. Operations with the given color are executed in the same processor. ("Execution unit" and "processor" are used interchangeably in this section.)

2.4.1 Topological cover

A collection of application-specific heuristics, topological cover attempts to decompose the data-flow graph into identical execution units without solving the graph coloring problem. Because the operations mapped to each unit are usually different, some of the execution units may be underutilized. In the example in Figure 2-1, the execution unit contains a shift register, a multiplier and an adder. They are fully utilized only in the second time cycle.

The resulting system is very similar to systolic architectures [Kun82]. Unlike systolic systems, interconnects with topological covers are usually not regular. As a disadvantage, topological cover is tied to the target technology. There is no known heuristic for topological cover, except manual intervention. For this reason, it is not a feasible solution to most automated designs, unless a large database of designs is available. With a sufficiently large database and efficient pattern matching, topological cover may be partly automated (with a very high runtime).

2.4.2 Concurrency

A traditional approach to allocation, concurrency-based algorithms attempt to solve the graph coloring problem [Ata98, p. 28–10], [Hoc97]. Since this problem is NP-complete, heuristic approximations are available. The graph coloring problem is equivalent to the selection of maximum compatibility classes in sequential digital designs. The heuristics applicable to compatibility class selection are applicable to allocation.

In practical systems, concurrency-based allocation is extremely popular, if the scheduler

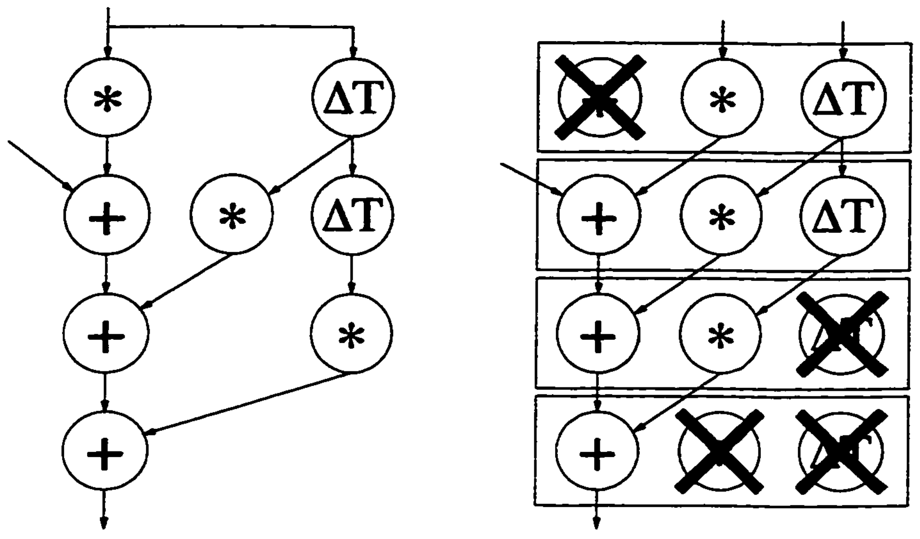

Figure 2-1: Topological cover

is capable of generating a schedule under resource constraints. Integrating the allocator with such a scheduler, the number of colors is influenced by both allocation and scheduling [NP98], providing instantaneous feedback.

2.4.3 Software allocation

Slightly different from hardware-based allocation, register allocation may not increase the number of execution units. In practical systems, spill code is used to implement the necessary number of processors. Because of the performance penalty of spill code, register overload is heavily penalized, especially in RISC systems.

Several variants of list scheduling offer extensions to handle strict resource limits. The solution to such problems is to reorganize the data-flow to a given concurrency by delaying some of the operations. A typical implementation of this greedy algorithm proceeds with a list scheduler under resource constraints. The allocator then simply scans the time cycles in increasing order, and assigns execution units sequentially. Since the schedule complies with resource constraints, there may be no conflict in allocation. Under these conditions, the allocator has to select the best execution unit to implement each operation.

As a disadvantage of resource-constrained schedulers, latency of the design is out of designer control. In practical systems, scheduling is performed under a resource constraint, and the constraint is relaxed until the latency becomes satisfactory. In software systems, spill code may be inserted after this step, if the number of required execution units exceeds that of the available ones.

2.5 Relevant research in system-level synthesis

The literature survey analyzed a number of publications from the field of system-level synthesis. Since the results of system-level synthesis-related research and development topics (clustering, partitioning, scheduling, allocation, integration) come from several different fields, some of the most interesting results may not be directly applied to generic multiple-context environment systems. This section, contains a brief summary of research directions from the field of hardware-software codesign.

Researchers at the *University of California at Riverside*, notably *Dr. Frank Vahid*, attempt to solve the partitioning problem at a sufficiently high level of abstraction [VG92]. To achieve the necessary abstraction level, problem descriptions are not treated as collections of structures, and system descriptions are functional, without explicitly specifying the underlying structure. (VHDL as a modeling language permits designs to be specified at both structural and behavioral levels.) High-level partitioning is performed at the *functionality level* [GVNG98], and the corresponding partitioning problem is described as *functional level* or "Specification Partitioning" [VG92]. Even if problems may be easily solved using functional partitioning, the necessary detail level may not be obtained since partitioning does not explore a large number of possible solutions.

The limited flexibility in moving functionality between different blocks, functionally partitioned systems tend to be more difficult to optimize. Even if the number of available transistors is increasing exponentially with time, the inefficiency in some functionally partitioned designs makes them too expensive for mass-production of microelectronics systems (as of today). Note that the number of available transistors has been increasing much faster than designer productivity, and there is no indication of a change in this tendency [Wir98, VG99].

As a very important advantage of the functional partitioning approach, it makes it possible to partition software components among several processors, an advantage not present in most other practically used hardware-software codesign methods. In addition to the requirements of multiprocessor software implementations, functional partitioning enables design reuse. Extending already existing systems is also easier, since the existing modules need to be described in high-level languages (i.e., behavioral VHDL, Java or C), without regard for the underlying structure [Dew97]. Specification partitioning has the ability to change system descriptions, which would be impossible under traditional CDFG-based design processes. Since specification changes introduce an additional layer of complexity to the design steps, this dissertation research does not contain extensions to handle specification partitioning and higher-level CDFG manipulation. As a future development, an additional layer capable of functional partitioning may be inserted to the system, between the high-level description and the CDFG generation phase (see Section 7.2, p. 131).

An additional advantage of a higher-level description of system functionality is the ability to perform incremental performance approximations in the design flow. Since most partitioning steps relocate only a limited set of operations between different execution contexts, usually a small fraction of system usage maps has to be recomputed between iterations. *Dr. Frank Vahid* and *Dr. Daniel D. Gajski* have extended structural partitioning techniques with incremental evaluation, reporting impressive performance improvements in systems where changes between iterations are most likely to be gradual [VG95b]. Incremental evaluation techniques, which are very effective in most practical hardware-software codesign systems, may require more knowledge of the input system CDFGs than a generic hardware-software environment.

In the well-known *Ptolemy* design environment, developed at the *University of California at Berkeley*, systems are designed at the functional level, not unlike the research group of Dr. Frank Vahid. The target environment of Ptolemy is mainly DSP-related, and therefore offers unique possibilities of expanding functionality in hardware. Utilizing DSP features also presents unique challenges unknown in general-purpose microprocessors [ML97, BML98, BML97] and requires extensive experience in practical DSP applications. Since DSP-capable microprocessors tend to change significantly between device generations, system synthesis details must be constantly updated in the design environment, or designers are unable to exploit the full capabilities of the CAD tools. The designers of Ptolemy have chosen an approach where users are capable of integrating incremental changes to the Ptolemy design environment. Since users may interface their custom components to existing Ptolemy modules, the design environment may be extended locally, until features are merged back to the original code base.

Important results have been presented by the *Eidgenössische Technische Hochschule (ETH) Zürich* in Zürich, Kanton Zürich, Switzerland. ETH researchers, among other activities, have extended the *occam* programming language with constructs that may easily be synthesized into silicon. Efficiency of the synthesized structure is not the most important design goal, observing the increasing gap between available and utilized transistors in state-of-the-art systems [Wir98, VG99].

Since ETH researchers have significant experience in compiler design and optimizations, ETH results concentrate on the actual synthesis process itself, and some of the results may not be easily extended to multiple-context environments. The hardware-oriented research of the *occam* programming language at ETH has been successfully utilized in practical projects several times. As the primary field of experience of ETH researchers is in the field of software systems, the optimization capabilities of currently existing ETH system-level synthesis tools are limited to results of compiler theory. As the current version of the ETH synthesis environment is primarily ASIC-oriented, an additional level of complexity is present in the layout generation phase. Extensive ETH research covers synthesis of smaller embedded systems [Tei97]. In [ZEK$^+$98], a communication model similar to that of this dissertation is presented, with additional coverage of non-deterministic specifications. (The added complexity of such non-deterministic descriptions is outside the dissertation investigations.)

Building on the previous experience with compiler design and reconfigurable hardware systems, *Dr. Ian Page* from the *Programming Research Group of the Oxford University Computing Laboratory*, Oxford, Great Britain, concentrates on programmable hardware modules only. Typical projects of the Laboratory are focused on generating custom processors from high-level software descriptions. The Laboratory uses descriptions written in the *occam* and *Handel* programming languages and generates FPGA descriptions (connection lists and bitfiles). Practical systems have been demonstrated in moderate-bandwidth signal processing applications.

The Oxford University Computing Laboratory does not try to handle the hardware-software codesign design process as a traditional development cycle. Since their synthesis process generates a custom "microprocessor" in reprogrammable hardware units, an already existing microprocessor framework is required to avoid extremely long synthesis cycles. This approach may increase the ratio of wasted silicon to a very high level, especially in smaller designs. Even if the number of available gates in reprogrammable systems increases in an exponential fashion over time [Wir98, VG99], straightforward code generation without optimization steps may be infeasible.

Because the Oxford University Computing Laboratory researchers do not wish to make a clean distinction between hardware and software as target systems, the laboratory has a slightly different approach to system descriptions than generally accepted. Instead of relying on VHDL descriptions for hardware and C code for software subsystems (or combination thereof), the laboratory strictly enforces the policy of using a common description language to cover problems [Pag95]. The programming language *Handel*, a derivative of *occam*, is a deliberately Spartan subset of functionality (the only exception being type conversion, which has extensive support to reduce hardware waste). By providing less comfort for designers, the Handel environment reduces the burden on optimization, and decreases compiler complexity considerably. Creating a higher level of abstraction and additional optimization steps in the Handel compiler is a future development plan of the laboratory.

Dr. Ahmed Jerraya from the TIMA laboratory (*Techniques de l'Informatique et de la Microelectronique pour l'Architecture d'ordinateurs, Techniques of Computer Sciences and Microelectronics for Computer Architecture*) from the Grenoble National Technical Institute (*Institut National Polytechnique de Grenoble*, INPG) relies on an extensive set of industrial applications rather than academic background as a research tool [AJ97]. Previous TIMA/INPG projects have been successfully completed in telecommunications, control systems and digital signal processing. As a consequence of previous research, TIMA research treats the reusability of system-level synthesis designs as one of the most important parameters.

Because previous TIMA research is widely available and reusable, most practical TIMA projects attempt to partition the system based on complex submodules. The design process does not involve optimization across submodule boundaries, and the partitioning problem does not deal with higher-granularity clusters. Because of this reason, the basic TIMA development process does not scale easily to other, previously unexplored projects. These problems are expected to be solved as TIMA moves on to closer academic collaboration with other universities.

Since the TIMA designs generally explore a limited subset of solutions, design times may be much shorter than other hardware-software codesign design processes. By investigating a strictly limited choice of system partitions and implementing the one with the smallest cost function value, a tradeoff is being made between system performance limits and development time. This tradeoff, which appears relatively early in the design cycle, may have a serious impact on system capabilities, since it reduces the designer's degrees of freedom considerably. The selection of potential system partitions is therefore crucial to such a design process. In the case of the TIMA laboratory, there is an extraordinary amount of currently existing development knowledge, and the repeated successful reuse of such knowledge has proven to be successful in industrial TIMA projects. Since the hardware-software codesign knowledge base of BME (Technical University of Budapest, Hungary) is limited compared to the TIMA laboratory, implementing a design process primarily on reuse of existing standard designs is not feasible today.

Two research groups at the *University of Washington* have investigated the partitioning problem in soft-programmable FPGA structures. Expecting the advances in FPGA technologies, extensive benchmarking and algorithm development has been targeted in the early nineties, while FPGA implementations became capable of building the required complex structures only recently. The inherent problems of unpredictable propagation times inside FPGA packages change the focus of the partitioning process somewhat. In addition to multiple restrictions on system topologies and limitations on the number of signal propagation levels, highly nonlinear (exponential) timing penalties have to be introduced to graph cuts [HB95b]. The high penalty values are caused by the exponential increase in delays (load capacitances) when signals are leaving and entering FPGA packages. In addition to the differences between internal and external propagation delays, delays inside packages depend on compilation circumstances and may be difficult to predict in most practical systems. Even if recent advances in FPGA routing technologies decrease the standard deviation of internal propagation delay distributions, FPGA timings should not be modeled as fixed values.

The University of Washington research results include detailed analysis of partitioning algorithms adapted to supercomputing [HB95a] and hardware systems with multiple FPGAs [HB95b].

Several universities and industrial researchers are active in the field of graph partitioning itself, since this problem is one of the most frequently encountered problems in several areas of computer science and electrical engineering.

Chapter 3

Multiple-Context High-Level Synthesis

This chapter introduces the process model and the heuristics that were found useful in SLS-oriented extensions of the HLS problem. Assuming familiarity with the steps of HLS, compilation, clustering, and partitioning, the chapter presents the necessary changes between a single-environment and a multiple-context HLS process. Differences are analyzed to approximate changes in computational complexity and to verify that existing heuristics are able to process the different structures.

The chapter provides a system of mathematical notations. It also presents a formal description of the control-data-flow graphs in a way which is compatible with existing scheduling (and allocation) notations and yet accommodates descriptions of multiple-context environment systems.

The MCHLS extensions over single-context designs are not limited to environments with two execution contexts (binary partitions), and the results may be extended to handle systems with more than two execution contexts.

Since the *Multiple-Context High-Level Synthesis* (MCHLS) design process is assumed to be a transparent extension of *High-Level Synthesis* (HLS), the inputs and outputs of the MCHLS design are the same as the traditional HLS process. The only observable difference is in the output, since MCHLS terminates with a register-transfer level description of hardware subsystems and a data-flow graph for software modules; the latter is obviously not present in HLS results (Figure 3-1).

The high-level description input of both HLS and MCHLS is assumed to be a flow-graph, which in turn is usually generated from a description at an even higher level of abstraction. Most of the time HLS problems are given as an algorithm, or a sample implementation in a sufficiently high-level programming language. The usual choice for describing data-flow graphs is to generate them from standard programming languages, typically C, C++, or, increasingly, Java [YMS+98].

The popularity of the Java language comes from the wide range of systems capable of executing Java, ranging from miniature embedded systems to large computing clusters. The Java programming language is conveniently standardized [LY96] and portable across hardware platforms to provide a good algorithm description language. Another advantage of object-oriented language descriptions is information hiding. Object-oriented languages offer enforcement of hiding implementation-dependent and hardware-specific details behind object (class) interfaces. Practical hardware-software codesign implementations exist with C++ and Java descriptions of communication protocols, where the actual protocol representations remain invisible to the designer beyond class methods (expanded only at compile time from libraries) [VT97, ET98].

As an initial step of HLS in a HSCD problem, the separation of hardware and software components is attempted in an iterative way. As with most computationally expensive problems, typical systems are designed as a tradeoff between performance and optimization time. Since most applications are definitely targeting primarily high performance or small system cost, it is feasible to start a system-level synthesis design from one of the possible extremes (i.e., pure software or pure hardware), and move smaller subsections between execution contexts at a time.

Hardware-software partitioning is presented as essentially equivalent to a graph coloring problem with a cost function based on edges connecting vertices of different colors. The traditional hardware-software codesign problem is binary (coloring the graph with two colors) with a hardware and a software context (color).

More complicated systems, like multiprocessor systems or multiple-board hardware, require a similar, more complicated algorithm. In these extended systems, the graph coloring problem is not binary, as the number of execution contexts is more than two. This dissertation concentrates on binary partitioning, with extensions for multiple execution contexts

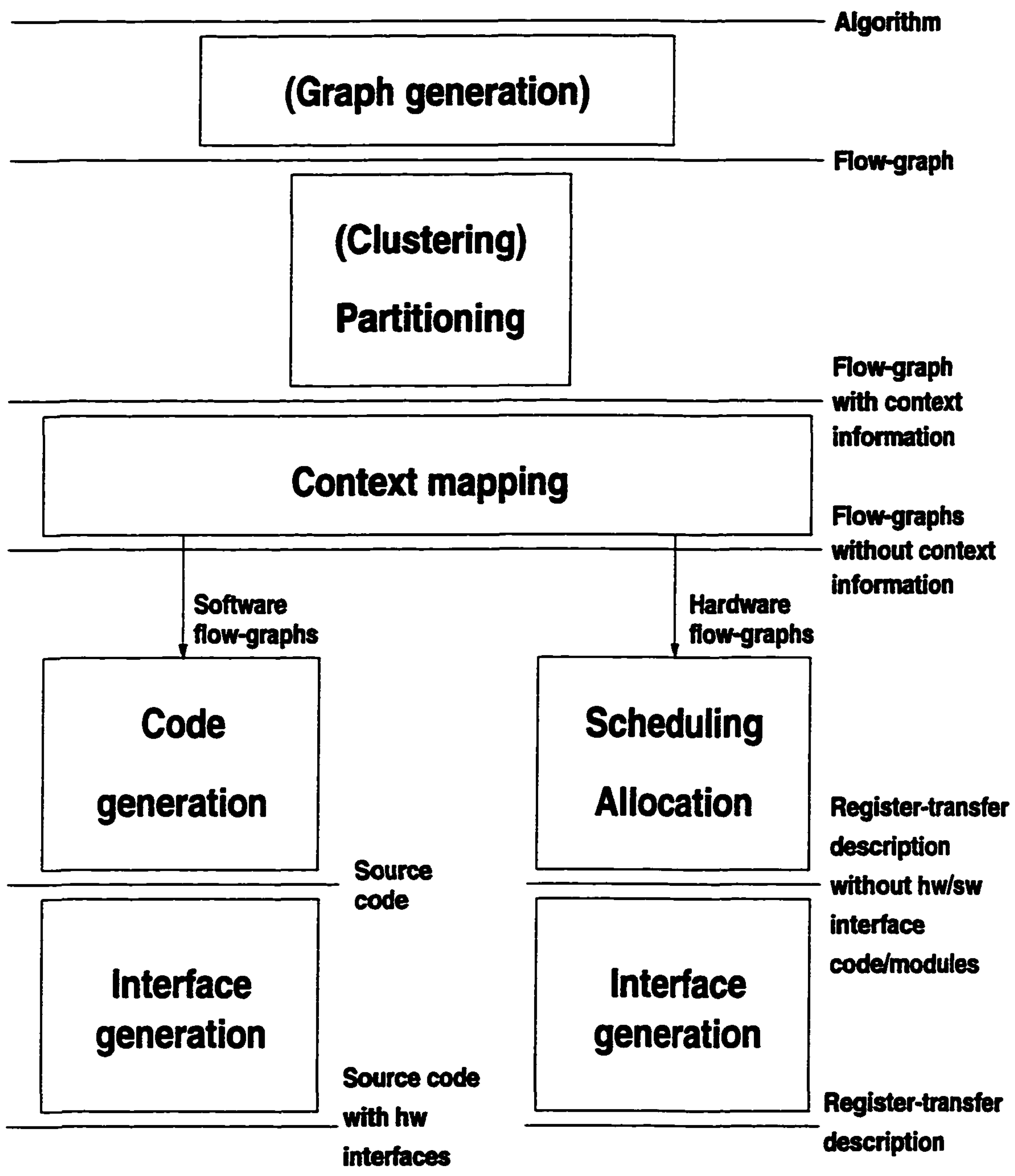

Figure 3-1: Changes of attributes and representation in multiple-context HLS

as an optional feature. As shown later, results of the dissertation use only heuristics that are capable of working on more than two execution contexts. By selecting the algorithms in such a way, we preserve the freedom of applying the dissertation results to systems beyond traditional (binary) hardware-software codesign.

Since it is desirable to reuse as much of the existing results as possible, a modular framework is beneficial to the SLS process. Extensive research has been conducted for efficient heuristics for clustering, partitioning, scheduling, and allocation. By implementing clustering, partitioning, scheduling, and allocation in separate modules, the freedom of selecting different heuristics or approximation algorithm for a given task is possible. Since scheduling and allocation stages of SLS may be realized in an integrated way, a feasible solution is to implement the clustering and partitioning stages as wrappers around the existing scheduling and allocation libraries (Figure 3-2). The clustering and partition stages simply supply input to an "external" component for scheduling and allocation. By separating different models, the task of debugging and simulation also becomes easier, since different representations of the problem become available at the boundaries between different steps of the design process. (Generating observable results during the design process may be conveniently reused during simulations and testing [Ros98].)

Since the most popular scheduling and allocation packages are usually incapable of dealing with the complications of multiple-context environment systems, the design framework transforms multiple-context environment data-flow graphs to single-context environment descriptions. It is the responsibility of the partitioning algorithm to hide the hardware-software boundary details from the scheduling and allocation stages. Since the MCHLS process describes the CDFG with context-switch information as well, data reduction is required after the partitioning stage. A step is inserted after partitioning to annotate the data-flow graph with context switch vertices. A suitable name for this step is *context mapping*, since it transforms abstract extended elementary operations ("FFT in software") to fixed numbers based on the available technology and libraries ("FFT in t_i cycles on n_i samples") (Figure 3-1).

Context mapping is the step in multiple-context high-level synthesis that expands a multiple-context data-flow graph by generating context-specific attributes to elementary operations. Context mapping also inserts context-switch vertices to model the

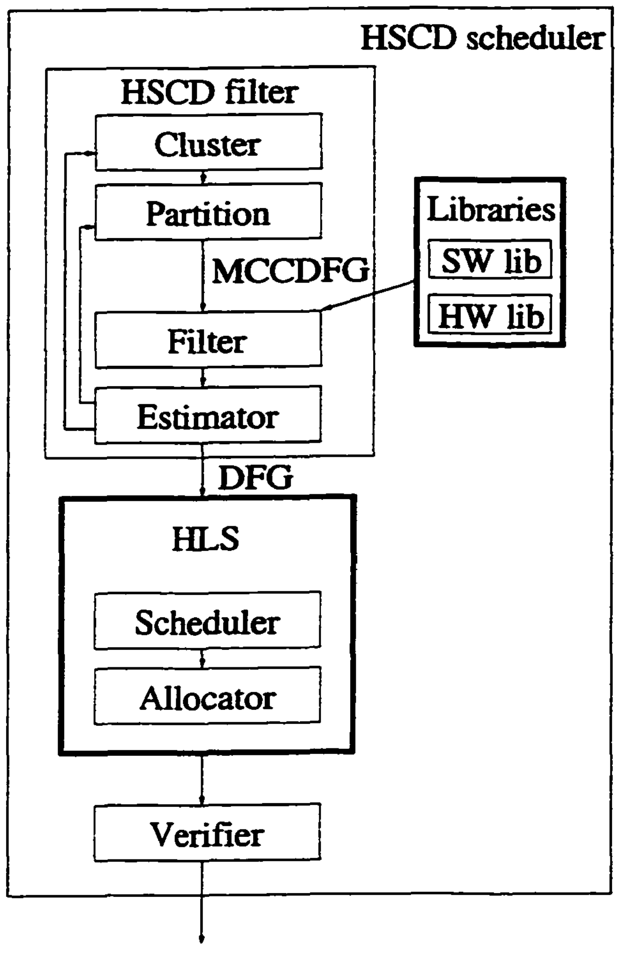

Figure 3-2: Partitioning as a filter for HLS

communications on data transfers across execution context boundaries.

The input of the data reduction is the *multiple-context CDFG (MCCDFG)*, and the output is an expanded form of the CDFG. The output is a CDFG which may be separated to execution contexts. The CDFG sections belonging to different execution contexts then may be processed directly by the scheduling and allocation stage and the code generator (Figure 3-1). Note that the compilation steps of software include instruction scheduling and allocation, but are treated as part of the compilation process, and not discussed in detail.

The scheduling and allocation phase of both software generation and hardware synthesis returns an intermediate representation. The output of software generation is source code for a state-machine description realizing the functions of the data-flow graph passed to it. This raw source code lacks further optimization, and relies on the efficiency of the target compiler environment for performance improvements. For specific systems, the code generator may be extended by knowledge of the underlying compiler to generate more efficient structures; in the current dissertation investigations no such step is taken to maintain the portability of a high-level approach.

The output format of hardware synthesis is usually *register-transfer level* information. Since most HLS design environments feature elaborate, stand-alone RTL synthesizers, RTL synthesis may be treated as a black box system, and no special consideration is made to optimize the RTL description. By relying on the services of this system-specific, non-portable level, the dissertation research remains generic enough to be useful in different environments.

3.1 Transfer model of multiple-context environments

In order to accommodate a mixed hardware-software context environment (*multiple-context environment*, MCE), one must be able to represent execution context in addition to the elementary operations of the DFG. Two trivial possibilities are:

1. attach an execution attribute to each vertex
2. label edges with context switch information.

