
Platform Based Reconfigurable Architecture Exploration via Boolean Constraints
by Douglas Densmore

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley,
in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:

Alberto Sangiovanni-Vincentelli
Research Advisor

Date

* * * * *

John Wawrzynek
Second Reader

Date

Platform Based Reconfigurable Architecture Exploration via Boolean Constraints

by

Douglas Densmore

Bachelor of Science in Engineering (Univ. of Michigan, Ann Arbor), 2001

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Electrical Engineering

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Alberto Sangiovanni-Vincentelli, Chair

John Wawrzynek

Spring 2004

Platform Based Reconfigurable Architecture Exploration via Boolean Constraints

Copyright 2004
by
Douglas Densmore
Version 4.0

Abstract

Platform Based Reconfigurable Architecture Exploration via Boolean Constraints

by

Douglas Densmore

Master of Science in Electrical Engineering

University of California, Berkeley

Alberto Sangiovanni-Vincentelli, Chair

Reconfigurable Hardware by its very nature requires that the topology of the hardware be set to a particular configuration given a particular application. With the introduction of *programmable system-on-a-chip* devices containing reconfigurable analog and digital components, comes the need to introduce a methodology to efficiently port applications to these devices. This approach should favor *reuse* and provide the necessary *programming abstractions* so that the platform features are appropriately exported up to the designer. This paper presents the background of reconfigurable architectures and places them in a *platform based design* approach to this problem culminating with a set of tools and corresponding framework for Cypress Semiconductor's Programmable System-on-a-Chip (PSoC). This toolset is dubbed "CAPE" and is an entire framework for exploring configurations on the PSoC. These tools make use of a Boolean constraint formulation based on CNF SAT and test this methodology successfully on two application areas.

Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

To Carl Robinette
“Spend it wisely”

Acknowledgments

I would first like to thank my advisor Alberto Sangiovanni-Vincentelli without whom my research at Berkeley would not be possible. His support and encouragement has been tremendously positive and he has provided a very strong example of hard work and dedication. He extended an invitation to me early on in my graduate career to do research in his group and I will be forever thankful.

In addition, I would like to thank my fellow graduate students for providing support during the completion of this project. In particular, Will Plishker, Donald Chai, Alessandro Pinto, Abhijit Davare, and Trevor Meyerowitz. Naturally, I would also like to thank all else whom I interacted with positively both in Alberto's group and the DOP center. I learned a great deal about research and academics due to the great environment and support I received here. There is no way to list all who have aided me thus far but know that you are appreciated.

Researchers that have provided opportunities to me throughout my academic career are Prof. John Hayes, Gordy Carichner, John Moondanos, Felice Balarin, and Yoshi Watanabe. All of the staff and those affiliated with Berkeley's Cadence Design Systems lab also were invaluable to me. My experience with Donald Sawdai and U of M's UROP program was also invaluable.

Cypress Semiconductor was very supportive and I would like to acknowledge Sanjay Rekhi, Michael LaBouff, Barbara Schremp, Dennis Seguine, Sri Purasai, and Warren Synder. They always were eager to supply both equipment and support. I would not have begun this project if it were not for their assistance.

I also need to thank Iyibo Jack who worked on this project with me during the summer of 2003. His "grunt work" was key to getting this project running. I want to wish him the best of luck in his future endeavors.

Naturally, I would like to thank my second reader Prof. John Wawrzynek. Your effort and time concerning this report has not gone unnoticed and I am indebted to you.

Friends such as Dale Winling, Neel Varde, Chris Burke, Jake Montgomery, Ryan Owen, Steve Berke, Nils Hernandez, Moses Morales, and Pat Collins helped make my life outside of school refreshing and supportive. They helped to make my free time something to look forward to.

To my family I would also like to show appreciation. In particular my mother, Barbara, and my father, Leroy deserve special praise. Their unique perspectives on life and guidance helped to make me a complete person. They always made me proud of who I am and helped me to see what I could become. My siblings, Matt, Diana, Luke, and Kate each have been tremendously influential in my life and made me realize that a family is far more important than academics. My whole extended family has also been tremendously supportive and I thank them for their love and wisdom.

Finally I would like to thank Megan Schatz. Without her support and encouragement I not only would not be in graduate school but I would also be much less of a person. She inspires and motivates me to not only be the best researcher and engineer that I can be, but more importantly the best person I can be. I wish her the best and the possibility of true happiness. I am forever in her debt and will always be in awe of her passion.