In the latter case the destination and source contexts are encoded as an attribute of the edge (i.e., the data transfer) and context information is not represented in vertices (i.e., the elementary operations). A combination of the two methods is chosen, as representation changes from edge attributes to vertex attributes after the partitioning phase.

In the beginning, by storing the execution environment as a vertex attribute, one increases the information content of the system DFG without increasing the number of vertices. As partitioning assigns elementary operation vertices to execution contexts, changing a vertex attribute is easier since topological DFG properties (such as the number of vertices or edge information) do not change with each step.

After partitioning, context switches should be merged into the DFG so that scheduling and allocation do not have to deal with properties of the DFG data transfers. (Most popular scheduling algorithms are unable to deal with connection information if it is represented in DFG edges instead of vertices.) After such a transformation, the DFG describes the context switch as a property of a fictitious elementary operation. A suitable method of transforming context switches to vertices is to insert a "transfer" vertex to the system. *Transfer vertices* encode the context switch information of each context-switch edge in a vertex without changing data in any way. These transfers are practically delays corresponding to the cost of crossing context boundaries, and should be treated as delays in every other respect. (Additional side effects, such as byte-order reordering or synchronization on context boundaries, are assumed to occur inside the vertex, without affecting the data itself. If there is a requirement for non-trivial data transformations, they are better represented as a functional vertex in addition to the context switch.)

Since context-switch vertexs are inserted only where a data transfer crosses the boundary between two execution contexts, the increase of vertex numbers is moderate. (Especially since a good partition minimizes the cost of cuts, which is related to the number of edges crossing partition boundaries.)

Even if each additional vertex increases later optimization times, changing the representation model of context switches improves efficiency for a number of reasons:

1. All information is stored in vertices after graph generation. Data transfers (the edges of the DFG) do not contain any additional information other than direct dependencies.

Since the final structure representation relies on a single kind of data, implementation becomes easier. The only data structure a scheduler has to deal with is a set of vertices, with connections serving no other purpose than describing direct data dependencies. This is the native operation model of most scheduling and allocation heuristics.

2. As no new information is obtained on the data transfer after relocating "edge information" to "vertex information", the properties of the transfer vertices need not be changed after partitioning.

   Since vertex properties are fixed before scheduling and allocation steps, the CDFG may be optimized with algorithms that are able to handle constant execution times only. Most of the heuristic, polynomial complexity algorithms offer good scheduling and allocation properties only if vertex execution times are fixed for each elementary operation before scheduling.

   The properties that are available at the time of transformation include bit width (n), start (s) and destination (d) execution context. Based on this information, the execution time and complexity requirement of the data transfer may be expressed as

   $$t(n, s, d) \text{ and } c(n, s, d),$$

   respectively. The $t(\cdot)$ and $c(\cdot)$ functions are based on heuristic results or optimization targets as well as hardware architecture and interfaces. Both $t(\cdot)$ and $c(\cdot)$ functions are expected to be available at the start of the HLS process, and are supplied as technology libraries to the MCHLS environment.

The time and cost functions are usually highly non-linear in nature as functions of bit width or execution environments. Intel microprocessors, where 8-bit transfers may take the same amount of as 32-bit transfers are good examples of such non-linear behavior. Misaligned, heavily penalized RISC memory accesses are also typical in this sense. As an example, a PowerPC 403GC (a RISC microprocessor optimized for and widely used in embedded applications) suffers a slowdown of up to several hundred times with misaligned data since hardware hides the misaligned access by throwing an expensive exception on every misaligned memory access.

By using transfer time as the single significant attribute of context switches, the amount of information is reduced to a feasible level. As implementation details of the transfer implementation are not required until the stage of interface generation, disregarding all non-time-related attribute of context switches does not discard important information. [JRV+98] The interface synthesis at the end of the synthesis process generates the software and hardware for implementing the transmission protocol (Figure 3-4).

3.2 Multiple-context data-flow graphs

The DFG of purely hardware-oriented HLS offers a simple, yet effective way of modeling elementary operations in the algorithm. In order to accommodate the different optimization criteria of the HSCD environment, changes must be made to the elementary operation model of HLS, since the MCE of HSCD requires additional flexibility to handle HSCD-specific features of HLS. Additional features may be difficult to handle in later steps of the design process, since already existing scheduling and allocation functions are usually unable to use this information. This section describes the necessary changes to accommodate the requirements of a multiple-context environment. The section also describes data models in the multiple-context environment design process, and the reasons why changing representation is beneficial.

To keep HSCD problems at a manageable size, initial performance estimation and optimization has to stop at a relatively high level of abstraction so that the number of blocks to optimize (*problem size*) stays low. This is possible if problem decomposition does not expand the complete problem hierarchy in the beginning, or if clustering reduces system vertex count considerably. By using a top-down approach, high-level decisions affect optimization efficiency at lower levels. At a high level of abstraction, "elementary" operations are no longer truly elementary by HLS definitions, since they are not executing exactly one elementary operation in the data path. This contradicts with a basic assumption of high-level synthesis since elements of the initial data-flow graph are supposedly truly elementary in a traditional HLS process. (One of the reasons for the assumption is that optimizations are not always effective when performed in a top-down way. Expanding the DFG to truly elementary operations opens up new possibilities in optimization.)

Elementary operations are functional operations that may be realized directly with one register-transfer level primitive. (Definition replicated from p. 19)

Composite operations are functional operations that are realized by internal decomposition to multiple elementary operations.

In most practical hardware-software codesign systems, DFG vertices are not truly elementary operations during partitioning; unlike traditional HLS approaches, the vertices in the multiple-context environment DFG are usually composite. Dealing with composite operations is definitely an advantage in partitioning, but may hinder efficient scheduling and allocation. To increase the performance of scheduling and allocation, the internals of the vertices should be expanded before scheduling.

As an example of elementary and extended elementary operations, consider a popular DSP application, *GSM speech processing*. GSM cellular phones feature a sophisticated model of speech compression with finite state machines providing correction to the reconstructed data stream. A complete block diagram of the prediction unit (GSM Regular Pulse Excitation - Long Term Prediction (RPE/LTP) encoder) consists of 19 functional blocks [Pin96, p. 22], which is useful for partitioning purposes (Figure 3-8, p. 99). None of these blocks are primitive in the HLS sense, since they feature decoders, quantizers, and basic filters. (In fact, several of the blocks are used as standalone benchmarks for HLS. Two examples are the inverse filter (e_9) and the weighting filter (e_{13}).)

Optimizing a high-level representation (with few vertices) is definitely useful as an input to the partitioning process, since individual blocks tend to show very high locality of data. Partitioning functional blocks with extremely high data locality should definitely not attempt to put context boundaries inside such blocks. The example system, as verified by experimentation, is an example where the initial representation has been clustered effectively. Note that in this system, for the passes of scheduling and allocation, the functional blocks should be expanded to true elementary operations. (This approach is taken in [Pin96], where functionality is described as a occam source code, and partitioning occurs at a higher level, roughly corresponding to the first level of functions in the source code. Clustering at a function level is efficient since local storage is hidden inside clusters, and local variable manipulations do not have to pass through context-switches.)

In the dissertation, for the ease of discussion, composite operations will be called "extended elementary operations" (EEOGs), regardless of their actual implementation. Similarly, the CDFG of composite operations is referred to as the *extended elementary operation graph* to differentiate between truly elementary and composite operations. Both elementary and extended elementary operations are denoted with e_i in the corresponding graph.

After partitioning, extended elementary operations should be expanded to their full internal structure, and scheduling and allocation should use this "full" (i.e., truly "elementary operation") representation. The elementary operation graph takes its final form just after technology mapping, where extended elementary operations get replaced by their internals, to enable global scheduling and allocation on the whole design.

The step following scheduling and allocation, *register-transfer level (RTL) synthesis* is expected to deal with actually generating the necessary primitives even for composite operations.

The operation model in traditional HLS assigns the following properties to elementary operation e_i:

Start time, v_i, the time operation e_i receives all its input data and may start processing it. By time cycle v_i all inputs are assumed to be available and stable. Timing information is generally expressed in dimensionless units, *time cycles*, making designs scalable with technology [PK89]. Time cycles are numbered from 0, as usual.

Successor set, S_i, a set of elementary operations that use the output value of e_i directly (*immediate* or *direct successors*).

Direct successor: An elementary operation e_j is a direct successor of e_i if and only if at least one of the data inputs of e_j is the output of e_i.

The following notation is used to denote direct successor relationships:

$$e_i \rightarrow e_j$$

"Immediate successor" is used interchangeably with "direct successor" in this dissertation. The inverse relationship is *direct predecessor* or *immediate predecessor*:

Direct predecessor: An elementary operation e_i is a direct predecessor of e_j if and only if $e(j)$ is a direct successor of e_i. "Immediate predecessor" is used interchangeably with "direct predecessor" in this dissertation.

Using the above notation, the definition of the successor set is

$$S_i = \{e_j : e_i \to e_j\}$$

The successor set is empty if and only if the elementary operation supplies a system output (assuming there are no loops in the system). Similarly, a predecessor set is defined:

Predecessor set, P_i, a set of elementary operations that are direct predecessors of e_i:

$$P_i = \{e_j : e_j \to e_i\}$$

Execution time, t_i, the total time required for the elementary operation to produce its output.

Even if a considerable set of scheduling and allocation heuristics exists for constant-execution time systems, allowing different execution times for elementary operations increases the usefulness of a hardware-software codesign design environment. Practically all models of real hardware-software codesign systems require different execution times for efficient designs. Also, heuristics manipulating only uniform execution time elementary operations are usually based on integer linear programming (ILP). ILP-based scheduling and allocation algorithms tend to be impractical for real systems because of the prohibitive cost of actually implementing the solutions [HLYC91, KL97].

Elementary operations may start processing their inputs when all of their immediate *predecessors* have produced their output. In other words,

$$\min v_j = \max_{e_i : e_i \in P_j} v_i + t_i$$

The following additional conditions are applicable to timing calculations (Figure 3-3):

- e_i requires all its input data to hold a stable value during the whole duration

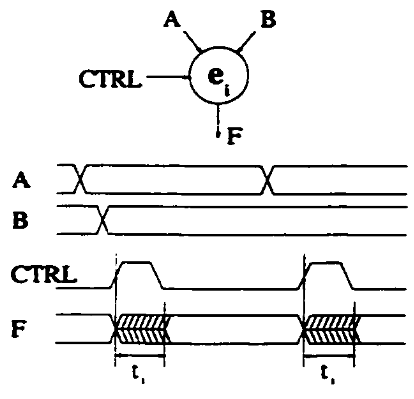

Figure 3-3: Elementary operation timing diagram

time t_i.

- e_i may change its output during the whole duration time t_i.
- e_i holds its actual output stable until its next start.

Execution time is assigned after partitioning, and may depend on data (operation) size as well as operation type (j_i).

Operation type, j_i, identifies the resource requirement of elementary operations. Operations of the same type may be executed in the same kind of processor [PK89].

Note that the choice of operation types depends on the environment. In some systems, for example, multiplication and addition are performed in general-purpose ALUs, thus $j_i = j_k$ for an addition and a multiplication. In most practical systems, multiplication and addition are separated to reduce cost; in these systems the operation types obviously differ for addition and multiplication vertices.

The operation type attribute is used during scheduling and allocation, and is a property of the initial problem graph (i.e., it is fixed at the time of graph generation, and does not change later).

It must be noted that the definition of operation type has to be extended in MCHLS. Making an operation type attribute depend on the execution context as well makes allocation easier later, as shown below.

ASAP execution time, s_i, denotes the first possible time cycle elementary operation e_i may be started. Similarly,

ALAP execution time, l_i, denotes the last possible time cycle elementary operation e_i may be started.

ASAP and *ALAP* are the starting time cycles for the trivial greedy scheduling strategies, *As Soon As Possible* and *As Late As Possible.* ASAP attempts to start execution of elementary operations as soon all their predecessors become available. ALAP delays starting an operation as late as possible, triggering the elementary operation at the last time cycle when system latency is not increased.

In addition to the properties of elementary operations in HLS, the following attributes have to be represented in the multiple-context data-flow graph:

Execution context, x_i, denotes the environment in which e_i is executed. This attribute is assigned during partitioning, and may not change later. After context mapping, the execution context attribute is not referenced directly, since all operations will be defined in terms of timing and resource usage, which is sufficient for practical scheduling and allocation steps. (In other words, in the MCHLS design process, the significant information about "software FFT" is limited to its execution time and operation type.)

Operation complexity, n_i, is the significant data size of the input data of e_i. This attribute is important since it may influence implementation costs, operation timing (see below) and also since it affects requirements for data transfers. A typical measure of n_i is the bit width of input values, but other measures of complexity are possible.

Depending on the operation type, inputs of different sizes may be present in each elementary operation. In this case, operation complexity is chosen to be size of the representative input. As an example, the data width of a multiplexer (or decoder) is a better representation of multiplexer (or decoder) complexity than the number of control bits.

Operation category, k_i, is similar to operation type j_i, but it is an initial property of the elementary operation. An example operation category could be "Fast Fourier Transform (FFT)" without reference to implementation. Initial operation type depends only on the elementary operation itself and has no connection to the execution context of the given vertex. The operation category is used only during partitioning and context mapping, where operation category and execution context are combined to generate the operation type of the elementary operation. To represent the differences between software and hardware implementations, one must reconsider the properties of operation type as well:

Operation type, j_i, in multiple-context environments includes execution context as well. As an example, a software Fast Fourier Transform (FFT) as an extended elementary operation is definitely different from a hardware implementation for modeling purposes, and the operation types of e_i (a "software FFT" vertex) and e_j (a "hardware FFT" vertex) are different, even if both are FFT implementations (i.e., $k_i = k_j$).

Operation type in multiple-context environments may be determined after the partitioning step, and is done in the context mapping step. For each operation of the data-flow graph, the context mapping assigns the operation type as a function of vertex execution context and operation category:

$$j_i = j_i(k_i, x_i)$$

For accurate calculations, the operation type may include operation complexity as well. This is the case, for example, if the underlying hardware modules may be generated as different models based on complexity. A practical example could be multiplication under primary time constraints (as well as secondary limits on silicon area). Implementing a parallel multiplier in such an environment is feasible only up to a complexity limit, since the silicon area of a straightforward parallel multiplier grows as a quadratic function of complexity. Since the primary constraint is time, implementing parallel multiplication is a good choice since time is not increased if complexity increases. Once the complexity requires too much silicon, multiplication must be performed using less silicon area, which requires different implementations (such as serial multiplication). It is definitely an advantage if multiplications under a given size are represented as different operations from those over the size limit. Even if the benchmark applications did not exploit this possibility, *HSPIPE* is capable of

Property	Notation	Assigned during
Start time	v_i	scheduling
Predecessor set	P_i	graph generation
Successor set	S_i	graph generation
Execution time	t_i	graph generation
Operation type	j_i	graph generation
ASAP execution time	s_i	scheduling
ALAP execution time	l_i	scheduling

Table 3.1: Elementary operation attributes in HLS

dealing with this kind of variable operation types. (Users must supply the necessary parameters with the descriptions of elementary operation types in order to utilize this.)

The idea of different operation types for different operation complexities may be extended to consider different implementations based on the available time frames of operations . Since the necessary heuristics of these extensions depend on implementation details, they are not included in the dissertation investigations.

The operation type attribute is used in the scheduling and allocation steps. Assigning different operation types to elementary operations of the same category and different execution contexts speeds up allocation. Since two elementary operations of the same category, when mapped to different contexts, do not compete directly for the same resources, the allocation process should not set up concurrency relations between them. Using the operation type as an allocation attribute (i.e., if $j_k = j_l$, e_k may compete with e_l for system resources), the number of constraints in the system is decreased. Thus, even if $k_5 = k_{15}$ (both e_5 and e_{15} use "quantizer" resources), the allocator heuristic need not set up a constraint on the concurrency of e_5 and e_{15}, as $x_5 \neq x_{15}$. In other words, even if the execution time of e_5 and e_{15} overlaps, there is no resource violation problem since e_5 (mapped to software) ties up a CPU resource, while e_{15} occupies a piece of ASIC circuitry in the same time cycles (in hardware).

Using the above notations, the following equalities may be set up as timing constraints in the system:

Property	Notation	Assigned during
Start time	v_i	scheduling
Predecessor set	P_i	graph generation
Successor set	S_i	graph generation
Execution time	t_i	context mapping
Operation type	j_i	context mapping
ASAP execution time	s_i	scheduling
ALAP execution time	l_i	scheduling
Execution context	x_i	partitioning
Operation complexity	n_i	graph generation
Operation category	k_i	graph generation

Table 3.2: Elementary operation attributes in MCHLS

1. Every elementary operation must start execution not sooner than its ASAP time cycle, and not later than its ALAP cycle:

$$s_i \leq v_i \leq l_i$$

2. No elementary operation may be triggered before every one of its predecessors has finished executing:

$$s_j = \min v_j = \max_{e_i : e_i \in P_j} v_i + t_i \tag{3.1}$$

3. ASAP times of elementary operations may be found by calculating the maximum of the sum of ASAP cycles and execution times of their direct predecessors. The resulting equation is the special case of (3.1):

$$s_j = \begin{cases} \max\limits_{e_i : e_i \in P_j} s_i + t_i & \text{if } P_j \neq \emptyset \\ 0 & \text{if } P_j = \emptyset \end{cases} \tag{3.2}$$

Note that direct input operations have no predecessors, and their ASAP cycle is cycle 0 by definition. ASAP time cycles for other elementary operations may be found in an inductive way based on (3.2).

4. A similar equation holds for ALAP time cycles. In this case, the ALAP time of every elementary operation must enable the direct successor vertices to finish execution be-

fore their ALAP times. The ALAP times of elementary operations producing system outputs is fixed so that the last system output is stable by time cycle L_0, where L_0 is a preset constant.

$$l_j = \begin{cases} \min\limits_{e_i : e_i \in S_j} s_i + t_i & \text{if } S_j \neq \emptyset \\ L - t(j) & \text{if } S_j = \emptyset \end{cases}$$

Elementary operations supplying system output values are assumed to have an ALAP value set by system requirements if there is a latency constraint (L). In systems without latency limits, all resource requirements may be fulfilled, since inserting sufficient delay would resolve all resource conflicts. Such a solution, is not practical in most systems.

3.3 Multiple-context HLS design process

The multiple-context data-flow graph described in the previous sections is a suitable extension to traditional high-level synthesis system designs. Assuming familiarity with the heuristics described in in Chapter 2 and the basic steps of high-level synthesis, this section describes the extended multiple-context design process, *multiple-context high-level synthesis* (MCHLS).

As discussed in Chapter 1, the dissertation research targets synthesis from data-flow graph descriptions, namely, a CDFG description of the problem. The design process terminates with a scheduled, allocated data-flow graph for hardware execution context(s), with additional control information extracted from the CDFG (to generate the controller logic). Software synthesis terminates at raw source code level, where a straightforward code generator maps the functionality of the software subsystems to unoptimized, raw source code. In the initial version of the dissertation results, no attempt has been taken to optimize source code in any way, as the task of improving performance is left to the compiler. It must be noted that the optimization process (namely, scheduling/allocation) attempts to consider software performance during performance estimations. In the initial model, a straightforward, CISC implementation is assumed without the dynamics of RISC CPUs. Extending the system to handle optimization in RISC architectures requires changes in one module library only ("Technology library" in Figure 4-1), without affecting the design framework in any other way.

The *design phase* of MCHLS should be different for final production runs and the iterative partitioning process. Since efficient scheduling and allocation tends to last longer than fast solutions (such as list scheduling), the time requirement of high-performance heuristics may be prohibitive, especially if the partitioning process evaluates a large number of partitions. On the other hand, saving time during the final synthesis phase may increase system cost considerably because of lower performance of extremely fast heuristics. To maintain a balance of design time and performance, the design process is assumed to differ for iterations and production runs.

A feasible way of reducing the runtime of the partitioning iterations is to apply an efficient algorithm to extrapolate fast, inefficient scheduling and allocation results to results obtained by slower, better heuristics. Implementing such a set of heuristics, one is able to reduce the execution time of the partitioning iterations by using fast, inefficient heuristics followed by a fast correction step (Figure 3-4.). Since the iteration rounds do not have to actually produce efficient solutions, just give estimations on their distance from the optimum, such an approach is definitely useful for MCHLS. (Instruction scheduling and allocation are NP-complete problems where good guesses may be given to the quality of results even if the optimum solution is unknown.) In fact, the same idea is present in linear cryptoanalysis, where highly nonlinear (cryptographic) functions are approximated with linear approximations and small, faster nonlinear corrections [Sch95]. (The goal of applying such corrections is very different from MCHLS.)

3.3.1 Iterative steps

The MCHLS design cycle is assumed to consider a large number of possible partitions to select one that may be used to synthesize an efficient implementation. Unfortunately, there are at least two additional NP-complete steps after partitioning and before register-transfer level synthesis. Since both of these steps (scheduling and allocation) must be finished before the efficiency of the partition may be evaluated, the scheduling and allocation methods should terminate fast. On the other hand, not surprisingly, the efficiency of scheduling

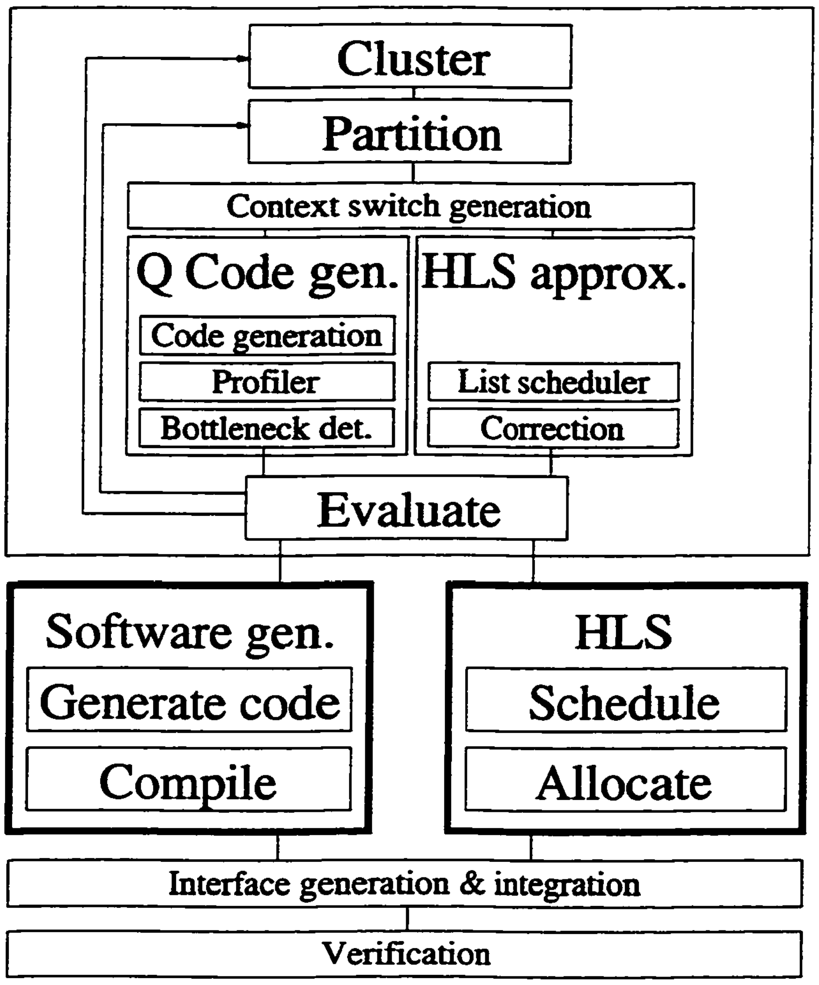

Figure 3-4: Evaluation and iteration steps in switched-context high-level synthesis

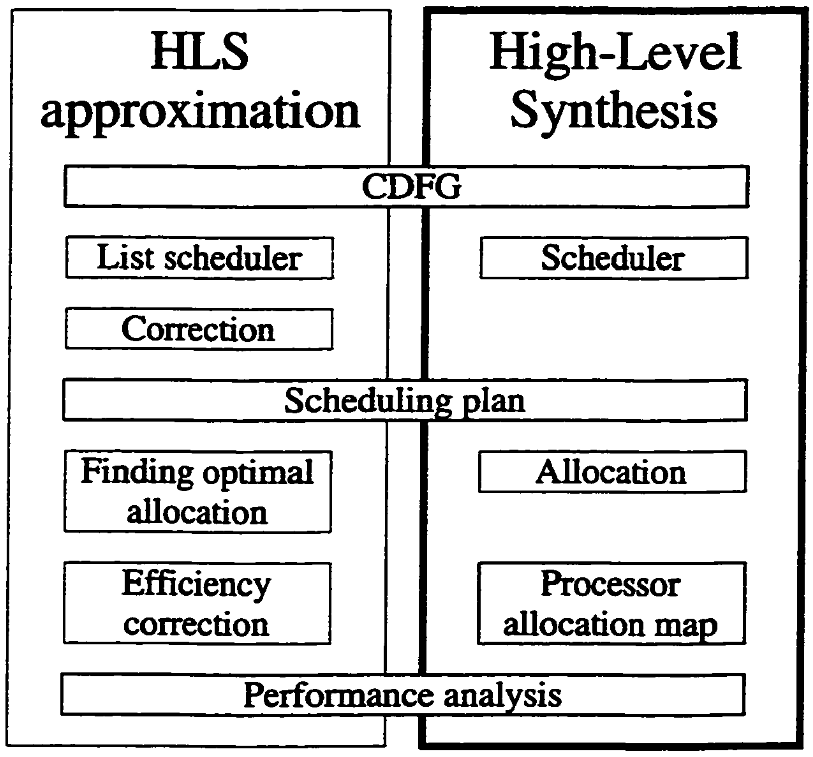

Figure 3-5: Approximation, scheduling and allocation of hardware modules

algorithms tends to decrease with decreasing time complexity [Hoc97, "Approximation algorithms for scheduling", p. 1]. The relationship between runtime and efficiency of allocation algorithms is very similar.

To reduce the runtime of the MCHLS design process, the number of iterations where slow, efficient heuristics are applied must be minimized. To achieve this, two additional steps are introduced to the design process after context mapping: a very quick profiling phase for software sections and a fast, inefficient list scheduler with the related allocation steps (Figure 3-4). These steps are assumed to produce a guess on the output of optimization that would be produced by a better (slower) heuristic on the same subsystem. Once the results of these fast approximations are satisfactory, the design process may continue to the final code generation and HLS. (Additional rounds of verification are needed after these steps, to make sure that the result extrapolated from the fast approximation was correct.)

Approximations of software runtime and performance may be found in a way that is similar to the approximation of HLS efficiency. (This is not surprising, since the steps of compiler optimizations use some of the same features, namely instruction scheduling and register allocation.) Instead of software scheduling and allocation, a better approach could be a quick implementation of code generation from an elementary operation graph and then a profiling pass. Assuming the ASAP and ALAP time cycles of the elementary operations are known, a probabilistic description of the system resource usage may be found. (A similar approach is described in [PK89], where parallelism is described as a set of distribution functions, with resource usage bounded below by the maximum of these functions. Even if the above, *force-directed* scheduling algorithm was developed for hardware systems, the same may be applied to software as well.) By matching the theoretical parallelism to the results of profiling, one might be able to construct approximations of the results of more detailed, slower software optimization algorithms (Figure 3-6).

Assuming the code generation phase is fast, most of the speed advantage of the fast performance approximation comes from the difference between scheduling and profiling. Even if scheduling may reuse some of the information from previous steps, a simple profiling pass is definitely faster than running a higher-order polynomial scheduling algorithm. (An example of information passed to the scheduler from previous passes may be the structure of the CDFG. In fact, this is done in the final implementation by embedding structured comments in the generated source code. The performance gain is significant, since most of the complexity of the applied list scheduling heuristic is in building the dependency matrix, an $O(n^2)$ step in a straightforward implementation.)

In the target environment of *HSPIPE*, a more effective approach was found. Since the current target microprocessors are legacy, embedded CISC devices, the complexity of the compilation process was much lower than a contemporary, RISC core would have required. (In fact, generated code and compiled binaries for the Intel 80C51 and similar microprocessors tends to be very regular and predictable.) Because in this case code generation and compilation are both easy to describe, a more efficient way of estimation code efficiency was found. Most of the reduction in time came from the fact that software and hardware runtime estimations could be partially merged.

The software and hardware estimations are merged by relying on a user-supplied technology library describing software execution in terms of hardware. This library should

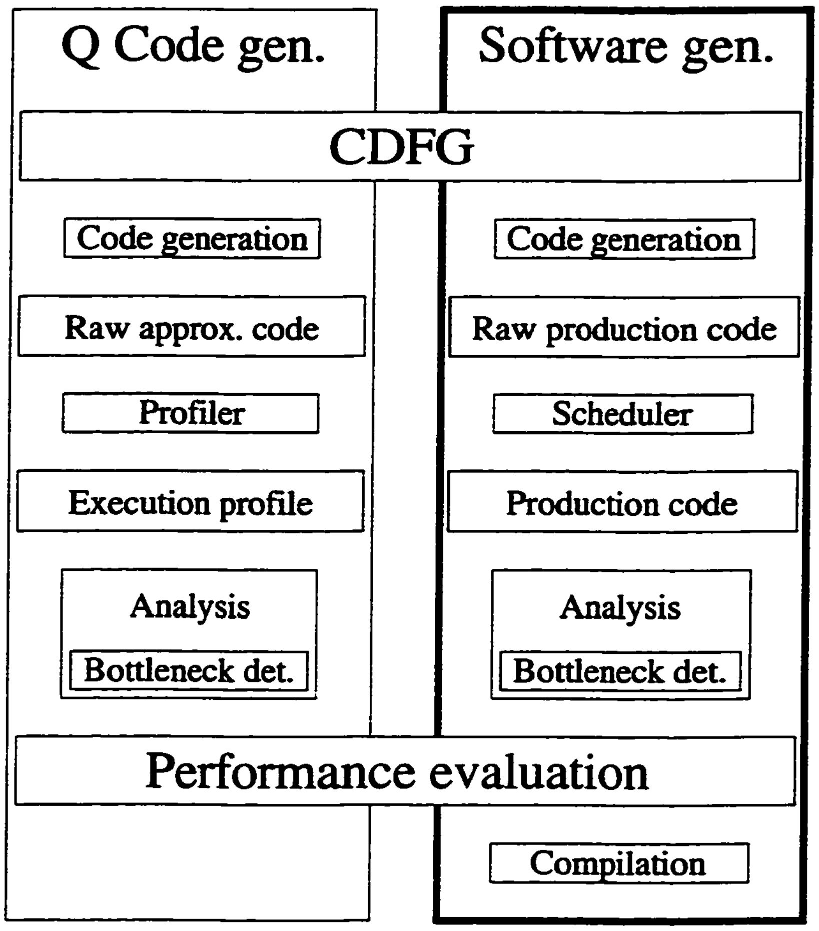

Figure 3-6: Approximation and production code generation of software modules

provide timing information and resource usage of instructions, using metrics of the hardware environment (i.e., expressing execution times in the same time cycles, etc.). Describing software execution in such a way, software execution context(s) may be merged with hardware approximations, since software and hardware components may be simulated in the same scheduler and allocator. Assuming the interface generator measures time in the same units, the whole hardware-software system may be described in the same time scale.

As well as scheduling a mixed hardware-software system, allocating resources to elementary operation may be done in a similar, integrated way after context mapping. The primary problem of allocation is the separation of operations that may not conflict because of their different contexts. Such could be an example where software and hardware FFTs would be present in the system EEOG.

Note that in the current implementation, there is no support for RISC architectures. Should the target environment include RISC or higher-performance CISC architectures (which tend to rely on more RISC features as time passes) in the future, the current software estimation should be upgraded.

As an example, a stripped-down version of a practical system is used. The block diagram in Figure 3-8 is a section of a GSM speech coder circuitry [Pin96, p. 22]. This part of the GSM DSP application is often used as an example of practical hardware-software codesign systems because different versions of the GSM cellular phone standard are extremely popular. Because of the enormous size of the GSM market, telecommunications-related applications tend to be thoroughly analyzed for cost optimization, and the industry sees immediate applications of hardware-software codesign.

The example used in this chapter is only a section of the GSM coder since the whole system would be too large for effective demonstration in the text. (Certain submodules of limited functionality, such as adders and shifters, have been removed from the original block diagram to increase readability.) Containing 17 blocks (extended elementary operations), it may be transformed easily to an extended elementary operation graph. (An additional box of limited functionality is required to make the extended elementary operation graph complete; this is e_{18} in Figure 3-9.)

The primary purpose of the example is a typical DSP application; it is a fixed-point

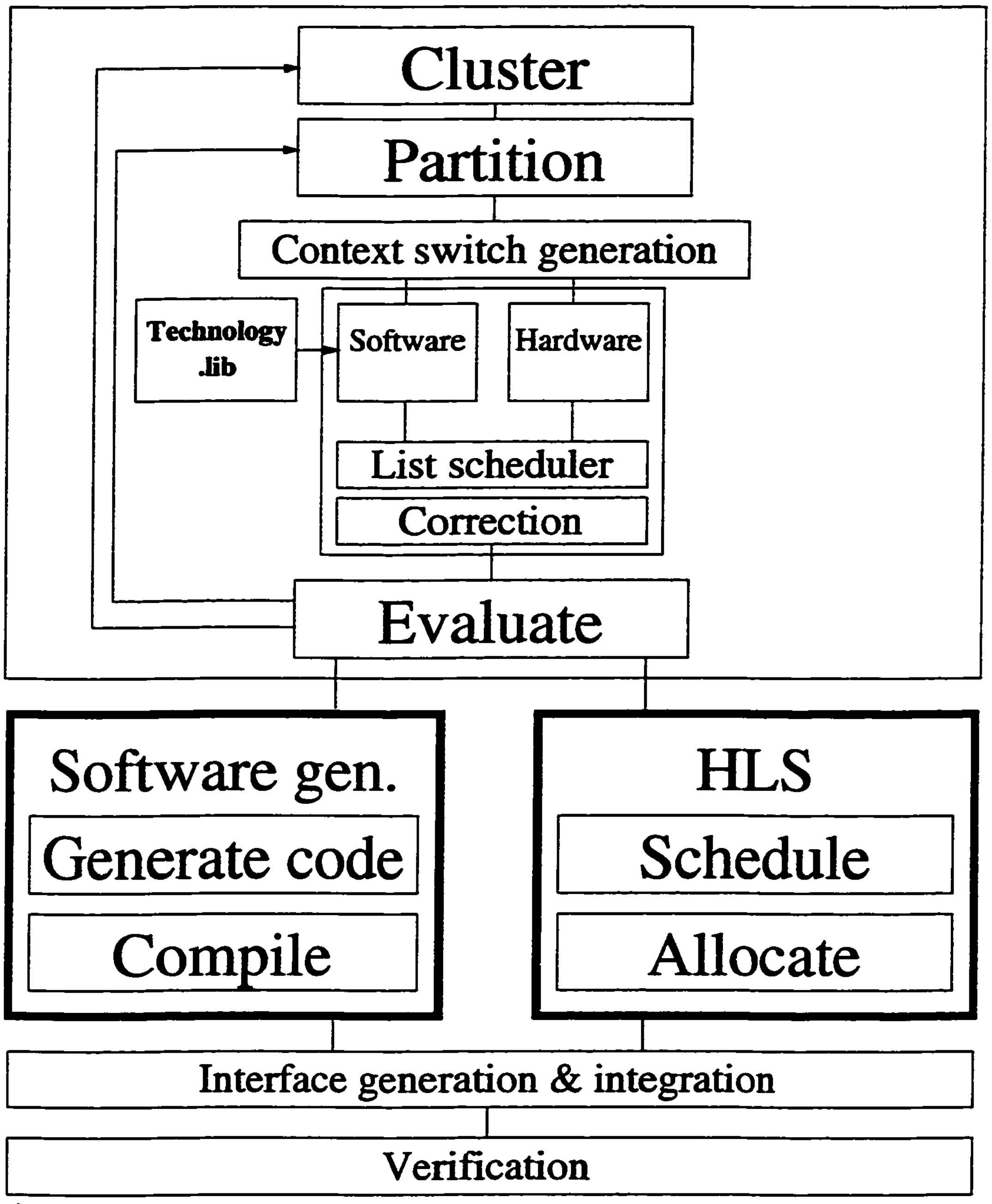

Figure 3-7: Evaluation and iteration steps without explicit software profiling

Linear Predictive Coder (LPC) where 160 13-bit resolution sound samples are compressed to 260-bit encoded blocks. The algorithm calculates an estimated time signal based on previous sound samples, and passes on the difference between the estimated time sequence and actual samples.

The blocks in Figure 3-8 are treated as extended elementary operations at the partitioning phase. (Numbers of blocks correspond to extended elementary operation numbers in the extended elementary operation graph, Figure 3-11)

The steps taken during iterative rounds may be executed in the following way:

1. Construct original DFG from problem description, source code, or prescribed dataflow graph. This initial DFG shall finalize only the following properties:

   - Initial operation type for vertices, k_i, without any context information (i.e., "FFT"), with no note on execution context or complexity. (Table 3.3, p. 101)
   - Operation complexity, n_i. This attribute is important only to operations where execution time or cost depends on the size of the incoming data.
   - Direct data dependencies, i.e., the set of $(e_i, e_j) : e_i \rightarrow e_j$ pairs. The properties of the data dependencies include the two vertices, and the bit width of the data transfer (equal to the operation complexity of the destination operation, n_j).

   The original attributes of the GSM example are given in Table 3.3. The following direct dependencies are present: $e_1 \rightarrow e_2$, $e_1 \rightarrow e_9$, $e_2 \rightarrow e_3$, $e_3 \rightarrow e_4$, $e_4 \rightarrow e_5$, $e_5 \rightarrow e_6$, $e_6 \rightarrow e_7$, $e_7 \rightarrow e_8$, $e_8 \rightarrow e_9$, $e_9 \rightarrow e_{10}$, $e_9 \rightarrow e_{13}$, $e_{10} \rightarrow e_{11}$, $e_{11} \rightarrow e_{12}$, $e_{12} \rightarrow e_{18}$, $e_{13} \rightarrow e_{14}$, $e_{14} \rightarrow e_{15}$, $e_{16} \rightarrow e_{17}$, $e_{17} \rightarrow e_{18}$, $e_{18} \rightarrow e_{10}$. The listing excludes system inputs and outputs. (The dependencies of elementary operations and system ports are used only in calculating ASAP and ALAP times.)

2. Partition the graph into execution contexts. This step divides the DFG into hardware and software parts. Partitioning generates the extended elementary operation execution contexts (x_i).

   The techniques described in Section 2.2.2 are useful for the partitioning step. During benchmarking, modified versions of the Kernighan-Lin algorithm have been used (including the most popular enhanced variant, the Kernighan-Lin-Fiduccia-Mattheyses

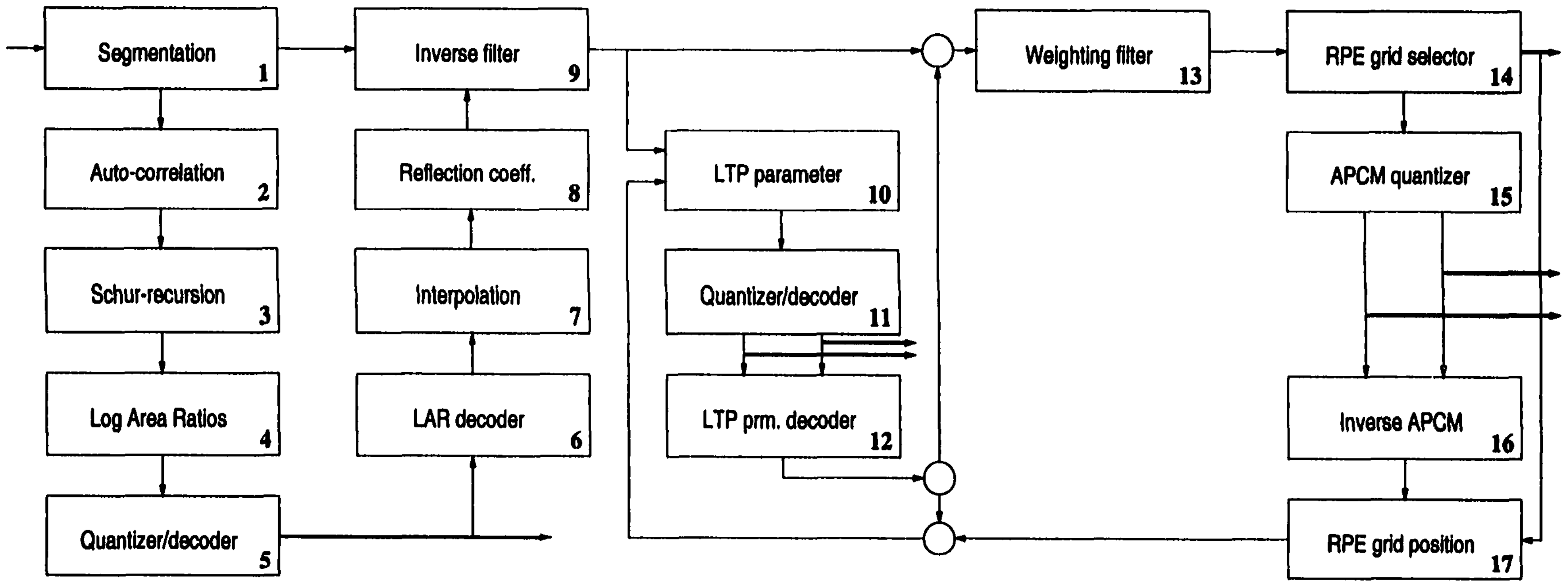

Figure 3-8: Block diagram of GSM example

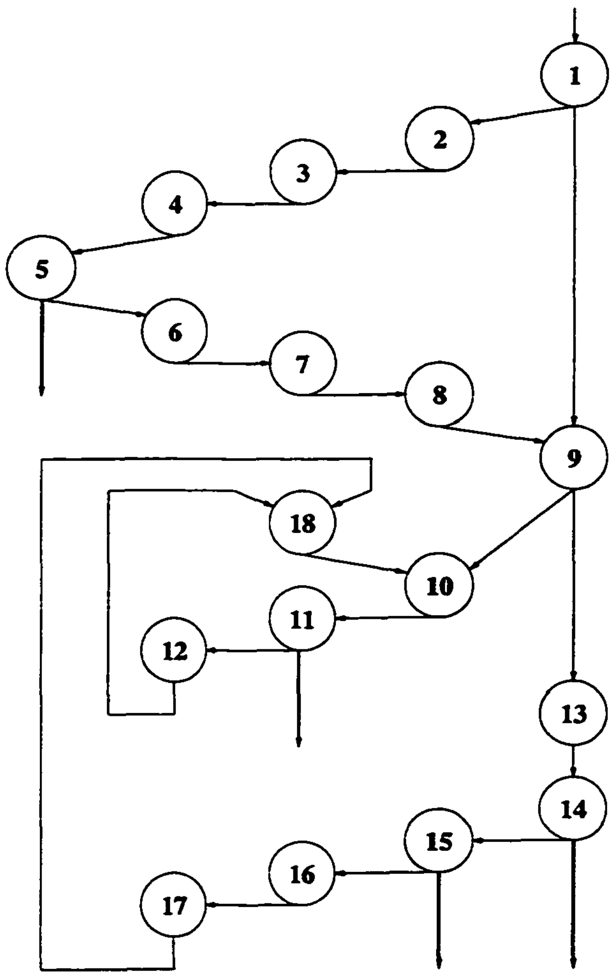

Figure 3-9: Extended elementary operation graph of GSM example

i	k_i	k_i (symbolic)	**Name in GSM block diagram**
1	1	segm	Segmentation
2	2	filter	Auto-correlation
3	3	schur	Schur-recursion
4	4	param	LAR calculation
5	5	quant	Quantizer/coder
6	6	decoder	LAR decoder
7	7	interp	Interpolation
8	8	coeff	Reflection coefficients
9	2	filter	Inverse filter
10	4	param	LTP parameter
11	5	quant	Quantizer/coder
12	6	decoder	LTP parameter decoder
13	2	filter	Weighting filter
14	9	grid	RPE grid selection
15	5	quant	APCM quantizer
16	5	quant	Inverse quantizer
17	9	grid	RPE grid positioning
18	10	adder	Adder

Table 3.3: Extended elementary operation attributes of GSM example

algorithm), but the modularity of the partitioning step makes it easy to upgrade to other algorithms.

3. Insert context-switch vertices to the system by replacing edges crossing execution context boundaries. The context-switch vertices are unary operations splitting context-switch vertices in two (Figure 3-12). Timing values of context-switch vertices depends on destination and source contexts, bit width, and communication protocol. This information is supplied by the used in context mapping technology files.

   Because of the timings of the context switch vertices, timing relations (ASAP and ALAP times, time dependencies) must be calculated after this step.

4. Expand the internals of extended elementary operations. Generate the necessary elementary operation graphs and merge them by replacing each with their subsystems' graphs.

   As an example, an internal section of the weighting filter is presented in Figure 3-10. The filter itself is a FIR filter with tabulated coefficients. It takes 40 samples of the

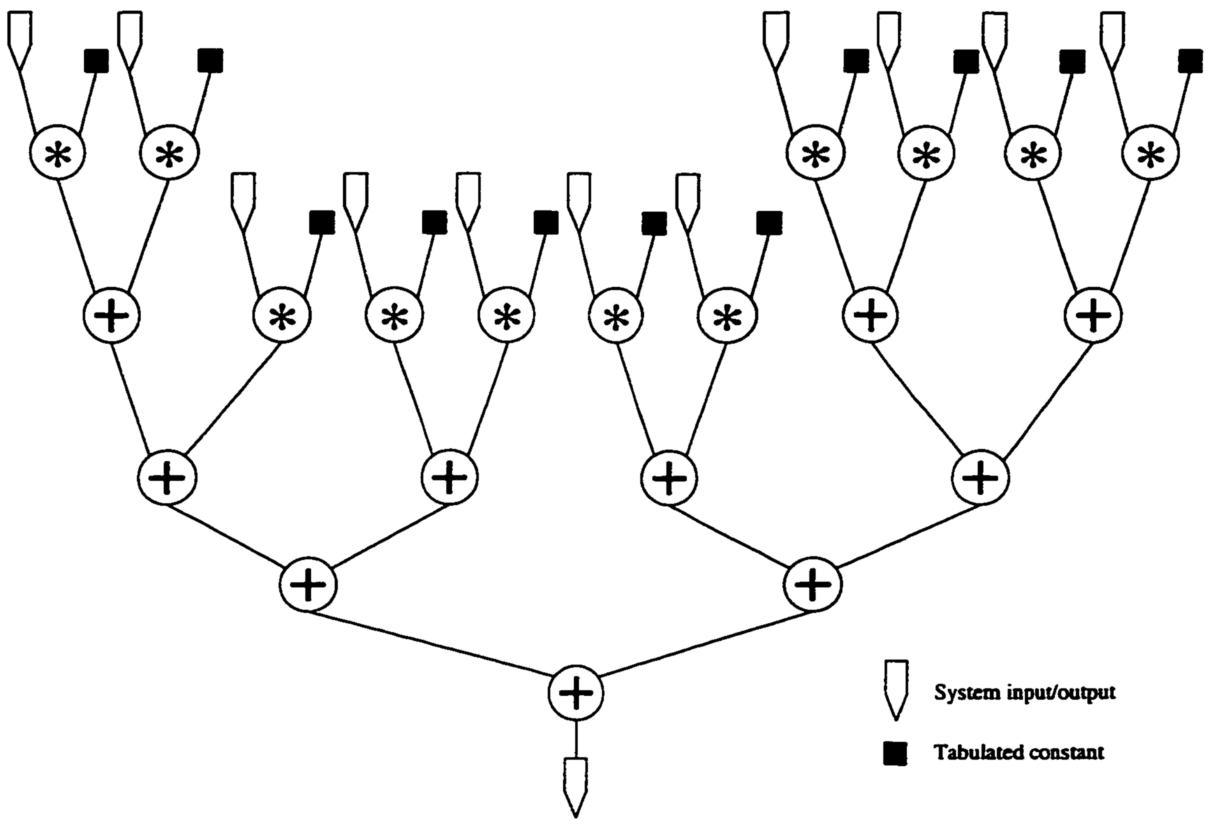

Figure 3-10: Section of FIR weighting filter

sound ($a_0 \dots a_{39}$) and outputs the same amount of samples ($b_0 \dots b_{39}$). (Even if the results would have 50 total samples theoretically, the filter output is truncated by discarding the first and last five samples.) The structure in Figure 3-10 has to be replicated for each filtered value. Each structure implements the equation

$$b_k = \sum_{i=0}^{10} H_i \cdot a_{k+5-i} \qquad k = 0 \dots 39$$

where the H values are tabulated constants. (The structure in Figure 3-10 has been generated by the algorithm described in Appendix A; the generation target was a maximum multiplier concurrency of six units.)

5. Assign final operation types to elementary operations. Since execution contexts are fixed by partitioning, the final execution type ("FFT in hardware") may be assigned

to elementary operations:

$$j_i = j_i(k_i, x_i)$$

The sample system, with the partition shown in Figure 3-11, has the final operation types shown in Table 3.4, p. 105.

In some systems, where processors are specialized for different data sizes, the final operation type may depend on operation complexity as well:

$$j_i = j_i(k_i, x_i, n_i)$$

In the current *HSPIPE* implementation, such a step is not taken. This serves the purpose of maintaining portability, since taking operation complexity into consideration would definitely increase the dependence on the underlying RTL synthesis tools.

Note that in the GSM example, the same functionality in different contexts has a different operation type. Even if functionally identical (or very similar) operations, e_5 and e_{15} have to be treated differently because of their different execution contexts ($x_5 \neq x_{15}$). For this reason, they are mapped to different final execution types (i.e., $j_5 \neq j_{15}$). (The same applies to e_{14} and e_{17}.)