God bless each of you!

Contents

List of Figures	7
List of Tables	9
1 Introduction	10
1.1 Motivation	10
1.2 Reconfigurable Architectures	11
1.2.1 Reconfigurable Architecture Density Advantage	12
1.2.2 Reconfigurable Architecture Classifications	12
1.2.3 Reconfigurable Challenges	15
1.3 Organization of Report	16
2 Cypress Semiconductor’s Programmable System on a Chip (PSoC)	18
2.1 Cypress and the GSRC	18
2.2 PSoC Architecture	19
2.2.1 M8C Microcontroller	20
2.2.2 Digital Blocks	21
2.2.3 Analog Blocks	22
2.2.4 User Modules	23
2.3 Integrated Development Environment	25
2.4 Third Party PSoC Tools	26
2.5 Configuration Exploration to Improve Performance	26
3 Platform Based Design (PBD)	29
3.1 Constraint Propagation	29
3.2 Performance Estimation	30
3.3 Successive Platform Refinement	30
3.4 PSoC and PBD	30
3.4.1 Configuration as Architecture Instance	31
4 Boolean Constraints	32
4.1 Preliminaries	32
4.2 Conjunctive Normal Form (CNF)	33
4.2.1 CNF Satisfiability	34
4.2.2 CNF SAT and PSoC	34
5 Project Overview	36
5.1 Previous Work	36
5.1.1 HySAM	37
5.1.2 DEFACTO	38

5.2	CAPE	38
5.2.1	CAPE GUI	40
6	Application Space	42
6.1	CAPE: ASL	42
6.1.1	Parsing ASL	42
7	Constraint Space	45
7.1	CAPE: ACT	45
7.1.1	Raw Resource	46
7.1.2	Topology	48
7.1.3	Performance	49
7.1.4	Scheduling	50
8	Platform Space	52
8.1	CAPE: PAT	52
9	Estimation Space	54
9.1	CAPE: PET	54
9.2	CAPE: ACE	55
10	Case Studies	57
10.1	μ -Law Companding	58
10.1.1	ASL description	58
10.1.2	PET Results	59
10.1.3	Comparison between reference configuration	61
10.1.4	General Conclusions	62
10.2	Pulse Coded Modulation	62
10.2.1	ASL description	63
10.2.2	PET Results	63
10.2.3	Comparison between naive configuration	65
10.2.4	General Conclusions	65
11	Conclusions	67
11.1	Theoretical Conclusions	67
11.1.1	General Limitations	67
11.1.2	Horn Clauses	67
11.1.3	Constraint Types and Granularity	68
11.2	Tool Development Conclusions	68
11.2.1	Basic PAT	68
11.2.2	Basic PET	68
11.3	Experimental Results Conclusions	68
11.3.1	Execution Time	68
11.3.2	Clause Generation	68
11.3.3	Constraints Mixing and Valid Configurations	69
11.3.4	SAT Solver Determinism	69
11.3.5	Results	69
11.3.6	It Works!	69

12 Future Work	70
12.1 Tool Work	70
12.1.1 Completion of the CAPE Toolset	70
12.1.2 CAPE Extensions	71
12.1.3 ACE Investigation	71
12.1.4 Testing	71
12.2 Theory Work	71
12.2.1 Literal Encoding	71
12.2.2 Generalizing to Reconfigurable Computing	71
13 Appendix	72
Bibliography	76

List of Figures

1.1	Merging of Design Concepts	11
1.2	μ Code classification for CCMs	15
1.3	Temporal Reconfiguration Processes	17
2.1	PSoC Architecture	19
2.2	M8C Address Space	21
2.3	PSoC Digital Blocks	22
2.4	PSoC Analog Blocks Topology	23
2.5	Analog Switched Cap Type A Block	24
2.6	Analog Switched Cap Type B Block [29]	24
2.7	Analog Continuous Time Block	24
2.8	Design Subsystems and Development Flow	26
2.9	Interacting Configurations	27
3.1	Platform Based Design Framework	30
3.2	PBD with PSoC	30
3.3	Platform Based Design Discipline	31
5.1	CAPE in Platform Based Design	39
5.2	CAPE GUI	40
6.1	ASL Example Code	43
6.2	Data Structures to Hold Parsed ASL	44
7.1	Peripheral Library Structures	46
7.2	Creating Shared Location Constraints	47
7.3	Creating Limited Constraints	48
7.4	Creating Topological Constraints	48
7.5	Creating Relaxed Topological Constraints	49
7.6	Creating Performance Type 1 Constraints	50
7.7	Creating Performance Type 2 Constraints	50
7.8	Scheduling Constraint Formulation	51
8.1	PAT in the CAPE Flow	53
9.1	Sample M8C assembly	56
10.1	μ -Law Compressor	58
10.2	μ -Law Expander	58
10.3	Abstract ASL for μ -Law Companding	59
10.4	Refined ASL for μ -Law Companding	59