6. Assign execution times to elementary operations. Execution times generally depend on operation complexity and type:

$$t_i = t_i(j_i, n_i)$$

7. Insert transfer vertices to the system, where applicable. The new vertices behave as transfer elementary operations, and do not count as functional operations. The new vertices' only practical attribute is their execution time, which is a function of the context switch source and destination, and the operation complexity of the data destination:

$$t_k = t_k(x_i, x_j, n_j)$$

An important part of system descriptions is the set of context-switch timing functions, which assign the time required to transfer the given number of bits through the context boundary. These functions may be given as explicit functions, timing diagrams,

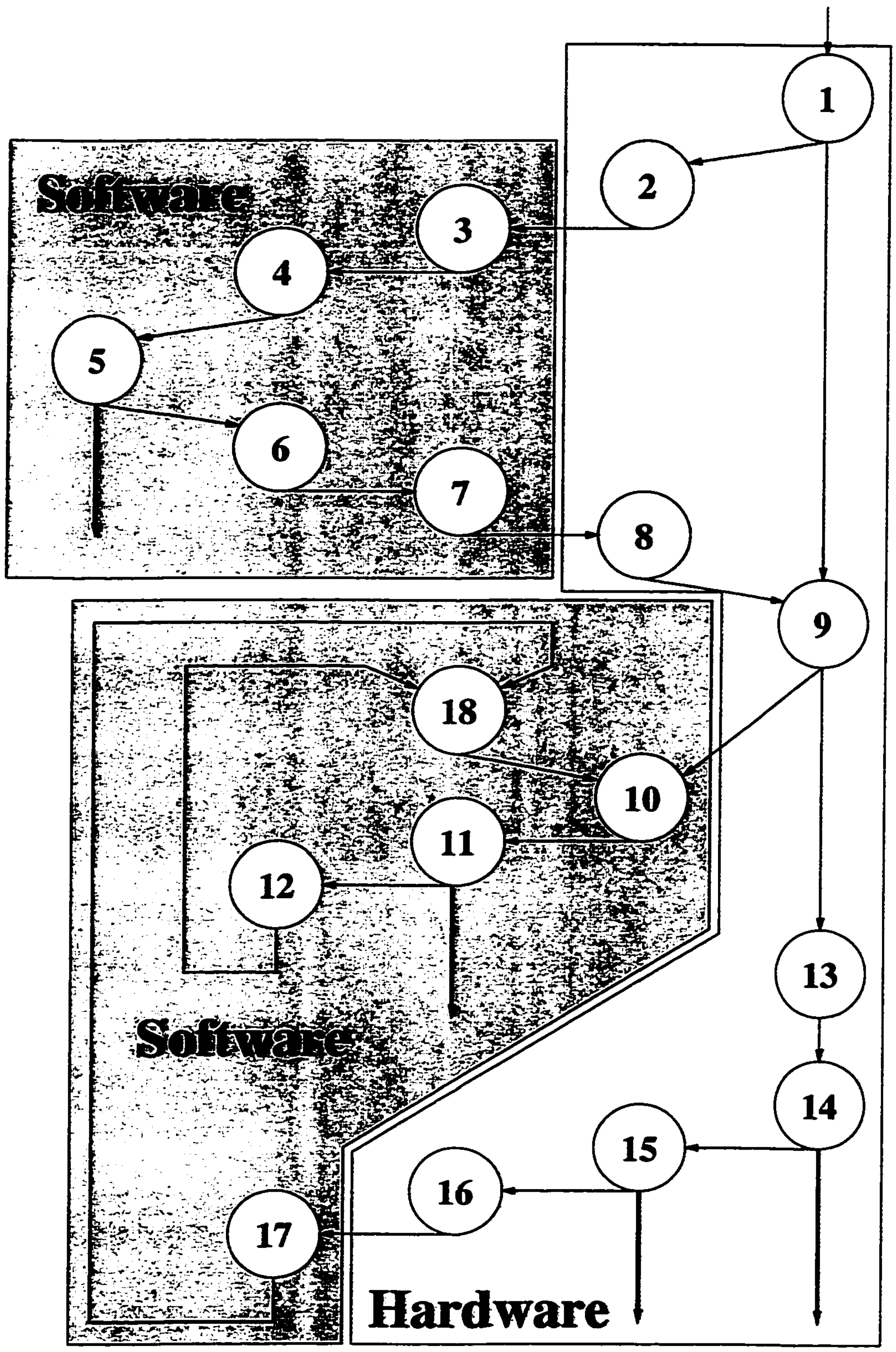

Figure 3-11: Extended elementary operation graph of partitioned GSM example

i	k_i		x_i		j_i	
	#	symbolic	#	symbolic	#	symbolic
1	1	segm	1	hardware	1	hardware segmentation
2	2	filter	1	hardware	2	hardware filter
3	3	schur	2	software	3	software Schur
4	4	param	2	software	4	software parameter
5	5	quant	2	software	5	software quantizer
6	6	decoder	2	software	6	software decoder
7	7	interp	2	software	7	software interpolation
8	8	coeff	1	hardware	8	hardware coefficients
9	2	filter	1	hardware	2	hardware filter
10	4	param	2	software	4	software parameter
11	5	quant	2	software	5	software quantizer
12	6	decoder	2	software	6	software decoder
13	2	filter	1	hardware	2	hardware filter
14	9	grid	1	hardware	9	hardware grid
15	5	quant	1	hardware	10	hardware quantizer
16	5	quant	1	hardware	10	hardware quantizer
17	9	grid	2	software	11	software grid
18	10	adder	2	software	12	software adder

Table 3.4: Extended elementary operation attributes of partitioned GSM example

or protocol descriptions. The context mapping step, when inserting context switch vertices, inserts an appropriate amount of delay to the system. The additional delay should cover the time requirement of the data transfer.

If context-switch circuitry is also subject to optimization, context-switch vertices may be assigned their own operation type, and it should be included in the system cost function. (The GSM example is presented with a single type of context switch vertex.)

8. Start scheduling and allocation.

   Scheduling and allocation depend on the following attributes of elementary operations: execution time t_i and operation type j_i. Scheduling generates the ASAP and ALAP values $(s_i,\ l_i)$, and assigns starting times to operations (v_i). Allocation maps the elementary operations to processors. The output of the scheduling and allocation step is an execution plan of the design, prescribing an estimated schedule of data and control transfers.

3.3.2 Production step

During production runs of MCHLS, after deciding a final partition, heuristics with the highest expected performance (or approximation algorithms with the best approximation ratio) are feasible to use. Since production runs assume a reasonable partition has been found, scheduling and allocation should proceed without further iterations. Because of the low number of executions (exactly one) and the possible cost increase because of inefficiency, scheduling and allocation algorithms should have the highest performance. (Note that choice of the algorithms may depend on application type. The modular nature of the MCHLS design process enables selective replacement of scheduling algorithm, allocation heuristics or both.)

The production step of MCHLS generates the final software and hardware data-flow graphs. The output is passed on directly to the code generator and the RTL synthesizer.

1. Start with the initial DFG shall containing the following properties for each extended elementary operation:

   - Initial operation type, k_i, without any context information (i.e., "FFT", with no note on execution context or complexity.

- Operation complexity, n_i.
- Direct data dependencies, i.e., the set of $(e_i, e_j) : e_i \rightarrow e_j$ pairs. The properties of the data dependencies include the two vertices, and the bit width of the data transfer (usually equal to the operation complexity of the destination operation, n_j).

2. Use the partition found by the last iteration of the previous stage. This is assumed to generate an efficient graph partition, as estimated by the fast (inaccurate) scheduling and allocation attempts.

3. Assign final operation types to elementary operations. Since execution contexts are fixed by partitioning, the final execution type ("FFT in hardware") may be assigned to elementary operations:

$$j_i = j_i(k_i, x_i)$$

In some systems, where processors are specialized for different data sizes, the final operation type may depend on operation complexity as well:

$$j_i = j_i(k_i, x_i, n_i)$$

4. Assign execution times to elementary operations. Execution times depend on operation complexity and type:

$$t_i = t_i(j_i, n_i)$$

5. Insert transfer vertices to the system, where applicable. The new vertices behave as transfer elementary operations, and do not count as functional operations. The new vertices' only practical attribute is their execution time, which is a function of the context switch source and destination, and the operation complexity of the data destination:

$$t_k = t_k(x_i, x_j, n_j)$$

Reaching this point, software and hardware are effectively separated in the CDFG. The last step changes the topology of the elementary operation graph. (Administering

i	k_i	k_i (symbolic)	Name in GSM block diagram
1	1	segm	Segmentation
2	2	filter	Auto-correlation
3	3	schur	Schur-recursion
4	4	param	LAR calculation
5	5	quant	Quantizer/coder
6	6	decoder	LAR decoder
7	7	interp	Interpolation
8	8	coeff	Reflection coefficients
9	2	filter	Inverse filter
10	4	param	LTP parameter
11	5	quant	Quantizer/coder
12	6	decoder	LTP parameter decoder
13	2	filter	Weighting filter
14	9	grid	RPE grid selection
15	5	quant	APCM quantizer
16	5	quant	Inverse quantizer
17	9	grid	RPE grid positioning
18	10	adder	Adder
19	5	cswitch	Context switch
20	5	cswitch	Context switch
21	9	cswitch	Context switch
22	10	cswitch	Context switch

Table 3.5: Extended elementary operation attributes of partitioned GSM example after context mapping

the changes is straightforward, since only unary operations are inserted.)

6. Start scheduling and allocation.

Scheduling and allocation depend on the following attributes of elementary operations: execution time t_i and operation type j_i. Scheduling generates the ASAP and ALAP values (s_i, l_i), and assigns starting times to operations (v_i). Allocation maps the elementary operations to processors. The output of the scheduling and allocation step is an execution plan of the design, prescribing an estimated schedule of data and control transfers.

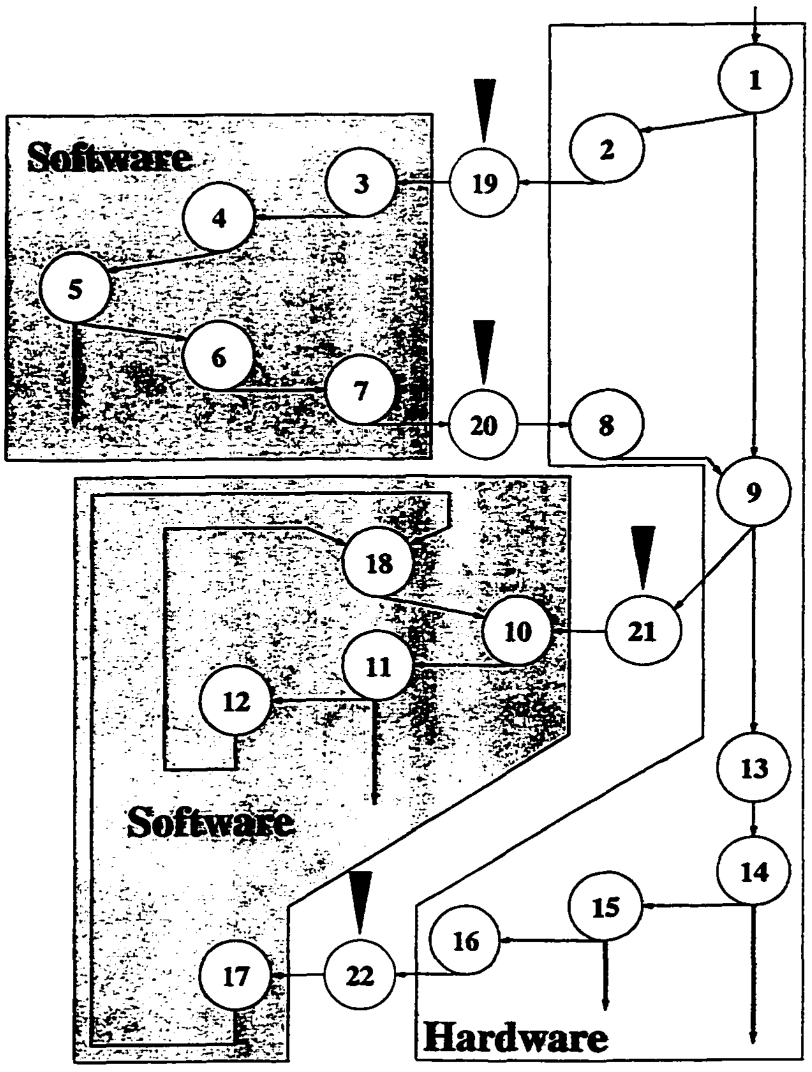

Figure 3-12: Transfer vertices in partitioned GSM example

Chapter 4

Implementation

This chapter is dedicated to implementation of the hardware-software codesign environment in the *PIPE* framework (*HSPIPE*). The chapter contains a description of *PIPE* internals to document the design process inside *PIPE* and to contrast the original *PIPE* design process with hardware-software codesign extensions (*HSPIPE*). The chapter describes the front end, the data transfers and internal representation, and the output formats of the *HSPIPE* system. The example representation requires user intervention for efficient usage, and therefore is not ready for industrial usage. (Most shortcomings are addressed, in Chapter 7.)

Since the goal of the example implementation is a technology demonstration rather than optimal behavior, the *HSPIPE* environment is a set of connected programs driven by scripts. The language of choice in the example implementation is *Perl* [WCS96]. As well as a very efficient high-level prototyping language, Perl programs (interpreted scripts) may be executed on a wide range of operating systems. Since *HSPIPE* components are available with full source code, students may study them in detail, and implement changes with very little effort. (The relative flexibility of the language also has great potential for enforcing intellectual property protection, since Perl source code is usually difficult to understand. Restriction of information was not a design goal in *HSPIPE*. The source code of *HSPIPE* modules is actually easy to understand by Perl standards.)

As illustrated in Figure 4-2, the functionality of the multiple-context environment design environment has been implemented as a module before actual *PIPE* execution. The multiple-context environment extensions generate an efficient partition of the input CDFG, construct a "single-context" description of it, and pass it to *PIPE*. As described in Chapter 3, multiple execution contexts are separated by assigning different operation types ($j_i \neq j_l$)

to elementary operations of the same operation category ($k_i = k_l$) if they are executed in different execution contexts ($x_i \neq x_l$). Such a transformation effectively separates execution contexts for purposes of scheduling and allocation, and single-context CAD tools (such as PIPE) may be used for scheduling and allocating such designs.

The system description of *HSPIPE* is taken as an input CDFG. (As discussed earlier, the CDFG is assumed to be fixed and immutable.) An initial partition is generated by the `init(1)` module. In the example implementation, `init(1)` iteratively transforms the critical paths of a purely software implementation to hardware, stopping if latency gets close to system constraints. The output of `init(1)` is a multiple-context CDFG (MCCDFG).

The partitioning process is implemented in a separate module, which in turn uses external components to evaluate the quality of the partition. The partitioning module, `klfm`, is executing iteratively until system performance is considered to be satisfactory. (In the example implementation, `klfm` actually uses the Kernighan-Lin-Fiduccia-Mattheyses algorithm.) Partitions are evaluated by a chain of external modules:

1. Context mapping (`cmap`). This module expands extended elementary operations to elementary operations, and generates j_i for elementary operations. Context-switch vertices are also generated in this step, modeling the time required to pass data between execution contexts.

   The context mapping step uses an external *technology library* to map operation categories and execution contexts to final operation types. No interface code is generated at this point, but the size of the interface is approximated. The external technology library contains a description of elementary operation timings for every possible execution context, as well as timing functions for context switches.

2. Scheduling and allocation (`sched`). In the example implementation, the scheduling step uses a special list scheduler with built-in allocation. The output of the scheduling step is a scheduled CDFG (SCDFG), where elementary operations are fixed to their start times (v_i).

   The scheduler is a resource-constrained list scheduler, with the corresponding allocator. Since the schedule may not violate resource constraints, the allocator is easy to implement.

3. Performance evaluation (`scheval`). The schedule generated by the list scheduler is checked for timing violations. (The list scheduler guarantees that there are no resource violations.)

   In the example *HSPIPE* implementation, this step returns the ratio of achieved latency over optimal latency.

If the performance of the current partition is assumed to be satisfactory, the partition is fixed, otherwise the partition module (`klfm`) enters another iteration.

Once the partition is fixed, the original MCCDFG is passed through context-mapping (`cmap`) and interface generation (`ifconfig`). The latter inserts the necessary code (in software) and control logic (in hardware) to interface to other execution contexts. Even if the context mapping step has been performed for the same partition earlier, saving the generated context information would be overkill. (Especially since the context mapping step is a very fast operation. It is practically static text replacement and functionality of the C "`#include`" mechanism.)

After the final context mapping step, the *HSPIPE* CDFG is passed through a filter (`cdfg2pipe`) to produce a *PIPE* input file. This step is necessary since *PIPE* uses attributes which are not used during the partitioning phase. The output file may be passed directly to *PIPE* for processing.

The implemented system is incapable of designing at a functional level, and designers must transform abstract problem descriptions to system control-data-flow graphs. The current implementation starts synthesis from the generated control-data-flow graph and finishes at a register-transfer level (Figure 4-1). The *HSPIPE* design environment may theoretically be extended to partition designs at a higher level, before the control-data-flow graph is generated (similar to [VNG95], where system descriptions are analyzed and simulated in VHDL before synthesis starts). Changing the level of system partitioning may be done as an additional step (translating a very high level algorithmic description to control-data-flow graph) as shown in Figure 7-1. Since such an extension would require extensive analysis of a very large problem set to select heuristics, it may not be performed as of May 11, 1999. After accumulating the necessary experience (several man-years worth of optimization and evaluation with the *HSPIPE* environment), the input step may be

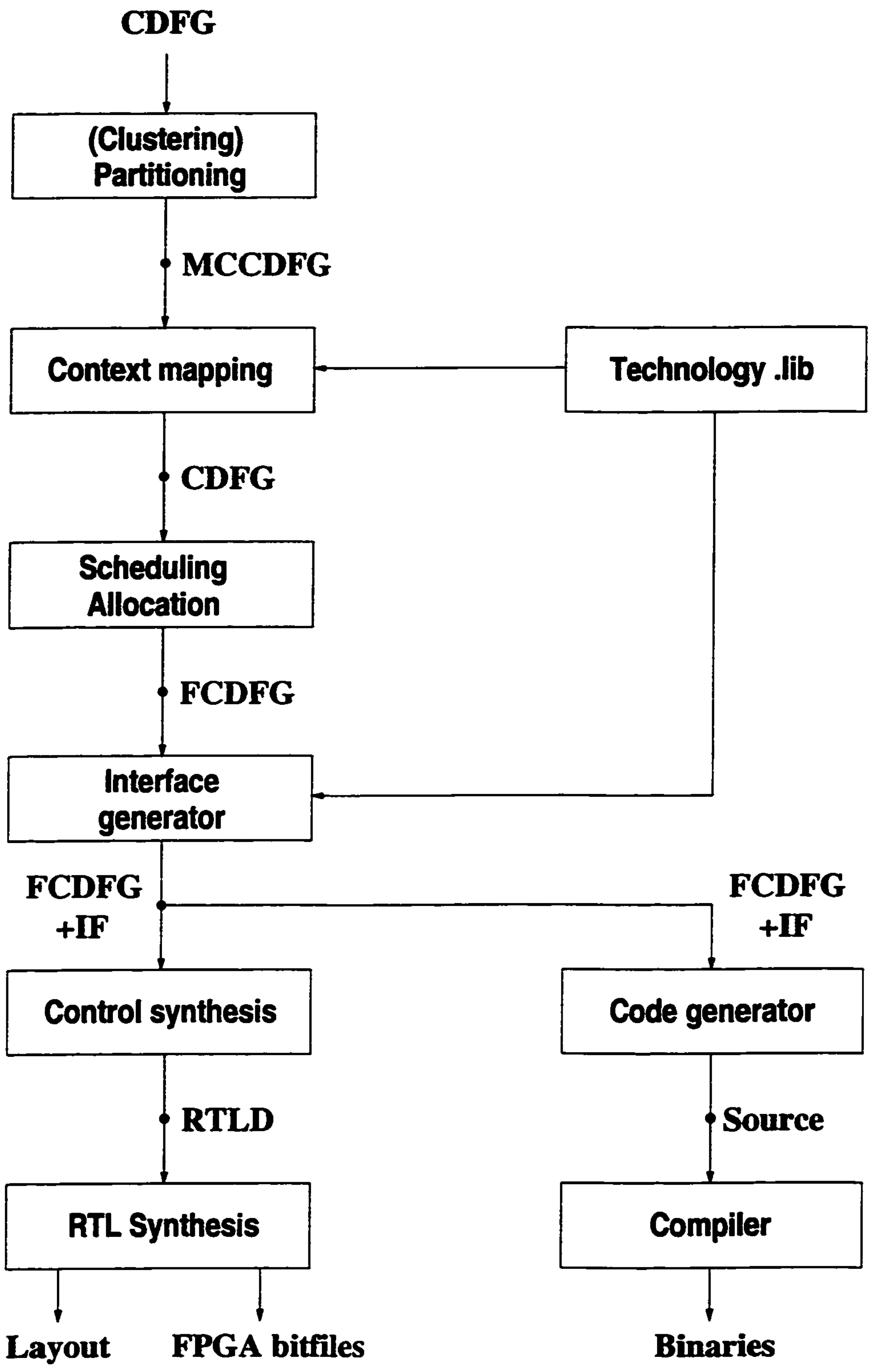

Figure 4-1: HSPIPE design process without functional partitioning

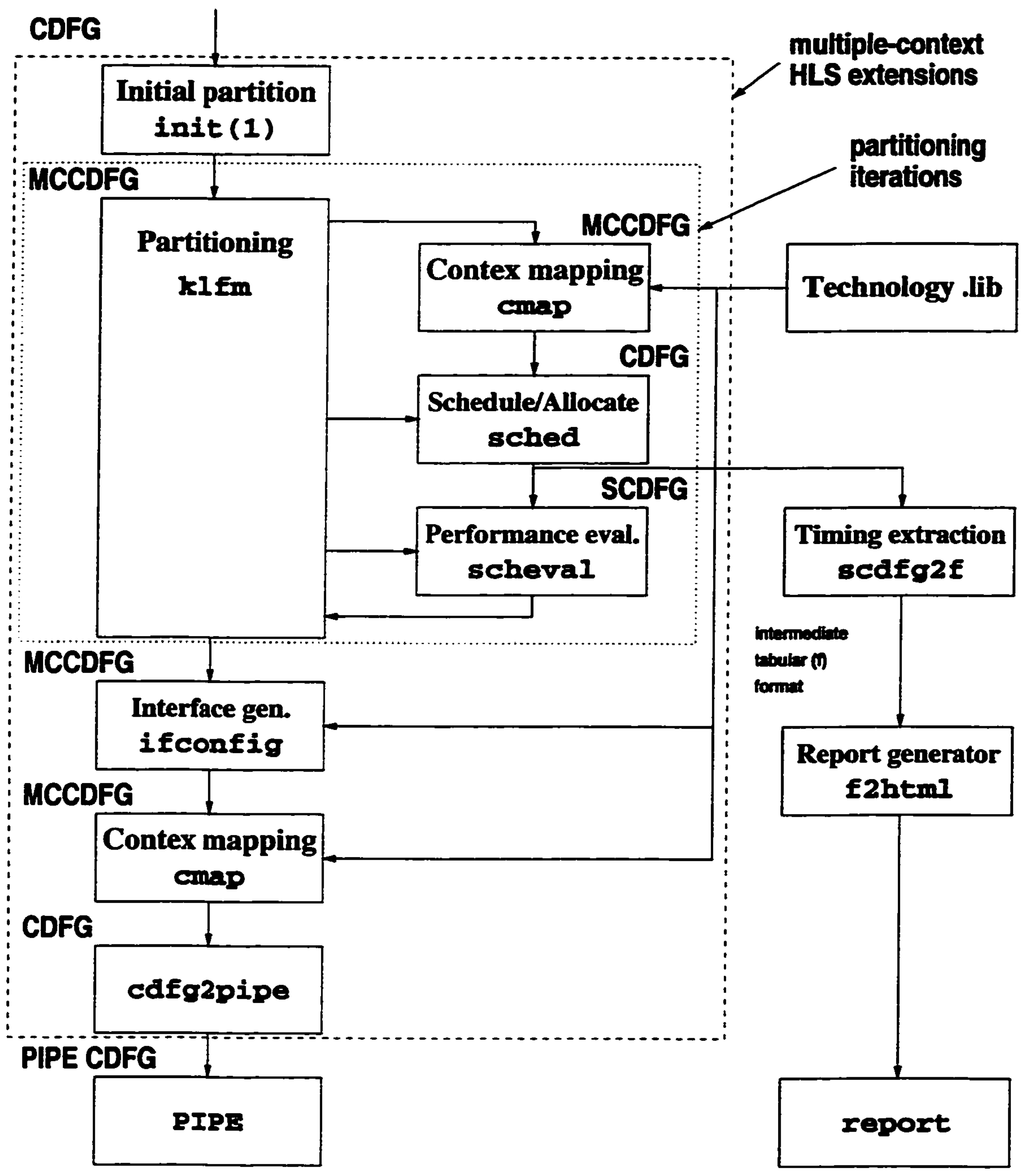

Figure 4-2: Components and data representation in the HSPIPE framework

introduced to the design flow. For this reason, functional partitioning requires further investigations at BME (Technical University of Budapest, Hungary). (Note that the design environment of the University of California at Riverside is fundamentally different from the one at BME (Technical University of Budapest, Hungary), and therefore the heuristics used by UCR researchers are not applicable to our target environment.)

4.1 User interface

The user interface of *HSPIPE*, similarly to *PIPE*, is Spartan (see Appendix B). Relying on user-supplied text files, and text-based technology libraries, the *HSPIPE* environment transforms these text descriptions to text input for a RTL synthesizer.

Extending the *PIPE* environment (and, similarly, *HSPIPE*) with a convenient *graphical user interface (GUI)* is definitely possible as a future development (Section 7.1). Certain sections of the *HSPIPE* extensions have been successfully expanded with GUIs (implemented in Tcl/Tk over the Perl scripts). Since the most important function of the current, example *HSPIPE* implementation is to provide a framework for algorithm experiments, implementing a GUI for the components is not an urgent task.