10.5 Abstract ASL for PCM	63
10.6 Refined ASL for PCM	64

List of Tables

1.1	Characteristics of Reconfigurable Logic	11
1.2	Architecture Technology Classification	12
1.3	Example Reconfigurable Architecture Classification	13
1.4	Horizontal/Vertical Axis Classification Example [37]	13
1.5	Reconfigurable Platform Offerings [37]	14
1.6	CCM Classification	16
2.1	Available PSoC Parts	20
2.2	PSoC Device Classification	20
2.3	M8C Registers	20
2.4	M8C Addressing Modes	21
2.5	Digital Block Configuration Registers	22
2.6	Analog Block Configuration Registers	23
2.7	Sample User Modules Categories	25
2.8	Sample PSoC User Modules	25
10.1	μ -Law Best and SAT Analysis	60
10.2	μ -Law Performance Estimation Comparison	60
10.3	Abstract μ -Law Structural Analysis	61
10.4	Refined μ -Law Structural Analysis	61
10.5	Abstract vs. Refined Comparison for μ -Law	62
10.6	Reference μ -Law Configuration	62
10.7	PCM Best and SAT Analysis	64
10.8	PCM Performance Estimation Comparison	65
10.9	Abstract PCM Structural Analysis	65
10.10	Refined PCM Structural Analysis	66
10.11	PCM Abstract vs. Refined Comparison	66
10.12	Naive PCM Configuration	66

Chapter 1

Introduction

The primary definition of configuration is: *relative arrangement of parts or elements*. This is often followed by a secondary definition: *functional arrangement* [14]. Both of these definitions are extremely relevant when talking about configurable computational hardware. In particular, it is the combination of both the words *relative* and *functional* that are the keys to understanding that a configuration is both in the context (relative) of a set of building blocks and that the arrangement is constrained to produce a particular behavior (functional). A configuration therefore is not some random collection of elements but rather a collection assembled to achieve some end. Naturally, how a configuration is created and with what methodology should be important to anyone studying such a phenomenon.

It is with this notion of configuration established that the addition of the prefix, “*re*” meaning again [15], indicates the repeated application of a configuration. Having this ability or the property of *reconfigurability*, would imply that not only are the elements composed in a functionally relative manner, but also that this process can be repeated. If one were to apply this to the notion of computational computer hardware, *Reconfigurable Hardware* would be the result. This hardware is often referred to as a particular *Architecture* which is intended to denote its classification and characteristics. It is how to determine and quantitatively analyze these *Reconfigurable Architecture* configurations that this investigation looks to address. In particular, how to exploit the ability to create several different configurations using the same architecture which realize the same overall functionality. Ultimately a toolset will be developed which allows a user to specify their high level functional needs and this representation will be transformed such that various configuration instances realizing this functionality can be explored and evaluated.

What follows is the motivation behind the development of reconfigurable architectures along with several examples of how one might attempt to classify these devices.

1.1 Motivation

Today’s computer hardware design environment is like no other in history. Due to a number of economic conditions, technological advances, and the continuation of Moore’s Law, there have arisen four competing forces which affect designs. These are *Time to Market Pressures*, *Deep Sub Micron (DSM) effects*, *Heterogeneity*, and *Complexity*¹. Each of these present unique dilemmas that must be faced in order for a company to produce products in a financially and technologically competitive manner. While DSM effects are an important topic, they will not be explicitly addressed below, but are mentioned as a contributor for completeness. Also note that there are typically many other factors influencing electronic design which are captured in the International Technology Roadmap for Semiconductors (ITRS) [17]. However, the three mentioned previously will be briefly commented upon.

Time to Market Pressures are due to the large number of companies currently competing in the design environment and the consumer’s expectation of when and how products will be released. This pressure requires that not only are designs done more quickly but concurrently one must be planning future releases. In order to combat this, design methodologies should favor **reuse, flexibility, and efficiency** as their defining characteristics.

¹Kurt Keutzer at the University of California, Berkeley calls this the “Quadruple Whammy”

Heterogeneity is present in today’s designs due to the applications which are being developed. Often times various components are integrated on a single die or composed on printed circuit boards (PCBs). How to ensure that the communication between these devices works correctly and to ensure that the interaction produces the desired behavior is yet another complication for the designers. In order to combat this, design methodologies should favor **separation of communication, computation, and coordination**. This is know as a *separation or orthogonalization of concerns*. This is also often used in *Communication Based Design* [1]. This concept will be addressed in Chapter 3.

Finally, *Complexity* manifests itself in today’s applications at the functional, implementation, and verification phases of application development and therefore implicitly can lead to correspondingly more complex hardware on which the application runs. This complexity makes all areas of the design cycle more difficult from specification to verification. In order to battle complexity, **abstraction and modularity** should be employed.

This report will culminate in the introduction of a toolset designed to combat this hostile design environment. To do so it utilizes both Reconfigurable Architectures and Platform Based Design. Reconfigurable Architectures address *reuse, flexibility, and efficiency*. Platform Based Design addresses *separation of concerns*. And finally the combination of the two into a cohesive toolset includes the concepts of *abstraction and modularity*. Figure 1.1 demonstrates this merging and gives examples in each area.

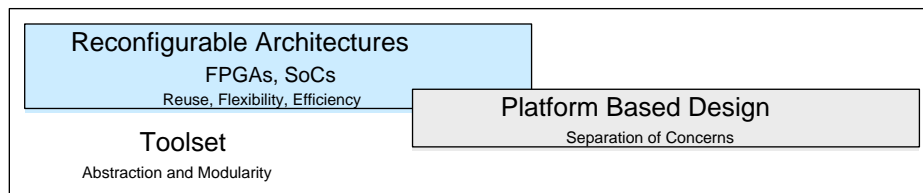


Figure 1.1: Merging of Design Concepts

1.2 Reconfigurable Architectures

Reconfigurable Architectures look to exploit certain characteristics in order to adapt to a variety of application areas. The notion being that their adaptive ability provides an advantage over their “static” counterparts. One reconfigurable hardware component can potentially replace many standard static hardware components. This naturally is not only a cost savings but also inherently a reuse issue. No longer must designers take time designing and learning about new parts. In [6] they give the following set of characterizations of reconfigurable logic as shown in Table 1.1. These are four ways in which reconfigurable architectures can gain performance over static architectures. However, there is no “one” reconfigurable architecture but rather a rich classification of architectures each with their own issues and benefits. It is these classifications which lend devices to specific applications and programming models. What follows is an exploration of these classifications. Understanding these classifications provides insight into the various ways that they can be programmed and ultimately used. More immediately however, this will aid in demonstrating why the toolset described in this report was developed as it was.

Characteristic	Description
Spatial Computation	Data processed by spatially distributing the computations
Configurable Datapath	Functionality and interconnection network of computational units is flexible
Distributed Control	Units process data based on local control
Distributed Resources	The required resources for computation are distributed throughout the device

Table 1.1: Characteristics of Reconfigurable Logic

1.2.1 Reconfigurable Architecture Density Advantage

A notable area in which to examine the motivation behind the use of reconfigurable architectures, is that of the computational density capabilities of such devices. Computational density is a good way to measure the effectiveness of die usage. This directly translates into needed die area which impacts yield and $\frac{\text{dies}}{\text{wafer}}$ which ultimately impacts profitability. Due to the characteristics mentioned in Table 1.1, particularly spatial computation, reconfigurable architectures are often able to provide significant computational density advantages over traditional computing devices (General Purpose Processors, DSPs, etc). DeHon in [11] provides an overview of this issue and draws several conclusions. First of these is that in general computational density is undermined by two factors: computational units *lack of use* and *overgenerality*. Both of these factors characterize the units in a general purpose processor or DSP much more so than a reconfigurable device such as an FPGA. Secondly, [11] goes on to demonstrate that by simply exploiting spatial computation in an FPGA you can significantly increase the computational density compared to a computation done temporally in a traditional microprocessor. FPGAs for example can increase computational density by reducing the instruction overhead as compared to a traditional computing device as well as controlling operations at the bit-level.