Since the current *HSPIPE* implementation is practically an experimental setup, storing internal information in text files has additional advantages over more integrated or graphical implementations. Having intermediate results in human-readable format reduces debugging and verification time considerably. Being able to edit input information by any tool (such as `vi`, which was used extensively as a development and testing tool) was extremely useful during implementation (in situations such as debugging and fault injection).

The results of the performance evaluation have been processed in an intermediate format, which in turn was translated into HTML.

4.2 Front end

During the analysis of standard benchmarking problems, some of the input CDFG have been generated by auxiliary programs. As an example, the FIR filter structures were generated by the **genfir** script, which provides CDFGs for FIR filters of arbitrary sizes. (Covering FIR filters actually covers a large number of topologically similar applications [Kun82].) In practical systems, inputs CDFGs may be generated from high-level language descriptions, schematic capture tools, or even functional descriptions. (An example system is presented in Appendix A.)

A possible future development task is interfacing functional partitioning to the *HSPIPE* environment. By treating the system CDFG as a design parameter, and encapsulating the *HSPIPE* environment in an additional round of iteration, designers could experiment with different realizations of the same high-level description.

4.3 Code generator

As an experimental extension, a basic code generator has been appended to the *PIPE* environment. Such a straightforward code generation process has been shown to be effective [ET98], assuming that compilation is efficient. (The *HSPIPE* environment relies on the efficiency of the underlying compiler, since it generates source code only.)

Context-switch vertices are assumed to be self-contained units provided by the user. On the software side, they are responsible for generating the code to interface to hardware on context-switch interfaces. The software component of the context-switch functionality is a set of I/O operations performing a handshake synchronization with the hardware execution context.

Depending on the hardware environment, the context-switch extended elementary operations either wait asynchronously for hardware signals (in a hardware-driven environment), or trigger hardware events (in a software-dominated environment). Most practical systems feature a combination of both.

The software sections of the GSM voice coding example have the extended elementary operations shown on Figure 4-3. Assuming the extended elementary operations are available as functions, the highest level of software functionality may be described as a straightforward replacement of functions. Note that the generated code is inefficient, since the straightforward code generation introduces false data dependencies by using a very small set of temporary variables.

```
while (1) {
        tmp1 = cswitch_19();
        tmp2 = schur_3(tmp1);
        tmp1 = param_4(tmp2);
        tmp2 = quant_5(tmp1);

        /* System output */
        sys_store(QUANT_5, tmp2);

        tmp1 = decoder_6(tmp2);
        tmp2 = interp_7(tmp1);
        csswitch_20(tmp2);
}
```

After expanding the extended elementary operations to elementary operations, the internals of the extended elementary operations are described as truly elementary operations. For software purposes, the generated interface code features extremely low level I/O operations.

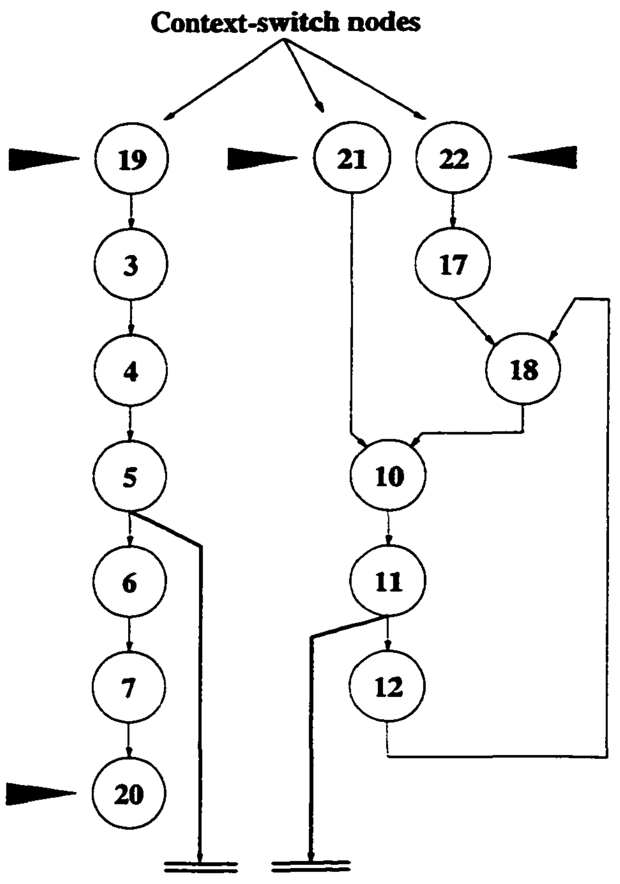

Figure 4-3: Software sections of the GSM example

Chapter 5

Analysis

The results of tests and performance analysis are presented briefly in this chapter. The chapter contains well-known benchmarks from the high-level synthesis community. The example problems were synthesized as multiple-context environment applications. A digital FIR filter application is used to demonstrate the results on classical benchmarks. The performance of the *HSPIPE* algorithm has been evaluated on a cryptographic application, RC-5 encryption.

The chapter contains no code or extended data-flow graph for verbosity constraints. The results of the chapter illustrate that a combination of the selected heuristics may achieve efficient results in reasonable runtime. During testing, a personal workstation obtained the requested results under an hour in each case, even in systems with a vertex count of up to 4096.

An important observation is that the partitioning process converges in practically all initial partitions obtained by refinement of extreme (purely software or hardware) configurations. Since the Kernighan-Lin-Fiduccia-Mattheyses algorithm terminates even in local minima, there are no convergence problems. The partitioning phase therefore may be run with perturbations of the initial partition. The solution with the lowest cost is then implemented.

5.1 Filters

A classical benchmark category of hardware synthesis is digital filters, both finite and infinite impulse response. Neglecting the effect of feedback edges, the topology of both filter types is very similar (a FIR is shown in Figure 3-10). Indeed, describing the topology of FIR filters covers a large subset of practical HLS target systems [Kun82].

The test runs of the *HSPIPE* implementation contained FIR filter CDFGs between 64 and 2048 inputs. A complete 2048-point FIR system, with the necessary support functionality, contains over 4000 vertices. Such a system is certainly large enough for testing heuristics. Exploiting the regularity of FIR applications, the FIR filter flow-graphs were generated in different sizes. A custom module was developed for this purpose (`genfir`).

A problem with the extremely regular (generated) filter structures (and similar topologies) is high symmetry. Because of the regular structure, the partitioning process initially considers identical changes to all possible improvements. Since the results of identical changes is practically uniform on all paths, the partition improvement algorithm had a large set of "best" moves to select from, and tie-breaking was necessary. The efficiency of the initial steps is therefore determined by the performance of the tie-breaking heuristic. (Most of the heuristics used by list scheduling are not applicable, since FIR data-flow graph paths are identical from multiplier vertices to the sums, and offer no difference in mobility.) Once the graph has been removed from its completely regular status, the improvement stage of partitioning proceeds without problems.

Note that regularity is a problem in synthesized graphs during generation of the initial partition as well. Similarly to reasons discussed before, there is a multitude of critical paths in the system, and tie-breaking might be necessary in selecting the path to be folded, especially during the first steps of generation.

The Kernighan-Lin-Fiduccia-Mattheyses algorithm under the boundary conditions used during testing converges typically to states where large subsections (practically individual FIR filters) are mapped to the same context, as shown in Figure 5-1. Note that similar results are obtained practically independent of the initial partition. Exploring the convergence process, obviously such a set of context switches maximizes possible parallelism inside the FIR submodules, and delays context switches as late as possible under the resource constraints. Since FIR structures have a strictly decreasing "horizontal cut size" (i.e., the number of concurrently active data transfers), delaying context switches as late as possible minimizes the number of transfer nodes.

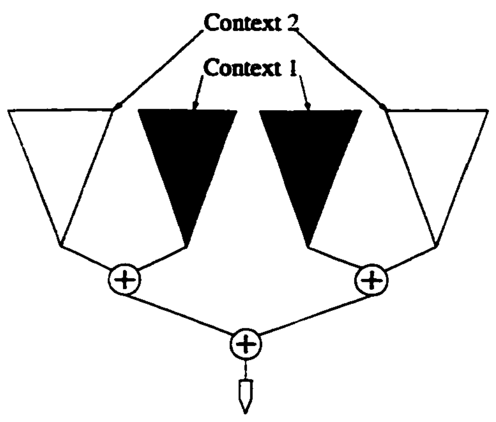

Figure 5-1: Typical final partitions of FIR filters

The results of hardware-software codesign are demonstrated on the cost-latency functions of a 512-point FIR filter. Since cost comparison between software and hardware systems is difficult, a straightforward method is chosen. Hardware cost is approximated by the silicon cost, which is proportional to the size of the required silicon. Software cost is proportional to the number of simultaneously executing elementary operations, and operation costs are identical for all instruction types.

The first model (Figure 5-2) shows the Pareto-points for ideal hardware and software systems. In these ideal systems, an infinitely large number of units may operate in parallel. (This is definitely false for practical software systems, and non-ideal software is shown later.) In addition to the extreme implementations, Figure 5-2 contains the cost function for the first multiple-context solution found by MCHLS. Obviously, the cost of a multiple-context implementation is always lower than that of a purely hardware system under the same time constraints, and may not be lower than the software solution.

For large latencies (over $L = 2000$), the solutions obtained by MCHLS are closer to software than hardware solutions. In these cases, most of the data-flow graph is mapped into software, and the number of context switches is small. (In some instances, context switches were completely eliminated.)

As the latency constraints are decreased (under $L = 1500$), more and more of the data-flow graph is mapped into hardware and the cost function of the multiple-context solution

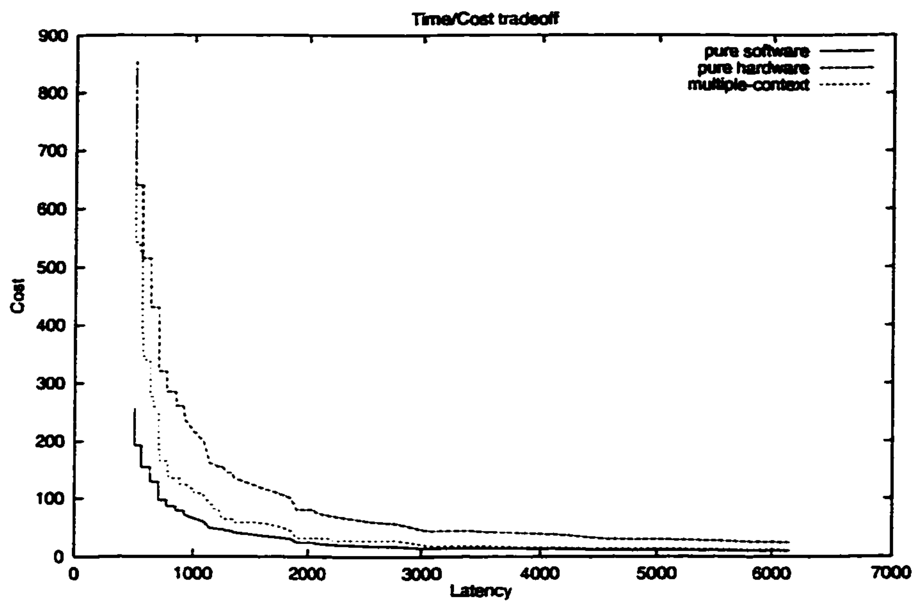

Figure 5-2: Hardware and software implementation costs assuming infinite software parallelism

gets closer to the purely hardware implementation. Because non-critical sections of the system are mapped to software, the system cost stays under the hardware system cost, demonstrating the usefulness of a multiple-context design.

In practical systems, software may not be parallelized beyond a certain limit. Since multiprocessor designs present problems beyond the scope of this dissertation, software parallelism is better modeled if it does not extend beyond the maximum achievable inside a single processor. As an example, consider a PowerPC microprocessor where two adders and two multiplies may be done in parallel. Assuming a maximum latency of $L = 6150$ and a corresponding software cost of 10 (no parallelism, at most one multiplier and one adder is utilized simultaneously), the theoretical minimum achievable by software solutions may not be less than $L = 3075 = 6150/2$. In fact, because of the structure, the actual minimum is $L = 3150$. No software solution may achieve $L < 3150$ (Figure 5-4).

Since the parallelism of software is limited, the cost difference between hardware and multiple-context systems is lower under these conditions (compare Figure 5-5 to Figure 5-3). The maximum cost savings was 40 units (absolute) (Figure 5-5), which is equivalent to a relative maximum of 42 %. The average cost difference between the hardware and

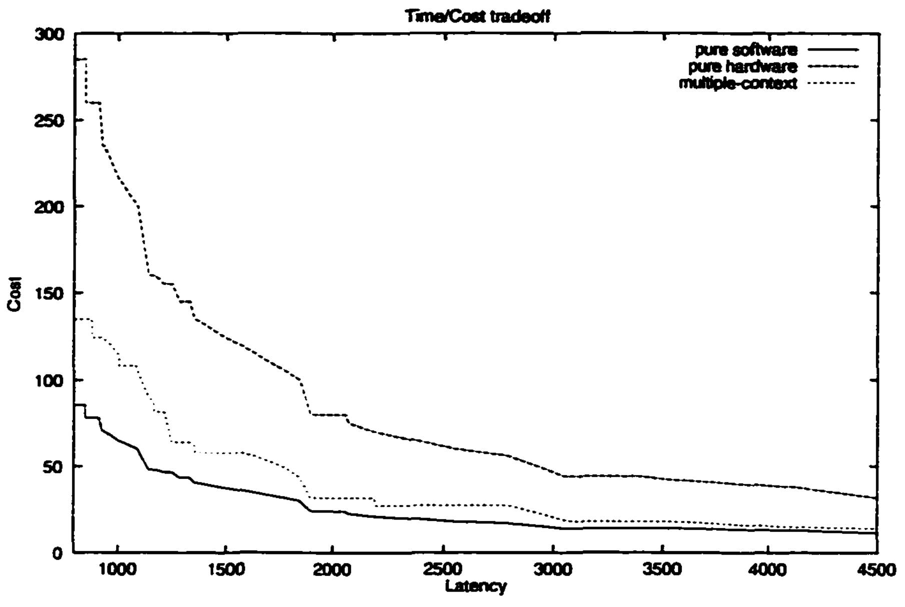

Figure 5-3: Transition between mainly hardware and mainly software solution

multiple-context solution was approximately 20 %.

5.2 RC-5 encryption

The RC-5 encryption algorithm is a typical application which may be implemented in a mixed hardware-software environment [Sch95, Sta98]. The algorithm is highly customizable, changing the number of iterations and similar parameters, yet the internals (the main loop core instructions) are based on a number of primitive operations. The basic operations are practically exclusive OR (XOR), rotation, and addition. An RC-5 encryption process is characterized by the following parameters:

Number of rounds, r,

Word size, w, measured in *bits*. The algorithm encrypts *two* words during each iteration.

Key width, b, measured in *bytes*.

Key value, K, a b-byte value used as an initial value for setting up a look-up-table before actual encryption

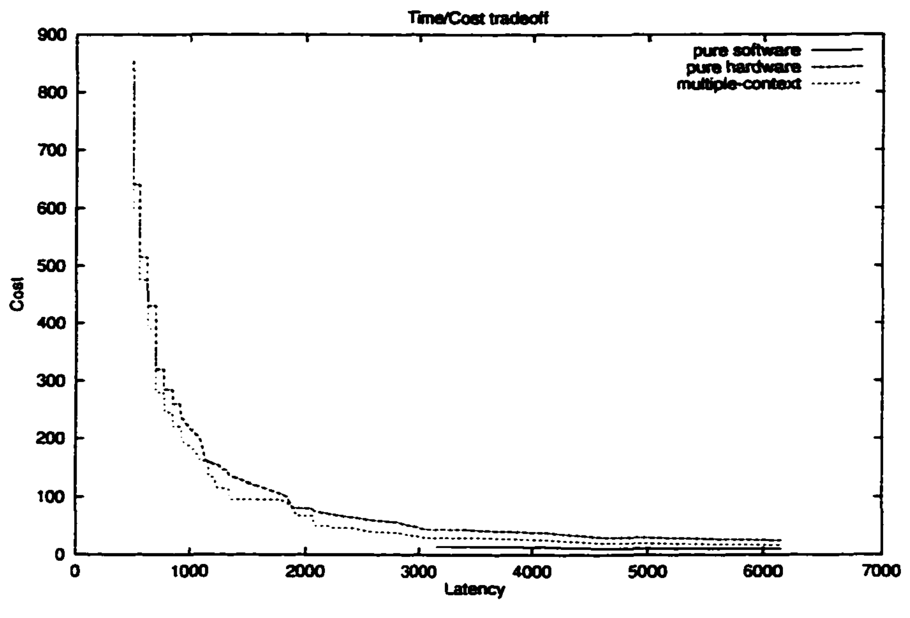

Figure 5-4: Hardware and software implementation costs assuming finite software parallelism

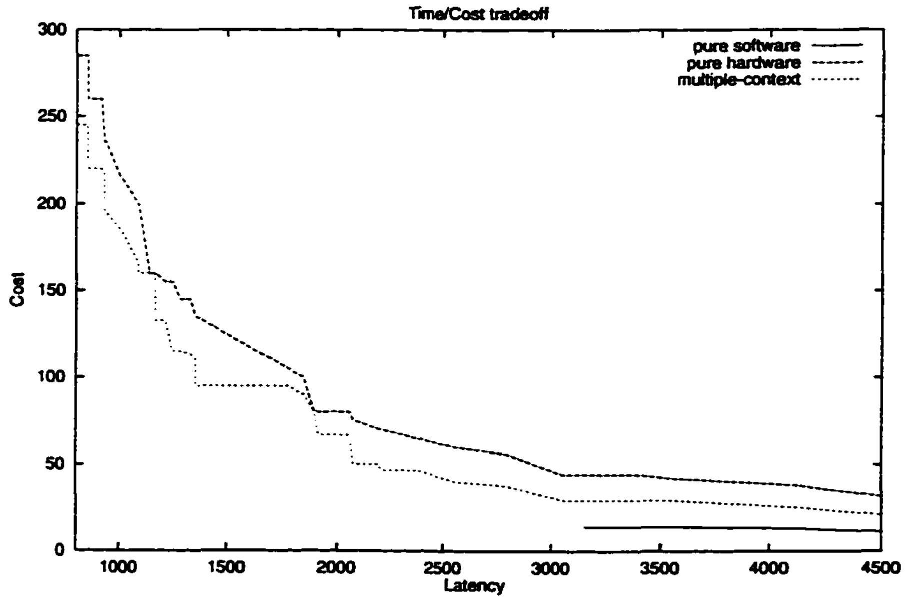

Figure 5-5: Transition between non-ideal software and hardware solution

Using the above notations, the algorithm may be described with the following pseudo-code:

```
A = A + S[0];
B = B + S[1];
for (i=1; i<=r; i++) {
        A = ((A ^ B) << B) + S[2*i];
        B = ((B ^ A) << A) + S[2*i+1];
}
```

where the XOR operation is ^ and << is a circular rotation instead of the usual C "shift". The S look-up-table is initialized as a function of the desired cryptographic parameters.

The RC-5 algorithm was developed to maximize the time required for a brute-force cryptographic attack. Timing and dependence analysis of the algorithm reveals that there is no chance for overlapping in the DFG, and so no pipelined execution is possible. Also the algorithm may be extended for very large memory sizes, which means software implementations may run out of reasonable cache ranges with the correct combination of parameters [GW96]. (Scalability of the algorithm was a primary design goal.) Since the algorithm efficiency is measured as the time required to check all possible keys by brute force, a feasible measure of any RC-5 implementation is the latency of the main loop (Figure 5-6).

The following limitations slow down both hardware and software solutions:

1. Two memory accesses are necessary in every iteration.

   This is a performance bottleneck in both hardware and software, since memory access times are significantly higher than cycle times of state-of-the-art off-the-shelf processors or custom hardware. Because of frequent memory accesses, parallelization requires local memory for all processors.

2. Even if data operations are performed on two target registers (A and B), the main loop may not be operated in an overlapping (pipelined) fashion, since data dependencies are strictly linear (Figure 5-7). This limitation applies to both hardware and software implementations.

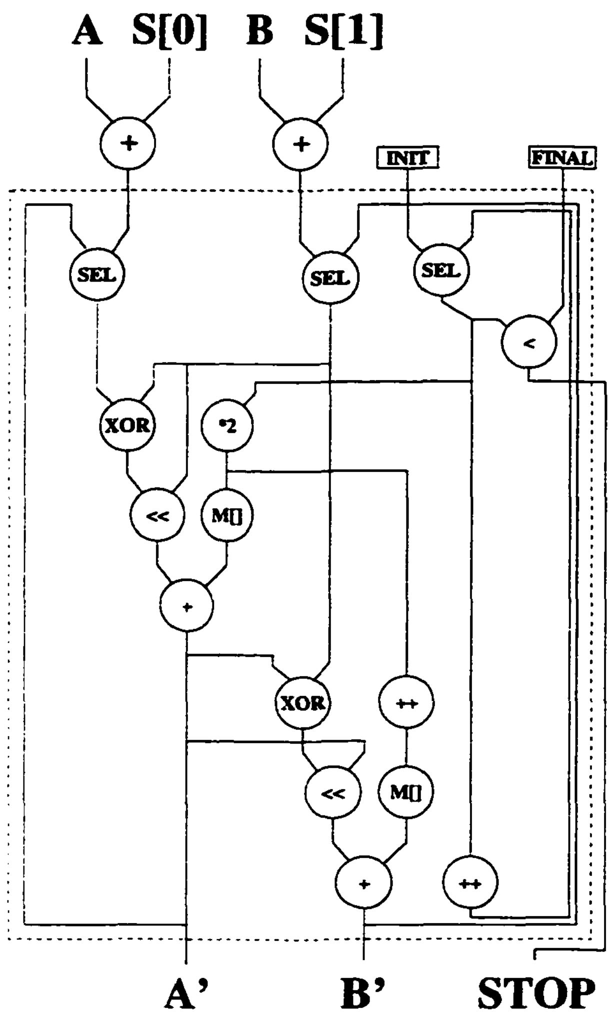

Figure 5-6: Main loop of the RC-5 encryption algorithm

3. Two data-dependent rotations must be implemented serially since the number of rotations is too high for a faster, combinatorial implementation.

Test runs of the partitioning process also demonstrated that mixed hardware-software implementations of the RC-5 algorithm are slower than both pure software and pure hardware solutions. Because of the direct data dependencies in the algorithm main loop, context switches inside the system increase the overall length of the critical path, restricting the implementation to single-context systems. There must be an even number of context switches on the highlighted data path in Figure 5-7. Figure 5-8 shows the latency of the loop for every valid combination of context switches on the loop. (The software timing model used an Intel 8051-class microprocessor. The loop becomes slower if a RISC CPU is used.) There leftmost configuration in Figure 5-8 ($L = 46$) represents pure software, the rightmost point ($L = 36$) is pure hardware. Even if some of the multiple-context implementations approach the performance of the purely software solution, only two get close (at $L = 48$). Obviously, hardware-software codesign is an inefficient tool to speed up brute-force decryption of RC-5.

As a related result, an ongoing distributed brute-force key search effort launched by *Distributed.net Technologies, Inc.*[1] has covered 7.913 % of the keyspace in 559 days. This number is remarkable since this practically infinitely parallelizable problem is investigated by distributed.net by using the idle cycles of 45541 computers simultaneously. (Data accurate as of 5 May, 1999.)

5.3 Convergence

Since the *HSPIPE* environment extends scheduling/allocation and partitioning heuristics in a modular way, the convergence of multiple-context development is determined entirely by the convergence of the underlying algorithms. In the sample implementation, the greedy Kernighan-Lin-Fiduccia-Mattheyses algorithm always terminates, but it may converge to a local minimum. Since the runtime of the Kernighan-Lin-Fiduccia-Mattheyses algorithm is predictable, multiple designs may be evaluated in polynomial time. By selecting the result with the best performance, even local minima may be avoided.