Computational density can be measured by $\frac{\text{BitOperations}}{\lambda^2}$ for example where λ is a unit of measure on the device die. This indicates the need for bit-level control as well as spatial computation. Especially appealing for reconfigurable computing is that as Moore's law continues to hold, the increased silicon area in FPGAs only increases its computational density. Where traditional microprocessors have to find creative ways to use silicon efficiently, FPGAs need to only continue the duplication of Configurable Logic Blocks (CLBs) and interconnect.

Since the density of advantage of reconfigurable computing devices is one of the main motivations behind their use, it is important that configurations be chosen intelligently to exploit this aspect. This is all the more reason to investigate the creation and evaluation of configurations for these devices.

1.2.2 Reconfigurable Architecture Classifications

In general the semiconductor technology or component organization that is used to make the device is a simple way to organize a device in terms of other device offerings. This leads to a fairly coarse classification but it is useful nonetheless. This is very true for reconfigurable architectures. This provides details for how, when, and to what extent these can be programmed. Table 1.2 describes this initial breakdown. Aspects of this can be thought of hierarchically in that some devices inherit attributes of the other and there is at times a "foggier" delineation between categories than one might like.

Device	Description
Programmable Logic Device (PLD)	PROMS, PLAs <i>Examples:</i> Flash Memory Devices from Intel [25]
Field Programmable Gate Array (FPGA)	Contains uncommitted configurable logic blocks (CLBs) <i>Examples:</i> Altera Cyclone FPGA [2]
Field Programmable Analog Array (FPAA)	Contains uncommitted configurable analog blocks (CABs) <i>Examples:</i> Anadigm AN10E40 [3]
System on a Chip (SOC)	Static and reconfigurable components at function unit level <i>Examples:</i> Cypress PSoC [31]
Hybrid Architectures	Static and reconfigurable components at function and bit-level <i>Examples:</i> Xilinx Virtex II Pro [40]

Table 1.2: Architecture Technology Classification

While this classification does show the diversity, it may be too coarse when really looking to examine how one can program these devices and how to best exploit their potential. In practice today, many designers are more interested in the programming model [27] (what is observable/controllable to programmer/developer) than they are the technology behind the device. It is difficult with this initial classification to answer the basic question "Why should I choose one over the other?".

In [6] a classification based on *Granularity, Host Coupling, Reconfiguration Methodology, and Memory Organization* is proposed. Explanations and examples are shown in Table 1.3.

Classification	Description
Granularity	Size of the smallest reconfigurable functional unit addressed by mapping tools Tradeoff between flexibility and performance overhead <i>Examples:</i> CLB, ADC, ISA (bit level, function unit, program control)
Host Coupling	Type of coupling to host processor Loose System Level/Loose Chip Level/Tight Chip Level <i>Examples:</i> Through I/O (SPLASH); Direct communication (PRISM); Same chip (GARP, Chameleon)
Reconfiguration Methodology	How the device is programmed <i>Examples:</i> bit stream (serial, parallel); dynamic; partial
Memory Organization	How computations access memory <i>Examples:</i> large blocks vs. distributed

Table 1.3: Example Reconfigurable Architecture Classification

Table 1.3 reflects a more programming model view of the devices. In addition, one can see more clearly what will constitute a “configuration” for devices classified in this manner.

In [37] they define a design space of *three orthogonal axes*. A vertical axis expresses the level of abstraction, a horizontal axis expresses the reconfigurable feature diversity, and a time axis represents the timing relationship of configuration to processing. Table 1.4 shows how the design elements (horizontal axis) can manifest themselves at different design levels (vertical axis).

<i>Design Levels (Vertical Axis)</i>	<i>Design Elements</i>	(Horizontal Axis)	
	Communication	Storage	Processing
Implementation	Switches/Muxes	RAM Org.	CLB/IP Block
Microarchitecture	Crossbar/Bus	Reg. File Size Cache Architecture	Execution Unit Type Interpreter Levels
Instruction Set Architecture	Address Size	Reg. Set	Custom Instr.
Process Architecture Systems Architecture	Intercon. Network	Buffer Size	Number/Types of tasks

Table 1.4: Horizontal/Vertical Axis Classification Example [37]

In addition to providing the definition of three axis, [37] also provides a list of both academic and commercial platforms both which are configurable (fixed at semiconductor processing) and reconfigurable. Table 1.5 reconstructs this with the addition of the Cypress PSOC (which is integral and the focus of this investigation) and the exclusion of the configurable only platforms. This is included to demonstrate a sample of the many product possibilities available at the time of this report. As the original source mentions, this is not intended to be exhaustive. Notice that the how these platforms fit on the vertical axis ² is also noted along with where more information on these devices can be found.

The vertical characterization is roughly defined that *ISA* indicates that the actual programmers view of the device can change. This could be the number of registers or instruction set. *M* indicates that function unit organization can change. This could be the number of function units, their connections to each other, or their performance characteristics. *I* indicates that the physical implementation can change so that for example one configuration might be more power efficient while another might be more performance oriented.

Table 1.5 and [37] are naturally not the only views on how to classify these machines. In [38], reconfigurable computing coupled with a *General Purpose Processor (GPP)* create a device referred to instead as *Custom Computing*

²(I)Implementation, (ISA) Instruction Set Architecture, (M) Microarchitecture, (P)Process Architecture

Platform (Product/Company)	Vertical Axis	Reference
	<i>Commercial</i>	
Excalibur, Altera	I, M	http://www.altera.com
MECA 41, PMC-Sierra	P	http://www.pmc-sierra.com
Morphics	P	http://www.morphics.com
E7/A5, Trisend	ISA, P	http://www.trisend.com
FPSLIC, Atmel	I, P	http://www.atmel.com
CS2112, Chameleon Syst.	M, P	http://www.chameleonsystems.com
PSoC, Cypress Micro	I, M	http://www.cypressmicro.com
	<i>Academic</i>	
GARP, UCB	I, ISA, P	http://brass.cs.berkeley.edu
PipeRench, CMU	M	http://www.ece.cmu.edu/research/piperench
MorphoSys, UCI	I, ISA, P	http://www.eng.uci.edu/morphosys
SPS, UCLA	I, M	http://www.cs.ucla.edu/elib/reconfigurable
C-RISP, KULeuven	ISA	http://www.acca.be

Table 1.5: Reconfigurable Platform Offerings [37]

Machines (CCMs). The idea is to exploit the GPP flexibility and the reconfigurable capability to achieve medium performance for a large class of applications. This architecture would fall into the Hybrid classification in Table 1.2. This work divides these CCMs into two criteria:

- **Vertical or Horizontal Microcoded machines**

A microinstruction which controls multiple resources in one cycle is horizontal. This is a very spatial organization of resources. In the extreme case all resources of the datapath are controlled. A microinstruction which controls a single resource is vertical. See Figure 1.2 for a visual of this approach. The tradeoff between the two types of machines is between a variety of factors including instruction decode/control overhead, power consumption, and data path control overhead (hazards, dependencies, etc).

- **Machines with or without a SET instruction**

This instruction initiates the reconfiguration of the raw hardware. This can be done at runtime and is accessible by the programmer. This is a very good method to also place a device on the time axis as defined in this section.

Notice that in Figure 1.2 diagram (a) shows the horizontal microarchitecture while (b) is an example of the vertical approach. These distinctions demonstrate issues such as scheduling and concurrent execution which are prevalent in today's microarchitectures.

Table 1.6³ shows this classification on a number of commercial and academic devices. Note how this table compares to Table 1.5 as denoted by *italicized bold* devices which appear in both tables. The overlap is indicative of the diversity of these devices and the difficulty one has in creating a definitive taxonomy.

A large distinction between devices can be made as to when you can configure them. This is partially revealed by the presence of a SET instruction. This is discussed in the next section. It is particularly relevant since at what point during the execution of the application the device configures changes very much how one thinks of the device applications and the scheduling and performance issues associated with them.