[1]`http://www.distributed.net`

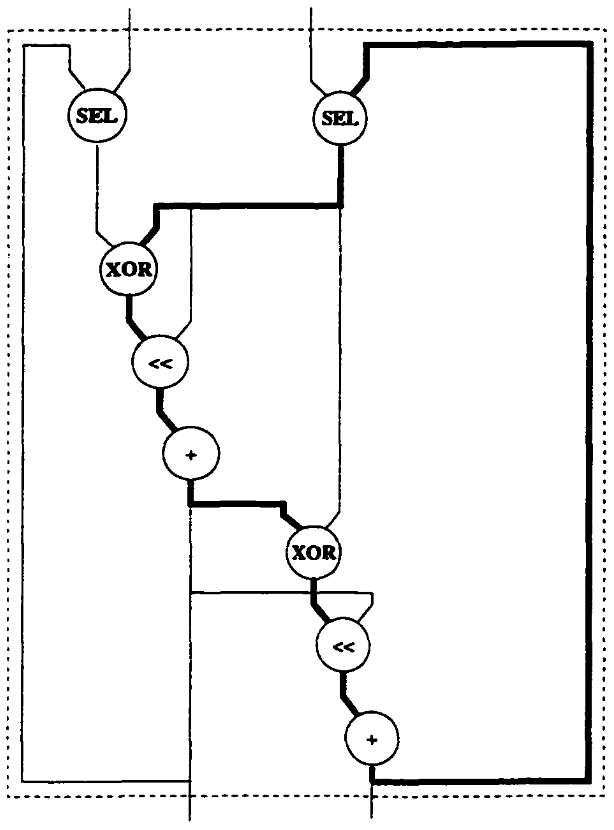

Figure 5-7: Main loop of RC5

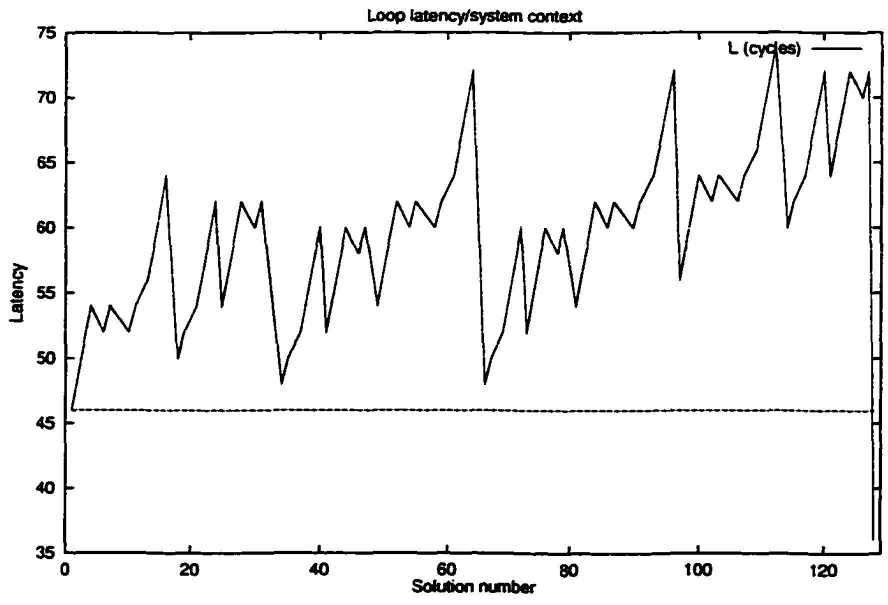

Figure 5-8: Latency of RC5 main loop as a function of configuration

5.4 Performance

The efficiency of the MCHLS process depends entirely on the performance of the production scheduling step.

In the FIR benchmark, the list scheduler in the iterations has produced results with an implementation cost of at most 20 % over the optimal solution under the same time constraints. (FIR filters are among the HLS benchmarks for which the optimal solution is known.) By repeating the partitioning iterations with small changes to the initial partition, the best values got within 10 % of the optimal solution.

Note that these performance values belong to the scheduling heuristics (in this case, a resource-constrained list scheduler). By changing the algorithms, different results may be obtained for the same benchmarks. The contribution of the MCHLS framework is simply preserving the quality of the schedules.

Chapter 6

Summary

The dissertation presents a model of a multiple-context environment as an extension of generally accepted system descriptions used in high-level synthesis. The multiple-context enhancements are upwardly compatible with the notations of high-level synthesis literature. The multiple-context extensions hide the details of context information from the underlying scheduler and allocator heuristics. Since multiple-context flow-graphs are transformed to a description without contexts, existing software or hardware design heuristics may be used in MCHLS.

The dissertation demonstrates that the above transformation preserves the necessary information to properly simulate and optimize multiple-context designs in existing single-context tools, while retaining properties unique to multiple-context environments.

The convergence of the MCHLS process is determined by the convergence of the partitioning heuristic. The designer has complete control over the design process, and may implement any partitioning heuristic. Similarly, the performance of the MCHLS design is set by the efficiency of the scheduler. As heuristics are standalone blocks in the MCHLS framework, any algorithm may be upgraded without influencing the properties of the other.

Chapter 7

Future Development

7.1 Integration with *Visual PIPE*

Since the *HSPIPE* environment is not intended for immediate industrial application, the lack of a graphical user interface is currently not a serious limitation. Should the *HSPIPE* CAD environment be extended, a GUI over the underlying text-based CAD tool would be a useful extension. Since creation of such a GUI is a completely unrelated project, it has been delegated to future developments. (Relying entirely on text files had the advantage of reduced debugging and verification time, as well as the capability to use self-modifying tools during development.)

In a traditional UNIX style, the GUI of *HSPIPE* may be simply a graphical shell providing the necessary configuration files and command-line switches to the underlying *HSPIPE* modules. Under UNIX, a wide selection of such GUI scripting is available, and creating a thin shell over the existing modules would not be difficult (assuming the interfaces do not change).

7.2 Algorithmic improvements

The dissertation investigations assumed a complete, fixed system description specified as a CDFG. One way to extend the capabilities of the *HSPIPE* environment is to provide greater flexibility by implementing functional partitioning before the currently existing *HSPIPE* steps.

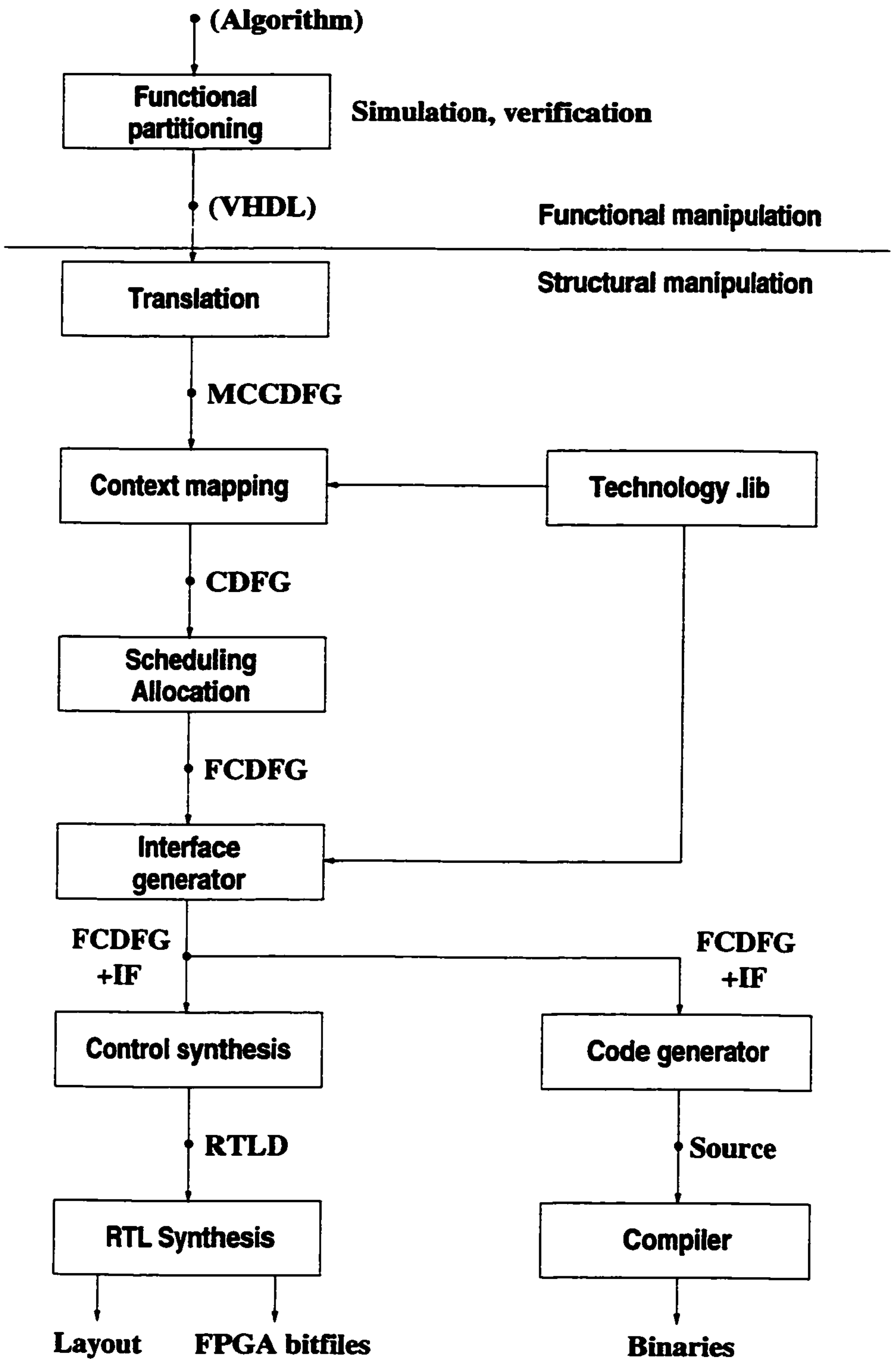

Figure 7-1: HSPIPE design process with functional partitioning

7.3 Extending cost matrices to non-binary partitioning

Since the heuristics used in partitioning and clustering are not limited to binary partitioning, the MCHLS design environment described in Chapter 3 may be used to generate non-binary multiple-context environment systems. Possible systems with more than two execution contexts may be multiprocessor systems (multiple software contexts), systems with dedicated hardware coprocessors (multiple hardware) or systems with remote connectivity, such as serial links (multiple hardware or software).

In non-binary systems, the transfer cost matrix (i.e., data transfer times, $t(s,d,n)$) is $N - by - N$, where N is the number of execution contexts.

7.4 Customized implementations of elementary operations

Using an underlying module library (both hardware and software), it is possible to implement the same functionality in several ways. Selecting a different implementation for different elementary operations depending on complexity is a generally accepted method in system-level synthesis (Section 3.2, p. 88). Instantiating a different version of the same module for different complexities is a popular optimization technique in embedded systems, where minimizing resource consumption is of extraordinary importance.

A similar technique may be applied in a slightly different way for operations of the same size. Under certain circumstances, operations with the same complexity could be implemented in different realizations, if important timing characteristics are not changed by changing realizations. Since the traditional trade-off is between space (cost) and time, some operations may be replaced by slower, cheaper implementations, if such a change does not alter the global performance measures of the system.

As an example, multiplication of a given size may be implemented with a parallel multiplier or a serial one, which have extremely different characteristics. A straightforward parallel multiplier operating on n bit operands requires $O(n^2)$ silicon area and practically $O(1)$ time, while a serial multiplier multiplies in $O(n)$ time and occupies $O(n)$ area in silicon. If scheduling prescribes a suitable start time for a multiplier, a parallel implementation may be replaced by a serial one, decreasing silicon requirements considerably.

An example system is presented in Figure 7-2. Two multiplications are scheduled as shown, while the original ASAP and ALAP times are the time frames illustrated. The multipliers are implemented in parallel, with an execution time of $t = 2$ cycles. Since their execution times overlap (in cycle t_0), allocation may not assign the two multiply operations to the same multiplier, and at least two multiplier units are needed in any implementation.

There might still be a possibility of reducing silicon requirements. Let's assume that direct successors do not start immediately after the start time of e_i. For example, if all of e_i's direct successors are started in their ALAP cycles, the output of e_i may does not get read in the time cycles shaded black, since its direct successors start processing no sooner than cycle $l_i + 2$. In this case, the parallel multiplier assigned to e_i may be replaced with a serial one (the alternate timing is shown in the rightmost column). A parallel multiplier of given complexity c_i executes in $t_i = 2$ cycles, while a serial implementation requires 8 cycles to finish. The serial multiplier, started in the same time cycle, still finishes in time under the given time constraints. The difference between parallel and serial implementations is the amount of time between stabilizing the output and actual usage (shown with time cycles shaded with black). The serial implementation is obviously still fast enough, but offers potential savings in silicon.

Changing the final implementation method for elementary operations may not occur before final starting times are set, but must be known before hardware is actually synthesized. These restrictions imply that technology assessment must occur after the allocation phase, and the final decision on implementation must be passed on to the RTL synthesizer. Since different implementations are required for elementary operations, technology changes after allocation are not possible without an extensive module library. As the *PIPE* module database does not currently contain the necessary different implementation methods for elementary operations, the *HSPIPE* system does not currently handle technology assessment.

It must be noted that implementing a technology assessment step after allocation may increase the length of the design process considerably. Since changing implementation timings may influence the restart period of the system, every technology assessment step must be immediately verified to detect violations of restart time constraints.

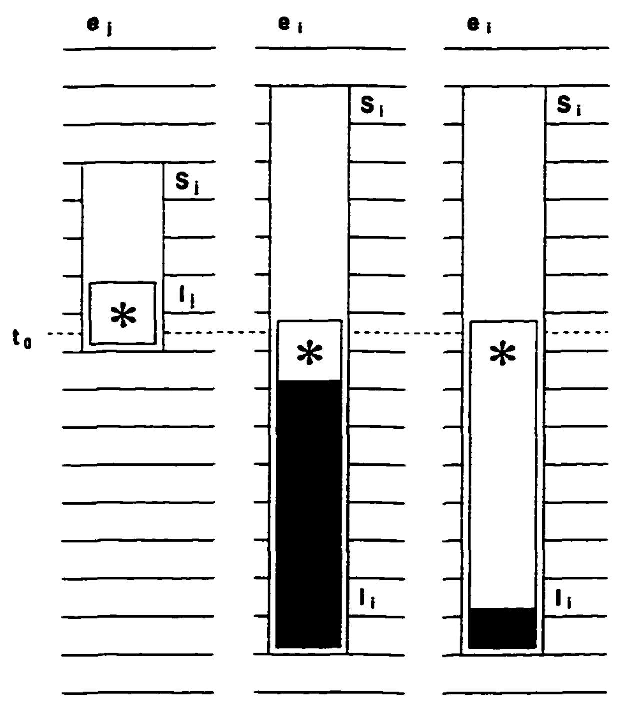

Figure 7-2: Tradeoffs between different implementations of elementary operations

Since software systems are constrained by the capabilities of the underlying hardware, similar tradeoffs are generally not available to low-level software modules.

7.5 Transfer cost functions for reconfigurable systems

Reconfigurable hardware, as an emerging technology in microelectronics systems, offers hardware speeds at the flexibility of software.

Most reconfigurable systems are currently implemented as on-the-fly reloadable soft-programmable gate-arrays. Such systems store their programs in a non-volatile (NV) memory much larger than their actual program RAM, and reload RAM contents from different sections of NV memory if required. Once loaded, the program executes at a speed much closer to hardware than that of software. Typical clock speeds of reloadable field programmable gate-arrays (FPGAs) are in the range of several hundred MHz (as of early 1999). Such speeds, coupled with the available gate number of several hundred thousand logic gates in each package and the number of I/O lines (several hundred per package) offer a computing power matching that of the majority of current microprocessors. Unlike microprocessors, FPGAs execute instructions encoded in hardware, offering practically no overhead on "instruction decoding" and execution.

Reprogrammable hardware relies on program loads from external storage, not unlike "overlay" executables executing on traditional microprocessors. Every time an operation requires data from NV storage, the system must halt execution as it transfers code from NV memory to FPGA main memory. Since unexpected program reloads introduce unexpected delays in algorithms, execution times may not be treated as uniform in the model of a reconfigurable system. Introducing an approximation of load times increases the usefulness of our model to include reconfigurable systems.

To represent the program loads, one must use an additional different cost function for modeling hardware reconfiguration. Any vertex executing an operation which might require reloading has to be tagged with a "reload" vertex. The function of a reload vertex is to model the additional delay before executing the given elementary operation. The reload vertex has no functionality if the currently loaded code context (τ_s where s stands for for *source context*) and the required context (τ_d for *destination context*) are identical. A non-functional reload vertex may be approximated with a zero execution time vertex (or simply omitted) if the controller is advanced enough or the compiler detects the unneeded load during optimization. In this case, the reload vertex is simply removed from the DFG. In systems where every potential load requires checking, the non-functional reloads require a small, typically fixed amount of time.

In practical terms, the code reload vertices may be modeled as delays, similarly to non-reconfigurable MCHLS. (These delays are not to be placed in the system as buffers, obviously, since the reload mechanism generates them.)

$$t(n, \tau_s, \tau_d) = \begin{cases} 0, & \text{for } \tau_s = \tau_d \\ f(n, \tau_d), & \text{for } \tau_s \neq \tau_d \end{cases}$$

Lacking hands-on experience with practical reconfigurable systems, creating the necessary libraries for code reload models is not possible as of May 11, 1999. For this reason, research on reload timing has been delegated to future development. (Once the reload functions are available, implementing support for on-line reconfigurable hardware in *HSPIPE* is trivial.)

Chapter 8

Glossary

ALAP: As Soon As Possible. Scheduling term for the latest time cycle when an elementary operation may start processing its input data. Delaying the start of data processing after the ALAP time of an operation violates maximum system latency constraints.

Approximation Algorithm: polynomial-time solutions minimization or maximization problems that have a guaranteed upper bound (lower bound for maximization problems) on the ratio between the worst case solution cost and the optimum solution cost.

ASAP: As Soon As Possible. Scheduling term for the earliest time cycle when an elementary operation may start processing its input data. Processing may not start before the ASAP cycle since data on operation inputs may not be stable before that time.

ASIC: Application-Specific Integrated Circuit. Custom integrated circuits designed to solve specific problems. ASICs are generally more efficient than commercial, general-purpose designs at the expense of extended development time and higher cost.

bitfile: binary file containing CLB configuration for soft-programmable FPGAs, downloaded to the FPGA upon power-up.

BME: Budapesti Műszaki Egyetem. Official Hungarian name of the Technical University of Budapest, Hungary.

CAD: Computer-Aided Design.

CDFG: Control-Data-Flow Graph. A graph representation of data propagation inside a system. Vertices are data operations, edges are data connections. The system contains additional information so that control structures may be generated from the data-flow.

CFG: Control-Flow Graph. A graph representation of control information inside a system.

CISC: Complex Instruction Set Computer. Microprocessor structure, where instructions are executed through internal decoding (microcode). The performance penalty of instruction decoding slows down instruction execution. CISC microprocessors typically contain a large number of addressing modes, low internal parallelism, and a small number of internal registers. See also: RISC.

CLB: Configurable Logic Block. Individually programmable, basic functional units of Xilinx FPGAs, capable of implementing small amounts of memory (implemented as several small look-up tables), combinatorial logic, small multiplexer blocks, or combinations thereof.

DAG: Directed Acyclic Graph. A directed graph without cycles, representing data transfers and operations in HLS. Usually used as a synonym of DFG in system-level synthesis context.

DFG: Data-Flow Graph. A graph representation of data propagation inside a system. Graph vertices are data operations, edges represent data transfers.

EEOG: Extended Elementary Operation Graph. A DFG describing system functionality as a set of non-elementary operations. EEOG operations may be decomposed to more than one primitives of the underlying RTL library.

EEPROM: Electrically Erasable Programmable Read-Only Memory. Read-only memory that may be reprogrammed on-line (without removing from the circuit) by applying a sufficiently high programming voltage to it (much higher than regular operational voltages). See also EPROM.

EOG: Elementary Operation Graph. A DFG describing system functionality as a set of elementary operations. Unlike an EEOG, EOG vertices are implemented as a single primitive of the underlying RTL library.

EPROM: Erasable Programmable Read-Only Memory. Read-only memory devices that may be reprogrammed off-line with radiation of sufficient energy levels (typically ultraviolet illumination). See also EEPROM.

FPGA: Field-Programmable Gate-Array. A class of microelectronics devices containing user-programmable logic blocks. FPGAs run their programmable submodules at hardware speeds, while retaining the flexibility of software for programming. The two main groups of FPGAs are permanently programmable and RAM-based (soft or re-programmable) FPGAs.

GA: Gate-Array. Semi-custom hardware systems where general-purpose transistor arrays are customized by generating custom metallization layers and connections over them.

HDL: Hardware Description Language. High-level programming languages and descriptions used as sources to generate hardware systems.

HLS: High-Level Synthesis. Automated processes converting high-level problem descriptions to low or medium-level hardware descriptions.

HSCD: Hardware-Software Co-Design.

HSPIPE: Hardware-Software (extensions to) PIPE. An extended design process, incorporating hardware-software codesign development to the hardware-oriented *PIPE* CAD environment.

IP: Intellectual Property (block). Generic term for standalone off-the-shelf submodules in synthesizable software/high-level descriptions (soft IP), technology-independent netlists (firm IP) or technology-specific layouts (hard IP). IP blocks may be integrated to custom designs through standardized interfaces. IP blocks typically implement standalone functions in a reusable, modular way.

Kernighan-Lin: an incremental, $\mathcal{O}(n^2)$ graph partitioning algorithm. Attempts to improve system metrics by investigating the effect of local changes (vertex attribute swaps) on the system cost function.

Kernighan-Lin-Fiduccia-Mattheyses: an incremental, $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$ graph partitioning algorithm [FM82], extending the original work of Kernighan and Lin [KL70]. Attempts to improve system metrics by moving individual DFG vertices from one execution context to another, and observing the effect of changes in the system cost function.

KL: See Kernighan-Lin.

KLFM: See Kernighan-Lin-Fiduccia-Mattheyses.

Loop unrolling: A compiler optimization technique for loop constructs. Improves compiled code properties by replicating the instructions in a loop, reducing loop overhead, and increasing potential instruction-level parallelism.

MCE: Multiple-Context Environment. Target systems where functional units are mapped to several disjoint, architecturally or fundamentally different *execution contexts* necessitating additional lower-level modules for communication between them. Examples are hardware-software codesign (software, hardware) or multiprocessors (software, software).

MCHLS: Multiple-Context High-Level Synthesis. Application of High-Level Synthesis techniques to generate systems where the target technology includes hardware and software components.

MCCDFG: Multiple-Context Control-Data-Flow Graph. A graph representation of data in multiple-context environments. MCCDFGs represent problems where solutions are generated in different executions contexts. Example execution content groups are hardware and software, multiple hardware (processor-coprocessor), multiple software (multiple CPUs, MIMD or SIMD) environments, or combinations thereof.

MIMD: Multiple Instruction, Multiple Data. Parallel, multiprocessor systems where processors are processing different instruction and data streams. See also SIMD.

NP Complexity: (NP) a complexity class of languages (problems) where a polynomial-time algorithm may verify the correctness of a solution.

NP-Complete: a complexity class of languages (problems) which have NP complexity and may be transformed to another problem of NP complexity using a polynomial algorithm. NP-Complete problems are the hardest problems in the NP complexity class.

NP-Hard: a complexity class of languages (problems) which may be transformed to another problem of NP complexity using a polynomial algorithm.

PIPE: A High-Level Synthesis CAD tool developed at BME (Technical University of Budapest, Hungary). The tool is capable of synthesizing pipelined hardware systems (hence the name).

Polynomial-Time: a complexity class of languages (problems) where a solution of a problem of size n may be found in $\mathcal{O}(n^k)$ time, where k is a constant. See also: NP-Hard, NP-Complete.

Recursion: a class of functions (or procedures in programming) where results are produced by the function using a value returned by repeated calls of the same function (for subproblems of smaller sizes). See also: Recursion.

RISC: Reduced Instruction Set Computer. Collective name for microprocessors and microcontrollers where the number of available instructions is very small. RISC instructions are executed without internal decoding (microcode) and are faster than the microcode of CISC processors. Most RISC devices heavily penalize external memory accesses, feature a high degree of potential parallelism inside the processor, and have a high number of internal registers. Efficient RISC code attempts to minimize the number of external memory accesses. See also: CISC.

RTL: Register Transfer Level. An intermediate-level description of hardware systems. RTL descriptions capture system properties as a set of registers, basic arithmetic units (ALUs), control flow (transfer sequences and conditional execution), interconnect network, elementary binary functions, and system hierarchy.

SIMD: Single Instruction, Multiple Data. Parallel, multiprocessor systems where multiple processors are processing data streams, while executing the same instructions.

SLS: System-Level Synthesis. The procedure of generating complete systems as an integrated design process, as opposed to non-integrated, lower-level procedures. Used as a synonym of Hardware-Software Co-Design in this dissertation.

SOG: Sea of Gates. A variant for Gate-Array (GA) semi-custom hardware technology, featuring two-dimensional arrays of pre-fabricated transistors with custom interconnects.

Spill code: auxiliary code sections in software, inserted after the register requirements of a given code section become available. Spill code may be necessary in software that requires more CPU registers than the available amount. Spill code is inserted after register allocation to save and restore CPU register contents and free registers for temporary storage.

Verilog: once proprietary, currently standardized (open) HDL. Similar to VHDL functionality, offers better properties at hardware integration and performs worse in high-level and mixed-level system descriptions. See also: VHDL.

VHDL: VHSIC (Very High Speed Integrated Circuit) HDL. A standardized HDL with different levels of abstraction, usable both for simulations and for direct hardware synthesis. See also: Verilog.

Bibliography

[ABIC$^+$98] Mohamed Abid. T. Ben Ismail, A. Changuel, C.A. Calderram, M. Romdhani, G.F. Marchioro. J.M. Daveau, and Ahmed A. Jerraya. A hardware-software co-design methodology for design of embedded systems. *Integrated Computer-Aided Engineering*, pages 69–83, March 1998.

[AHK96] Charles J. Albert. Lars W. Hagen, and Andrew B. Kahng. A hybrid multi-level/genetic approach for circuit partitioning. In *Proc. of the ACM SIGDA Physical Design Workshop*, pages 100–105, April 1996.

[AJ97] Mohamed Abid and Ahmed Jerraya. Towards hardware-software co-design: A case study of robot arm controller. *Journal of Microelectronics System Integration*, 5:167–182, 1997.

[AJV] Péter Arató, István Jankovits, and Tamás Visegrády. *High-Level Synthesis (draft)*.

[AL98] Cleve Ashcraft and Joseph W. H. Liu. Applications of Dulmage-Mendelsohn decomposition and network flow to graph bisection improvement. *SIAM Journal on Matrix Analysis and Applications*, pages 325–354, 1998.

[ASU88] Alfred Aho, Ravi Sethi. and Jeffrey Ullman. *Compilers, Principles, and Tools.* Addison-Wesley. Reading, MA, second edition, 1988.

[Ata98] Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook.* CRC Press, 1998.

[AV98] Péter Arató and Tamás Visegrády. Effective graph generation from VHDL structures. *Microelectronics Journal*, 29, March 1998.

[Bal84] Krishnamurthy Balakrishnan. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, pages 438–446, May 1984.

[BFS98] Alessandro Balboni, William Fornaciari, and Donatella Sciuto. Partitioning of hardware-software embedded systems: A metrics-based approach. *Integrated Computer-Aided Engineering*, March 1998.

[BML97] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Optimized software synthesis for synchronous dataflow. In *Proc. of ASAP 97*, March 1997.

[BML98] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing*, 1998.

[BR95] Vasanth Bala and Norman Rubin. Efficient instruction scheduling using finite state automata. In *Proc. of the 28th Annual International Symposium on MicroArchitecture*, November 1995.

[BR96] Vaughn Betz and Jonathan Rose. Directional bias and non-uniformity in FPGA global routing architectures. In *Proc. of the 1996 IEEE/ACM international conference on Computer-Aided Design*, pages 652-659, 1996.

[Bro95] Frederick P. Brooks. *The mythical man-month*. Addison-Wesley, Reading, MA, anniversary edition, October 1995.

[BS98] Giacomo Buoanno and Mariagiovanna Sami. Co-testing: Granting testability in a codesign environment. *Integrated Computer-Aided Engineering*, March 1998.

[CLR90] Thomas Cormen, Charles Leierson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[Cor96] IBM Corporation. *PPC403GC Embedded Controller User's Manual*. IBM Corporation, second edition, July 1996.

[DD96a] Shantanu Dutt and Wenyong Deng. A probability-based approach to VLSI circuit partitioning. In *Proc. of the Design Automation Conference*, 1996.

[DD96b] Shantanu Dutt and Wenyong Deng. VLSI circuit partitioning by cluster-removal using iterative improvement techniques. In *Proc. of the IEEE/ACM International Conference on CAD Systems*, 1996.

[Dew97] Allen Dewey. *Analysis and Design of Digital Systems with VHDL*. PWS Publishing, Boston, MA, 1997.

[DK91] Nikil D. Dutt and James R. Kipps. Bridging high-level synthesis to RTL technology libraries. In *Proc. of the 28th Design Automation Conference*, pages 526–529, 1991.

[DMG92] Giovanni De Micheli and Rajesh Gupta. System-level synthesis using reprogrammable components. In *Proc. of the European Conference on Design Automation Conference*, 1992.

[EH92] Ralf Ernst and J. Henkel. Hardware/software codesign of embedded controllers based on hardware extraction. In *Handouts of the 1st International Workshop on Hardware/Software Codesign*, 1992.

[Ele98] Electronic Frontier Foundation. *Cracking DES - Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly & Associates, Cambridge, MA, July 1998.

[ET98] Michael Eisenring and Jürgen Teich. Domain-specific interface generation from dataflow specifications. In *Proc. of Codes/CASHE'98, the 6th Int. Workshop on Hardware/Software Codesign*, pages 43–47, March 1998.

[ETT98] Michael Eisenring, Jürgen Teich, and Lothar Thiele. Rapid prototyping of dataflow programs on hardware/software architectures. In *Proc. of the Hawaii Int. Conf. on Syst. Sci.*, pages 187–196, January 1998.

[FM82] Charles M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. of the ACM/IEEE Design Automation Conference*, 1982.

[Fuh91] Thomas E. Fuhrman. Industrial extensions to university high level synthesis tools: Making it work in the real world. In *Proc. of the 28th Design Automation Conference*, pages 520–525, 1991.

[Gaj88] Daniel Gajski, editor. *Silicon Compilation*. Addison-Wesley, Reading, MA, 1988.

[GDZ98] Daniel D. Gajski, Raimer Doemer, and Jianwen Zhu. IP-centric methodology and design with the SpecC language. In *NATO ASI System-Level Synthesis*, August 1998.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[Gra66] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*. 1966.

[Gup93] Rajesh Kumar Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems.* PhD thesis, Stanford University, December 1993.

[GVNG98] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. *Transactions on VLSI Systems*, 6:84–100, 1998.

[GW96] Ian Goldberg and David Wagner. Architectural considerations for cryptoanalytic hardware. Technical report, Department of Computer Science, University of California at Berkeley, 1996.

[HB95a] Scott Hauck and Gateano Borriello. An evaluation of bipartitioning techniques. In *Proc. of the Chapel Hill Conference on Advanced Research in VLSI*. 1995.

[HB95b] Scott Hauck and Gateano Borriello. Logic partition ordering for multi-FPGA systems. In *Proc. of the International Symposium on Field-Programmable Gate Arrays*, 1995.

[HBE94] Scott Hauck, Gateano Borriello, and Carl Ebeling. Mesh routing topologies for FPGA arrays. In *Proc. of the 1994 IEEE/ACM international conference on Computer-Aided Design*, 1994.

[Hea93] Thomas Heath. Automating the compilation of software into hardware. Master's thesis, University of Oxford, September 1993.

[HKed] Bruce Hendrickson and Tamara G. Kolda. Graph partition models for parallel computing. *Parallel Computing*, (submitted).

[HLYC91] C.-T. Hwang, J.-H. Lee, and Hsu Y.-C. A formal approach to the scheduling problem in high-level synthesis. *IEEE Transations on Computer Aided Design*, April 1991.

[Hoc97] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, Boston, MA, 1997.

[HX98] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing : Technology, Architecture, Programming*. McGraw-Hill, January 1998.

[IEE88] IEEE. *IEEE Standard VHDL Reference Manual*. IEEE, 1988.

[Jha95] Pradi Kumar Jha. *High-Level Mapping for RT Components*. PhD thesis, University of California, Irvine, 1995.

[JRV+98] Ahmed A. Jerraya, M. Romdhani, C. Valderrama, Ph. Le Marrec, F. Hessel, G. Marchioro, and J. Daveau. Models and languages for system-level specification and design. In *NATO ASI on System-Level Synthesis, Proc.*, 1998.

[Kal95] Asawaree Kalavade. *System Level Codesign of Mixed Hardware-Software Systems*. PhD thesis, University of California, Berkeley, September 1995.

[Ker93] Daniel R. Kerns. Balanced scheduling: instruction scheduling when memory latency is uncertain. *Conference on Programming Language Design and Implementation*, July 1993.

[KK95] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical report, Department of Computer Science, University of Minnesota, 1995.

[KK97] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. Technical report, Department of Computer Science, University of Minnesota, 1997.

[KL70] Brian W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning electrical circuits. *Bell System Technical Journal*, 49:291–307, February 1970.

[KL97] Asawaree Kalavade and Edward A. Lee. The extended partitioning problem: Hardware/software mapping, scheduling, and implementation-bin selection. *Journal of Design Automation for Embedded Systems*, 2:125–163, 1997.

[Knu95] Peter Voigt Knudsen. Fine-grain partitioning in codesign. Master's thesis, Technical University of Denmark, Lyngby, 1995.

[Knu99] Donald E. Knuth. MMIX. Research project of the update to "The Art of Computer Programming", February 1999.

[Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15:37–46, January 1982.

[Las93] Gregor von Laszewski. A collection of graph partitioning algorithms. Technical report, Northeast Parallel Architectures Center, Syracuse University, May 1993.

[LE95] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Conference on Programming Language Design and Implementation*, June 1995.

[LH94] Robert Leland and Bruce Hendrickson. An empirical study of static load balancing algorithms. In *Proc. of SHPCC '94*, 1994.

[LLSV98] Jie Liu, Marcello Lajolo, and Alberto Sangiovanni-Vincentelli. Software timing analysis using hw/sw cosimulation and instruction set simulator. *Proc. of the Sixth International Workshop on Hardware/Software Codesign*, pages 65–70, March 1998.

[LSVS98] Luciano Lavagno, Sangiovanni-Vincentelli, and Ellen M. Sentovich. Models of computation for system design. In *NATO ASI System-Level Synthesis*, August 1998.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, September 1996.

[ML97] Praveen K. Murphy and Edward A. Lee. Optimizing synchronization in multiprocessor dsp systems. *IEEE Transactions on Signal Processing*, 45, June 1997.

[MW96] Wai-Kei Mak and D. F. Wong. Minimum replication min-cut partitioning. In *Proc. of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, 1996.

[NP95a] Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *MICRO-2*, pages 28–33, November 1995.

[NP95b] Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. In *Proc. of Supercomputing '93*, November 1995.

[NP98] Cindy Norris and Lori L. Pollock. Experiences with cooperating register allocation and instruction scheduling. *International Journal on Parallel Programming*, 26:241–284, September 1998.

[OKD97] Seong Young Ohm, Fadi J. Kurdahi, and Nikil D. Dutt. A unified lower bound estimation technique for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, April 1997.

[OR94] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning. In *Proc. of Supercomputing '94*, August 1994.

[Pag94] Ian Page. Automatic design and implementation of microprocessors. In *Proc. of WoTUG (World occam and Transputer User Group) '94*, 1994.

[Pag95] Ian Page. Constructing hardware-software systems from a single description. Technical report, Oxford Hardware Compilation Research Group, July 1995.

[PB96] Pradeep Prabhakaran and Prithviraj Banerjee. Parallel algorithms for force directed scheduling of flattened and hierarchical signal flow graphs. In *Proc. of the 1996 International Conference on Computer Design*, 1996.

[PD96] Robert Preis and Ralf Diekmann. *The PARTY Partitioning Library User Guide*. Universität Padernborn, Germany, September 1996.

[Pin96] Randall D. Pinkett. Hardware/software co-design and digital signal processing. Master's thesis, University of Oxford, May 1996.

[PK89] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioural synthesis of ASICs. *IEEE Transations on Computer Aided Design*, 1989.

[Qui94] Michael J. Quinn. *Parallel Computing, Theory and Practice.* McGraw-Hill, New York, 1994.

[Ros98] Wolfgang Rosenstiel. Rapid prototyping, emulation and hardware-software co-debugging. In *NATO ASI on System-Level Synthesis, Proc.*, 1998.

[Sch95] Bruce Schneier. *Applied Cyprography.* John Wiley & Sons, Arlington, TX, second, revised edition, December 1995.

[Sta98] William Stallings. *Cryptography and Network Security.* Prentice Hall, Upper Saddle River, NJ, second edition, July 1998.

[SW95] Richard Sites and Richard Witek. *Alpha AXP Architecture Reference Manual.* Digital Press, Newton, MA, second edition, 1995.

[Tei97] Jürgen Teich. *Digitale Hardware/Software-Systeme.* Springer, 1997.

[TV97] Linus Tauro and Frank Vahid. Message-based hardware/software communication in HDL/C environments. In *Proc. of the Asia-Pacific Conference on Hardware Description Languages*, August 1997.

[Vah97a] Frank Vahid. Modifying min-cut for hardware and software functional partitioning. In *Proc. of the International Workshop on Hardware/Software Codesign*, pages 43–48, March 1997.

[Vah97b] Frank Vahid. Port calling: A transformation for reducing I/O during multi-package functional partitioning. In *International Symposium on System Synthesis*, September 1997.

[Vah99] Frank Vahid. Procedure cloning: A transformation for improved system-level functional partitioning. *ACM Transactions on Design Automation of Electronic Systems*, 1999.

[VG92] Frank Vahid and Daniel D. Gajski. Specification partitioning for system design. In *Proc. of the 29th Design Automation Conference*, September 1992.

[VG95a] Frank Vahid and Daniel D. Gajski. Closeness metrics for system-level functional partitioning. In *Proc. of the European Design Automation Conference*, pages 328–333, September 1995.

[VG95b] Frank Vahid and Daniel D. Gajski. Incremental hardware estimation during hardware/software functional partitioning. *IEEE Transactions on VLSI Systems*, September 1995.

[VG98] Frank Vahid and Tony Givargis. Incorporating cores into system-level specification. In *International Symposium on System Synthesis*, December 1998.

[VG99] Frank Vahid and Tony Givargis. The case for a Configure-and-Execute paradigm. In *Proc. of the International Workshop on Hardware/Software Codesign*, 1999.

[VH98] John Villasenor and Brad Hutchings. The flexibility of configurable computing. *IEEE Signal Processing Magazine*, 15, September 1998.

[VNG95] Frank Vahid, Sanjiv Narayan, and Daniel D. Gajski. SpecCharts: A VHDL front end for embedded systems. *IEEE Transactions on CAD*, 14:694–706, 1995.

[VNG97] Frank Vahid, Sanjiv Narayan, and Daniel D. Gajski. Extending the Kernighan/Lin heuristic for hardware and software functional partitioning. *Kluwer Journal on Design Automation of Embedded Systems*, 2:237–261, March 1997.

[VT97] Frank Vahid and Linus Tauro. An object-oriented communication library for hardware-software codesign. In *International Workshop on Hardware/Software Codesign*, pages 81–87, March 1997.

[WCS96] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly, second edition, September 1996.

[WE93] Neil E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI design*. Addison-Wesley, Reading, MA, second, revised edition, 1993.

[Wir98] Niklaus Wirth. Hardware compilation: Translating programs into circuits. *IEEE Computer*, June 1998.

[YMS+98] James Shin Young, Josh MacDonald, Michael Shilman, Abdallah Tabbara, Paul Hilfinger, and A. Richard Newton. The JavaTime approach to mixed hardware-software system design. In *NATO ASI System-Level Synthesis*, August 1998.

[YW94] H. Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. In *Proc. of the 1994 IEEE/ACM international conference on Computer-Aided Design*, pages 50–55, November 1994.

[ZEK+98] D. Ziegenbein, Ralf Ernst, Richter K., Jürgen Teich, and Lothar Thiele. Combining multiple models of computation for scheduling and allocation. In *Proc. of the 6th International Workshop on Hardware/Software Codesign*, 1998.

Appendix A

Effective Graph Generation from VHDL Descriptions

This chapter originally appeared as [AV98]. It is reproduced here partially to complement some of the dissertation discussions on generating useful elementary operation graphs from higher-level descriptions.

Key words

Scheduling, data path description, data dependency graph, high-level synthesis.

Abstract

The transformation between a problem description and a data dependency graph is a step which results in a significant reduction of freedom during high-level synthesis. This paper presents an evaluation of different graph generation methods and makes a suggestion on the methods to be employed in the solution of such a problem.

High-level synthesis takes its input written in an artificial language, i.e. VHDL or one of several similar languages [IEE88]. These descriptions take the form of functions which must be transformed to a hardware realization using the steps of initial allocation, scheduling and allocation.

A.1 Data dependency graph

Problems that are given as functions may be transformed to data dependency graphs which feature operations as vertices in the graph and direct data transfers as edges. The *structure* (or *layout*) of this graph affects the efficiency of scheduling since it sets the time interval in which an operation may be moved (also known as *operation time frame*). The length of the time frame of an operation is equal to the *(operation) mobility*, with a higher mobility belonging to an operation which may usually be scheduled yielding better results.

Note that *layout* is used as a description of graph structure and not in the sense which is encountered in high-level synthesis.

It is usually better to postpone the limitations to the stage of scheduling. Otherwise the scheduling step may not influence the overall efficiency of the system since the scheduler gets the graph in a fixed state (without mobility) or in such a condition that the decreased mobility of operations results in an infeasible result.

The elementary operations in the graph may be described as

- every operation has one or two data inputs (not counting control signals).

  We investigate *binary operations* since a *unary operation* may be neglected from the data dependency graph without changing the topology of the system, if it is replaced with a suitable, topologically equivalent single-input, single-output vertex. This single-input, single-output vertex may be modeled as a timing constraint between the vertices connecting its predecessors with its successors.

  Note that binary is now used in the mathematical sense, i.e. an operation having two operands.

- every operation has exactly one data output.

  This single output may be connected to more than one of the operations, but it must be a single value.

Since scheduling algorithms affect the graph by inserting buffers as additional vertices between pairs of edges , original operations are usually referred to as *functional elements*. *Synchronization buffers* (i.e. buffers that guarantee that data arrives to different inputs of operations in a suitable time point) are not considered to be functional elements. These buffers are to be placed into the graph when the relative position of the elementary operations is fixed and so are not important at the stage of graph generation.

Buffers are inserted the graph to increase the data propagation time of a given edge (and to provide storage between slow operations) and are generally referred to as *delay*. As mentioned before, delay differs from functional elements as it is a unary operation.

The properties of the graph are described using two numeric values:

depth refers to the number of levels a graph has, i.e. the maximum number of sequentially executed functional elements. A graph with a greater depth requires more time to calculate its output values (i.e. has a higher *latency*).

width describes the maximum number of simultaneously executing operations in a graph. It is useful to use width both as a time function and as a parameter of the whole graph. As a time function, the width belonging to cycle t ($W(j,t)$) is equal to the number of type j functional elements executing in that cycle. As a parameter, $W(j)$ refers to the maximum of $W(j,t)$ for every cycle during the data propagation in the graph. To build a system with a width of $W(j)$, one must build a sufficient number of type j processors so that there are always enough processors to start all the operations scheduled to a given cycle. Therefore the number of processors ($M(j)$) and graph width must be so related that

$$\max W(j,t) = W(j) \leq M(j)$$

for every processor type.

For our investigations we take a simple function that evaluates the sum

$$S = a + b + c + d + e + f + g + h$$

This function is special since its operations are commutative and the operations work on data which are identical in nature (i.e. are composed the same way). This function is not a very special case since most types of the filters have the same structure except for the composition of the operation inputs [2]. The only significant difference is that this simplified model does not contain multiple operation types, while a filter usually contains graph vertices which are not elementary in nature: a FIR filter, for example, resembles the previous structure in such a way that (assuming a 8th order filter)

$$\begin{array}{llll} S_k = & w_k * x_k & +w_{k-1} * x_{k-1} & +w_{k-2} * x_{k-2} \\ & +w_{k-3} * x_{k-3} & +w_{k-4} * x_{k-4} & +w_{k-5} * x_{k-5} \\ & +w_{k-6} * x_{k-6} & +w_{k-7} * x_{k-7} & \end{array}$$

Should a structure be introduced to the system which performs the following operation:

$$f(w, k) = x_k * wk$$

, the graph system graph would be equivalent to the following:

$$\begin{array}{llll} S_k = & f(w_k, x_k) & +f(w_{k-1}, x_{k-1}) & +f(w_{k-2}, x_{k-2}) \\ + & f(w_{k-3}, x_{k-3}) & +f(w_{k-4}, x_{k-4}) & +f(w_{k-5}, x_{k-5}) \\ + & f(w_{k-6}, x_{k-6}) & +f(w_{k-7}, x_{k-7}) & \end{array}$$

This form of the problem is suitable for graph generation since it contains the required types of operations. The f functions must be realized in hardware (see Figure A-9.).

The number of functional elements in the graph is denoted by $N - 1$, which means that this problem type requires N operands. An elementary operation requires T cycles to produce its output after its inputs are stable.

A.1.1 Binary structures

One of the possible extreme structures of the graphs belonging to this function is the binary structure, composed in a way that data propagates in a binary tree. (Figure A-1.) Another possible name of this structure is the *triangular layout.* Since two is the maximum of data inputs for an operation, the binary structure is the global optimum if depth is to be optimized (i.e., it is impossible to find a structure with a lower latency than the binary layout if the number of commutative operations is the same). Since a binary tree has

$$V = \lceil \log_2 N \rceil$$

levels, the latency of the graph may not be lower than $V \cdot T$.

The width of a binary layout is equal to 2^V. The number of required processors is decreasing as data enters deeper levels of the graph since two operations supply the inputs of every operation in the graph. This results in an increasing number of unused processors as time increases.

Figure A-2. presents the utilization in a binary structure which has four levels and so a width of 8. The system is built with the same number of processors. Time is measured in the steps of T, so the utilization chart has one column for every processor and every new row means that another T time cycles have passed. We assume that no buffers are inserted between elementary operations.

The first row means the time between time cycles 0 and $T - 1$. During this period the first 8 elementary operations process their input data. After time cycle T data enters the operations which are directly connected to outputs of the previously utilized operations. Since every output is connected to one input and every operation has 2 inputs, the number of utilized operations is decreased to 4 in cycle T. After cycle $2 \cdot T$ the 4 operations finalize their outputs and the next level of operations begins executing. It is clear that this utilization is an exponentially decreasing function of time and so its width is determined by the number of operations on the entry level.

If the binary structure is described in mathematical notation using brackets, the graph presented in Figure A-1. is equivalent to the following organization:

$$S = \underbrace{(\underbrace{(a+b)}+\underbrace{(c+d)})} + \underbrace{(\underbrace{(e+f)}+\underbrace{(g+h)})}$$

A.1.2 Linear structures

Another extreme structure type is the linear structure, composed in a way that functional elements form a linear branch. (Figure A-3.) Since no pair of operations executes simulta-

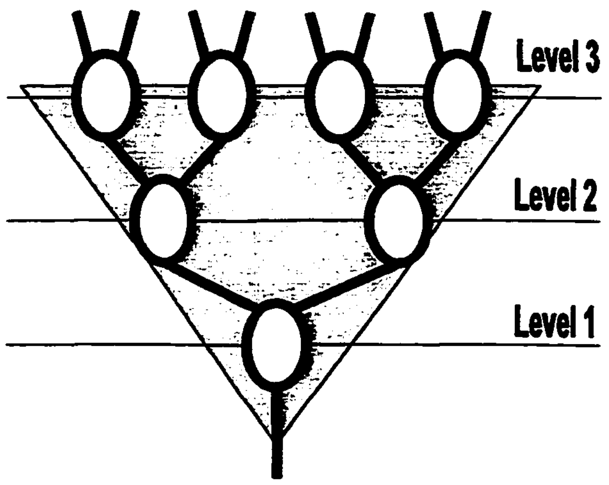

Figure A-1: Binary structure (Triangular layout)

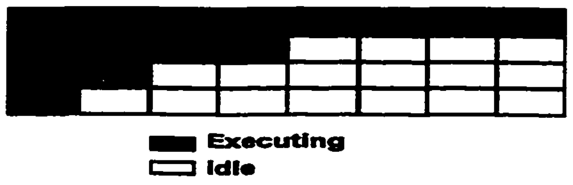

Figure A-2: Utilization of a binary structure

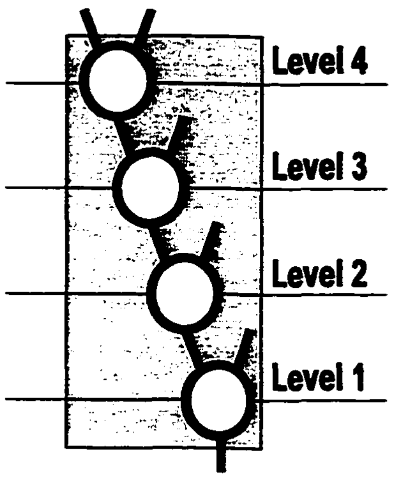

Figure A-3: Linear structure (Rectangular layout)

neously, this organization offers more chance for allocation than a binary layout.

The width of a linear structure is uniform 1. Since this structure requires less processors than a binary layout, it is generally cheaper. The disadvantage of the linear graph is the increased latency: since N operations must execute in a linear way, the result may not appear before $N \cdot T$ cycles after the system input appears. This approach does not deal with buffers inserted to the system, which increases this time.

Should buffers be inserted to the graph, the most probable way is to insert one buffer between adjacent functional elements. This step may be a result of obtaining the desired restart time, when functional elements have a high transfer score and must be separated using buffers. This increases the linear latency (LL) to

$$LL = N \cdot T + (N - 1)$$

The latency of a binary structure under similar conditions is extended to

$$LB = T \cdot \lceil \log_2 N \rceil + (\lceil \log_2 N \rceil - 1)$$

as the decreased depth reduces the number of connections between functional elements.

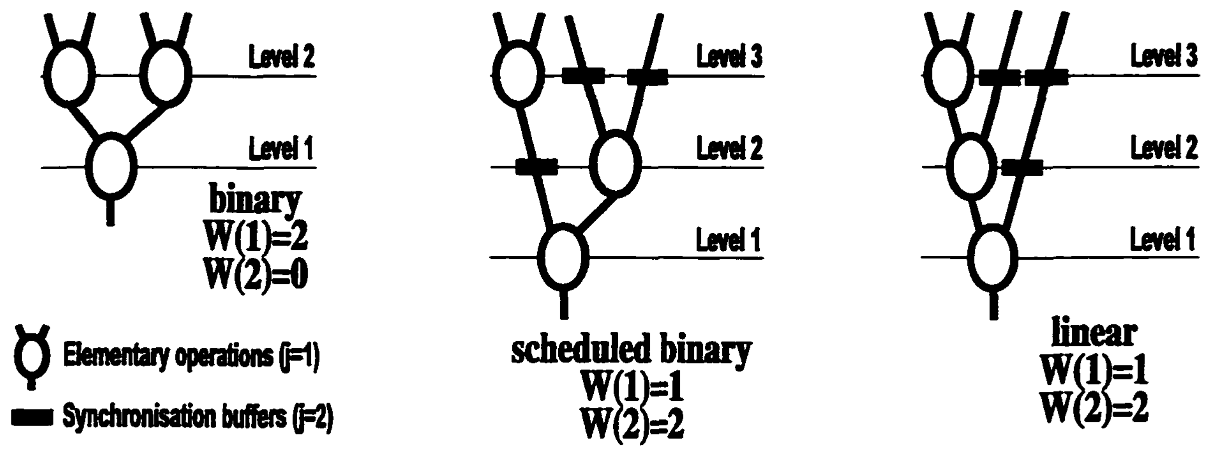

Figure A-4: Transition between binary and linear layout

A linear graph structure is the equivalent to the following bracket pairs:

$$S = (((((((a+b)+c)+d)+e)+f)+g)+h)$$

A.1.3 Transitions between binary and linear structures

Should a binary structure be scheduled using an algorithm capable of dealing with a hardware constraint, the resulting structure is an extended version of the binary triangle (an intermediate structure). As an extreme value, a maximum of one may be prescribed for graph width, which is equivalent to the requirement of the linear structure. The result of this scheduling is equivalent to the linear layout if we consider the width of the graph and the number of utilized buffers. (See Figure A-4.)

Since the binary structure may be extended to an equivalent of the linear structure, it is a useful starting point for feasibility calculations. Considering the maximal hardware requirements of a binary layout, a balance must be maintained between latency and graph width.

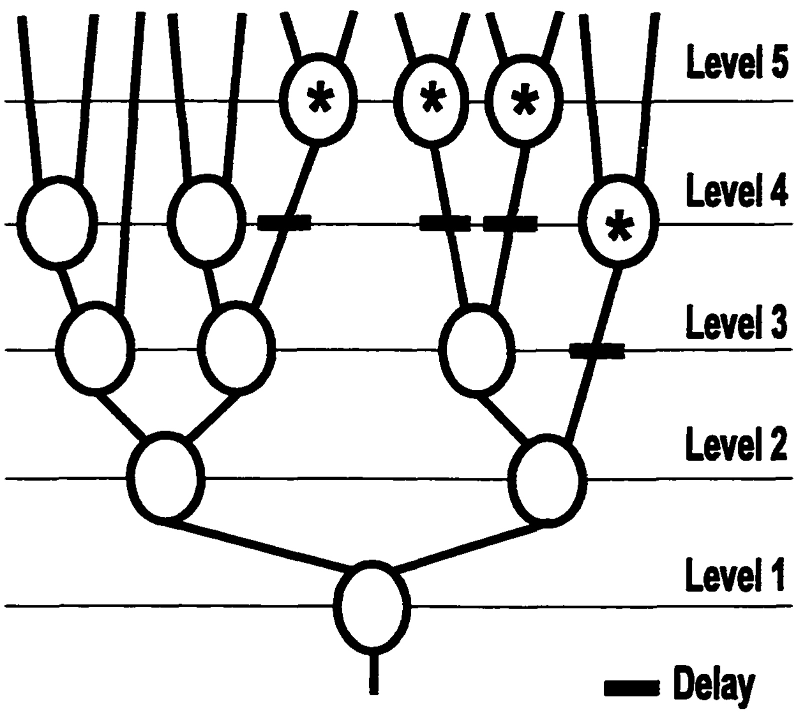

Figure A-5: Binary structure produced under a width limit

A.1.4 Fixed-width binary structures

A useful method to improve the properties of the binary layout is to order the operations in a binary tree with an additional condition for the maximum of the simultaneously used processors. This structure may be produced by an algorithm similar to list scheduling [PK89] (with a hardware constraint).

The example given in Figure A-5. started off as a binary layout and was treated with an algorithm similar to list scheduling. Since a width maximum of 3 was prescribed for the system, some of the operations (marked with a star) were moved in the graph (buffers were inserted after them) so that the number of simultaneously utilized processors was kept below 4. This step required the operations to be separated from their successors, which is indicated with the delay elements. Since the binary structure is symmetrical by nature, the operations to be moved could be selected in any way; the operations marked with a star were chosen because they exceeded the hardware limit (as it was checked from left to right).

Three methods are available for the generation of this structure. The first method generates the binary tree starting from the root (the last element), delaying any operations which would violate the hardware limit. Another possible method is to generate a full binary tree and move the operations starting from the operations which are nearest to the system inputs (leaves of the tree).

The third possible algorithm to generate a fixed width binary structure is to proceed with the generation of the binary tree up to the point where the width is equal to the width limit. Linear structures may be appended to these points so that the width is not increased further.

This algorithm may be implemented as a special type of list scheduling. Some of the list-based scheduling methods are known to be able to schedule a graph in such a way that it stays under hardware limits.

A.2 Generation of intermediate structures using list scheduling

List scheduling methods deal with a graph in such a way that a priority function is defined for operations. Time domain is scanned in increasing order, with suitable operations being started in the current operation, the rest is delayed. Priority function is based on the time frame of the operation, with the highest priorities being awarded to operations with the lowest mobilities. In a special form of list scheduling, an additional constraint may be prescribed. This constraint is the external hardware limit. Priority functions are adjusted to penalize a scheduling plan which would violate hardware constraints.

To generate a graph, the list scheduler has to be ran from the root to the leaves, i.e. in a way which is similar to an upside down time scale (or adapt a list scheduler which prefers the ALAP schedule over the ASAP placement). The functional elements are to be placed on every possible edge in the first stage (during the generation of the binary section) and directly following previously placed functional elements (during the generation of the linear branches). This list scheduling method is easily implemented with the introduction of a suitable cost function (which prefers a direct connection between functional elements to a functional element following a synchronization delay).

A.3 Properties of the intermediate structure

A fixed-width binary structure fulfilling the width constraint may be generated in the following way:

- Generate the necessary

  number of binary levels. These levels are completely filled with operations except for the last one (the one with the highest number of operations) which may be incomplete (i.e. with some operations missing). The condition to achieve this is:

  $$2^{A-2} < W \leq 2^{A-1}$$

  where A is the number of binary levels and W is the desired graph width. The conditions imply that the binary tree is deep enough so that it has a suitable number of operations at its last level and it is not deeper than needed (i.e. the number of operations on the second lowest level is less than W). On Figure A-6., a partial binary graph is generated for $W = 3$ (which implies $A = 3$).

- The last binary level consists of W operations. The remaining

  $$2^{A-1} - W$$

  vertices are not filled with functional elements; these connections are used as direct system inputs. This organization offers a total of

  $$2 * W + 2^{A-1} - W$$

  data inputs since the W vertices have two inputs each, while the rest of the connections is single (see Figure A-7.a.). Should linear branches be appended to the suitable vertices, one of the data inputs would be occupied. Since the number of suitable vertices is W, this means that the number of data inputs of such an extended binary structure is 2^{A-1} as the previous value is decreased by is W.

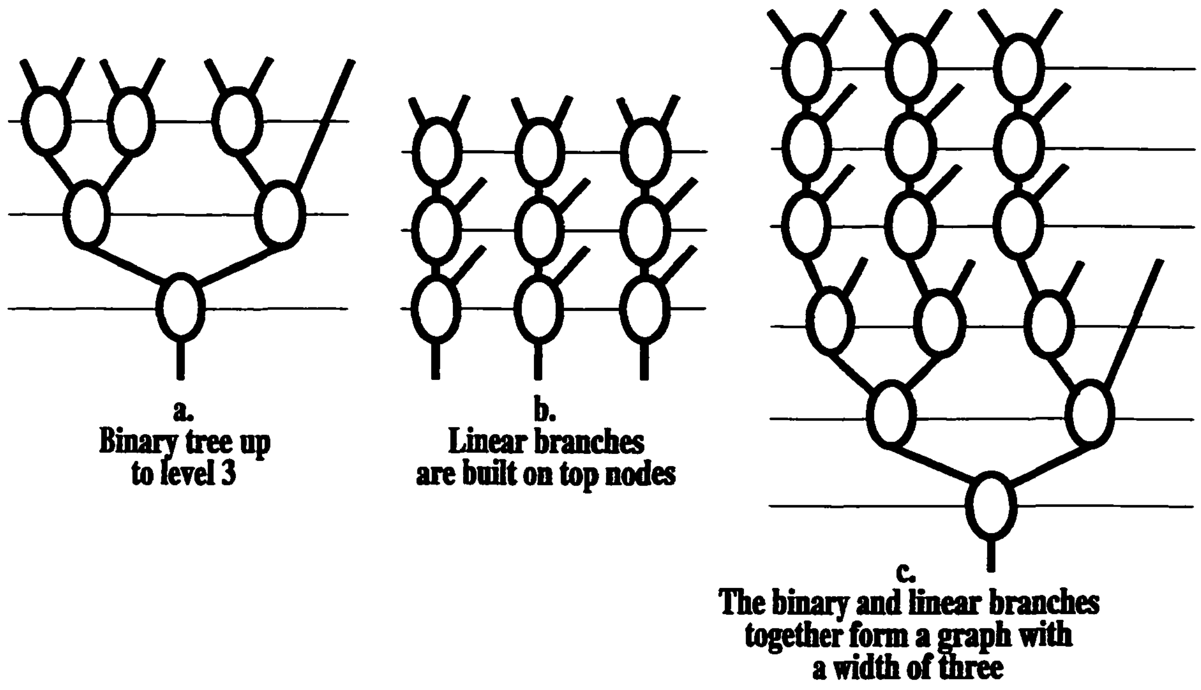
a.
Binary tree up to level 3
b.
Linear branches are built on top nodes
c.
The binary and linear branches together form a graph with a width of three

Figure A-6: Generating a fixed-width binary structure

- Linear branches are appended to the operations of the last level of the binary structure, where one input of these operations is connected to the output of the linear branch, the other is a system input. A linear structure with B levels calculates the result of B operations, and so it takes B+1 system inputs. The depth of the linear branches must be chosen so that the minimum and maximum values do not differ with more than one (since this results in the lowest latency). To find B, one must solve the following inequality (after finding A based on W):

$$W * (B + 1) + 2^{A-1} \geq N$$

(the first term counts the number of linear system inputs, the second the data inputs of the last level of the binary structure. See Figure A-7.)

This B is the maximum of the linear depth values. It is also possible to add linear branches with a lower number of operations (B-1). The number of these branches is found as $W * (B + 1) + 2^{A-1} - N$ since this decrease guarantees that the structure offers a total of N data inputs.

A.3.1 Capacity calculation for the structure

The generated graph with a width of W which may be used to perform identical commutative operations for N inputs, $N-1$ identical functional elements are needed. The intermediate structure should be generated in such a way that

- the number of binary levels is A, where

$$2^{A-2} < W \leq 2^{A-1}$$

- level A of the binary section is not necessarily filled with functional elements (where level 1 denotes the root level). Only W functional elements are put on level A. (The last level is filled if W is an integer power of 2.) This structure stays under the width limit since the width of the levels (which is strictly monotonously increasing) stays under $W+1$ (reaching W at its maximum).

- linear branches are built on the functional elements of level A. Since the number of branches is equal to W, width constraints are not violated. The linear layout does not increase the width further. The total number of arguments an intermediate structure may take is the following: A possible algorithm to generate a fixed width binary structure is to proceed with the generation of the binary tree up to the point where the width is equal to the width limit. Linear structures may be appended to these points so that the width is not increased further.

Figure A-7 is a graph where $W = 3$, $A = 3$ (implied) and $B = 3$ (exactly). The system resembles a FIR filter with $N = 16$.

A.4 Restrictions

This method is useful only in systems where the operations are commutative. This is the case with most of the filters, which are often encountered as high-level synthesis targets, both in benchmarks and in realization. The operations may be non-elementary, for example a FIR filter features two operations (a product and a sum) in each of the vertices. It is

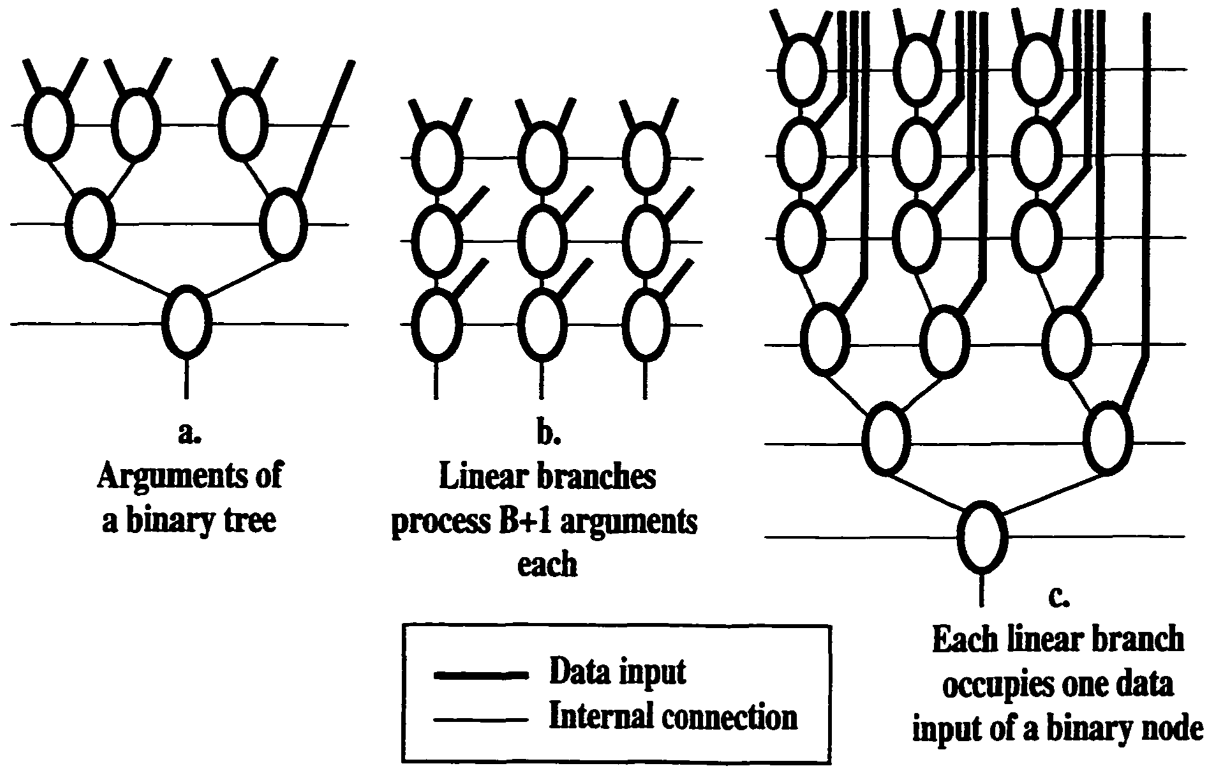
a.
Arguments of a binary tree
b.
Linear branches process B+1 arguments each
c.
Each linear branch occupies one data input of a binary node
Data input
Internal connection

Figure A-7: Data connections of a structure

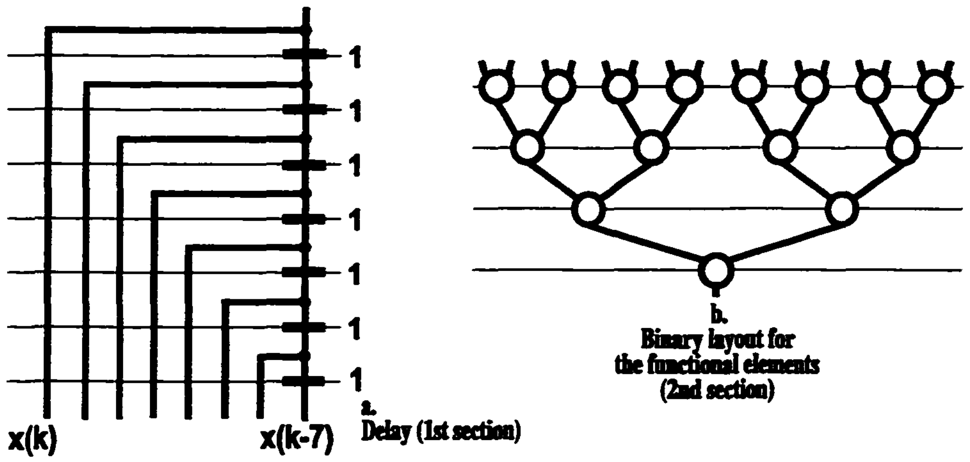

Figure A-8: A possible graph for digital convolution

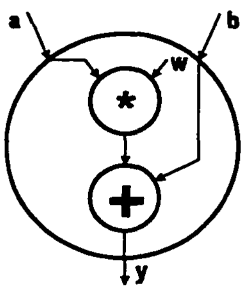

Figure A-9: Functional elements in a FIR filter

therefore feasible to schedule the internals after finding a suitable layout. As an example, the digital convolution (8th order version) may be altered so that it has the necessary delay at its inputs, see Figure A-8. The second part of the graph is then suitable for generation with this method, since its graph is then similar to the FIR algorithm in structure. (Figure A-8.b. shows the functional elements of a graph of a 16th order FIR filter.)

The functional elements are the same as in the FIR filter, since they consist of multiply-add pairs. In this case, an optimizable layout was generated after applying intuition. There are no known graph-generation algorithms which may be used in a general case.

Appendix B

The Design Tool *PIPE*

This chapter originally appeared as a Chapter of [AJV]. It is reproduced here partially to complement some of the dissertation investigations, especially about implementation details. It is also useful to understand the capabilities of the original *PIPE* environment (before multiple-context environment extensions).

PIPE was developed at the Department of Process Control (currently Department of Control Engineering and Information Technology), Technical University of Budapest, as an educational software tool for designing pipeline data flow devices.

PIPE uses an elementary operations graph (EOG) where the vertices of the graph denote elementary operations and the edges their data-connections.

Given a predefined restarting period *PIPE*—if necessary—inserts buffers to meet this period. Synchronization buffers are also inserted.

PIPE generates different variations of the graph by moving the synchronization buffers. For every variation allocation is performed: elementary operations, which are not working concurrently may be combined into one unit. *PIPE* tries to find these units.

The software itself is written in C++ and runs under several variations of the UNIX operating system.

B.1 Usage

The general format of the invocation of the *PIPE* program is:

```
pipe [-s] [-v] [-b] [-Xd] [-p graph] restart [input_file]
```

The name of the program is pipe. This should be in a directory which is accessible by the users (it is in their search path). Some run time parameters may be changed by optional command line switches following the program name. Their order is not important. Table B.1 summarizes them.

Switch	Explanation
-s	The scheduling is not tight. Care should be taken when using this switch as it may increase the number of variation by several magnitudes.
-v	Verbose mode. During the processing additional information is displayed. This includes the number of variations, number, place and types of buffers inserted and the current best graph.
-b	Buffers are normally excluded from allocation. This switch forces buffers to be allocated. This may lead to exponentially increased processing times.
-Xd	Activate d debug option. Only valid if pipe is compiled with debugging enabled. More than one debug option may be given, to list currently available options use -X-.
-p graph	Dump the input graph to a file named graph after inserting buffers. Useful for debugging.

Table B.1: *PIPE* command-line switches

The only compulsory command line parameter is the restart time which should follow the switches if any. This should be given as an integer greater than 2.

The last parameter is the name of the input file. If none is given, *PIPE* reads its standard input. The format of the input is described in detail in Section B.2.

B.2 Input

PIPE uses a simple hardware description language as input. This declares functional elements and gives the interconnection between them.

The following BNF (Backus-Naur Form) description illustrates *PIPE*'s input language:

```
graph       := graphid iodesc fedesc graphdesc outcn
graphid     := GRAPH : name
iodesc      := ioitem | iodesc ioitem;
ioitem      := INPUT namelist | OUTPUT namelist;
namelist    := NAME | namelist , name;
fedesc      := feitem | fedesc feitem;
feitem      := PROCESSOR NAME INPUT: NUMBER DELAY: NUMBER |
PROCESSOR NAME DELAY: NUMBER  INPUT: NUMBER|
PROCESSOR NAME NUMBER NUMBER;
signal      := name FEname (signallist);
signallist  := siglistelem | siglist , sigelem;
siglelem    := INPUTname | SIGNALname;
outcn       := OUTname SIGNALname;
```

Keywords and Identifiers

Inputs, outputs, processors (graph vertices) and interconnections (graph edges) are identified by identifiers of the maximum length of 32 characters (this is a compile time option, and may be changed).

They may contain alphanumerical (a–z, 0–9) characters and underscore. the first character can not be numeral. Identification is not case sensitive (also a compile time option). Forward declarations are not allowed.

The following keywords are reserved, and may not be used as an identifier: **graph**, **input**, **output**, **processor**, **delay**, **out**. Words are separated by blanks or tabs.

Graph declaration

The graph's name is declared by the graph keyword followed by a colon and the name of the graph. The following line declares a graph named **my_graph**:

```
graph: my_graph
```

I/O declarations

Inputs are outputs are declared by the input and output keywords, with the I/O identifiers separated by commas. This example declares **in_a**, **in_b** as an input and **out_x** as an output:

```
input: in_a, in_b
output: out_x
```

Processor declarations

The processor keyword is used to declare processing elements. Two properties have to be given here: the number of inputs and the delay (time from valid input to valid output). The following three lines are all valid declarations of a processor named sum with 4 inputs and a delay of 10:

```
processor sum 10 4
processor sum delay: 10 input: 4
processor sum input: 4 delay: 10
```

Processor instantiations

Processors are instantiated in a form similar to a function call: the arguments are the inputs, the value of the function is the output. Inputs may be named, i.e., using the output of a previously instantiated processor, or unnamed, when the input is an other processor. In this example a processor (**divide**) takes **m1** and **m2** as an input and its output is named as result:

```
result divide(m1, m2)
```

Of course, the processor divide has to be defined in a processor statement, and must have exactly two inputs.

In a similar fashion, divide takes m1 as one input and the output of decrement as the other input:

```
result divide(m1, decrement(m2))
```

Output connections

The outputs declared with the output keyword have to be connected to processor outputs. This line connects result to out_x:

```
out_x result
```

B.3 Output

PIPE's output contains the result of allocation: which functional elements are combined into one.

The result is a table showing which elementary operations have been allocated into one processor. The following listing is a sample output from a FIR filter example.

Processors number 10 and 13 contain two operations (aa2, aa7 and aa5, aa6), all the other processors contain only one. Note that allocation does not attempt to deal with operations that will not fit into one processor: in this case multipliers are not allocated, because their delay is more than half restart period. Their number is still given in the results listing.

```
------------ Results of the allocation ----------

Proc 1 => (a1,adder2)
Proc 2 => (a2,adder2)
Proc 3 => (a3,adder2)
Proc 4 => (a4,adder2)
Proc 5 => (a5,adder2)
Proc 6 => (a6,adder2)
Proc 7 => (a7,adder2)
Proc 8 => (a8,adder2)
Proc 9 => (aa1,adder2)
Proc 10 => (aa2,adder2) (aa7,adder2)
Proc 11 => (aa3,adder2)
Proc 12 => (aa4,adder2)
Proc 13 => (aa5,adder2) (aa6,adder2)
Proc 14 => (aa7,adder2)
```

```
Processor: adder2       -- 14
Processor: mult         -- 8

Number of buffers: 56
```

B.4 Example

The FIR filter (see the graph below) is a simple device containing adders and multipliers.

The following listing describes the FIR filter for pipe.

```
Graph: FIR_FILTER

Input: in1, in2, in3, in4, in5, in6, in7, in8
Output: out

Processor adder1 delay: 2 input: 1
Processor adder2 delay: 2 input: 2
Processor mult delay: 5 input: 1

m1 mult (adder1 (in1))
m2 mult (adder1 (in2))
m3 mult (adder1 (in3))
m4 mult (adder1 (in4))
m5 mult (adder1 (in5))
m6 mult (adder1 (in6))
m7 mult (adder1 (in7))
m8 mult (adder1 (in8))

aa1 adder2 (m1, m2)
aa2 adder2 (aa1, m3)
aa3 adder2 (aa2, m4)
aa4 adder2 (aa3, m5)
aa5 adder2 (aa4, m6)
aa6 adder2 (aa5, m7)
aa7 adder2 (aa6, m8)

out aa7
```

B.5 Installation

PIPE is distributed in C++ source. To compile, you will need the followings:

- A UNIX or UNIX like operating system (*PIPE* was compiled under MS-DOS, but is not guaranteed to work because the awkward memory management scheme of this system. It would probably mean little trouble to compile it under OS/2 or Windows NT). *PIPE* is verified to work under SunOS 4.1, HP-UX 8, HP-UX 9 and NetBSD-1.0.

- A C++ compiler. During development the Free Software Foundation's g++ compiler (versions 2.5.4 and 2.7.0) was used. (This compiler is available on the Internet from `prep.ai.mit.edu` via anonymous ftp.).
- Yacc or equivalent compiler-compiler. The precompiled grammar is provided in the file `gram.cc`. If you do not make changes in the grammar file, it is possible to install *PIPE* without yacc.

First unpack the compressed tar archive using the following command:

```
zcat pipe.tar.Z | tar xvf -
```

This should create a directory named pipe. Go to this directory. There is a configuration script, run it:

```
./configure
```

If necessary, edit the file `conf.h`, it contains some values that you might wish to change.

Start compilation:

```
make
```

After a while an executable named pipe should appear. Move this file where other users can access it.

PIPE was written by having portability a goal. Due to some incompatibility between the different UNIX systems, you may have to change the source. These changes should not be difficult.

Hardware-Software Codesign in a High-Level Synthesis Environment

Author/Designer	Tamás VISEGRÁDY
Supervisor	Péter ARATÓ, Andrzej RUCINSKI
Identification Code	DISS-DAL-HSCD-MAIN
Project Name	Hardware-Software Codesign (Dissertation)
Date Created	May 11, 1999
Revision #	1.1
Electronic Storage	**http://www.ece.unh.edu/links/tlv/main().html**
Physical Storage	*(identical)*

IMAGE EVALUATION
TEST TARGET (QA-3)