Runtime vs. Compile Time

Yet another way to classify architectures as brought up by [39] is runtime vs. compile time devices. Compile time devices have predetermined configurations that are decided at compile time that remain until the completion of

³Please see [38] for references to each particular CCM

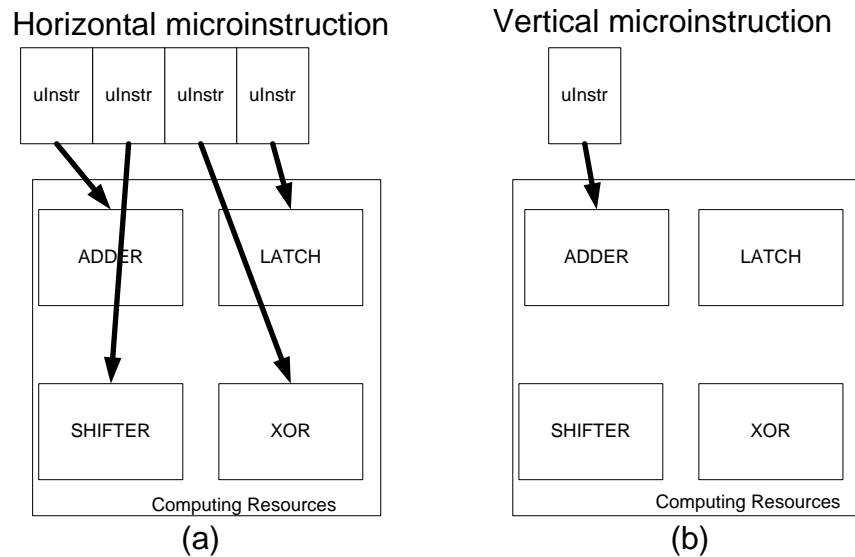


Figure 1.2: μ Code classification for CCMs

a particular task. Runtime reconfiguration involves the ability to repeatedly program a system with many smaller functions to complete a particular application and therefore the hardware adapts during runtime “swapping” in and out these functions to accomplish this. This is often referred to as “static” vs. “dynamic” reconfiguration as well [6]. In addition, this is often referred to as the “binding time” of the device.

This iteration between configuration and execution naturally will depend on the application and the device. How this reconfiguration is scheduled can have a dramatic effect on the performance. For example in [20] the authors introduce the notion of partially reconfigured systems. They differentiate between runtime reconfiguration (RTR) and compile time reconfiguration (CTR). They claim that in RTR systems additional performance can be gained if only the **necessary** changes are made between run time reconfigurations. The performance savings comes in two areas: *configuration time reduction and retention of intermediate values*. The retention of intermediate values refers to the fact that between configurations the intelligent preservation of data needed for further computation can speed up execution. Both of these are important aspects to consider in a reconfigurable device programming environment. A tool should include metrics which takes these factors into account.

Figure 1.3 shows these interacting issues.

This discussion shows that reconfigurable architectures can be classified by **technology, device characteristics (abstraction, diversity, timing), or microarchitecture**. Notice that these all demonstrate potential areas for abstraction. Each of these provides various divisions and demonstrates the need to carefully examine what aspect you are attempting to exploit in order to determine with device may perform better in a given application. The key insight is that *in order to program a device effectively, you must understand what classification it falls into in order to exploit its benefits fully*.

For more information on how reconfigurable architectures can be classified and their applications see conferences such as [21].

1.2.3 Reconfigurable Challenges

There are many interesting challenges when discussing reconfigurable hardware. Naturally, tools should be made to address these challenges so that the real advantages of the architectures can be exploited. As you can see, these challenges each take a different view depending on how the reconfigurable systems is characterized as mentioned in the Chapter. In [6] it lists four key challenges:

Vertical μ code		Horizontal μ code	
Explicit SET	w/o SET	Explicit SET	w/o SET
PRISM	PRISC	CoMPARE	<i>PipeRench</i>
PRISMII/RASC	OneChip	Alippi's VLIW	
RISA'	ConCISe 7	RISA''	
RISA''	OneChip-98'	VEGA	
MIPS + REMARC	DISC	RaPiD	
<i>Garp</i>	Multiple-RISA	Colt	
OneChip-98''	Chimaera	rDPA	
URISC			
Nano-Processor			
Gilson's CCM			
CCSimP			
Xputer/rALU			

Table 1.6: CCM Classification

- Static vs. Dynamic Reconfiguration
 - When, how, and to what extent a device reconfigures itself during the execution of an application.
- Design Methodologies
 - How applications are partitioned, mapped, and designed for reconfigurable computing.
- Multi-dimensional Optimization
 - Looking to optimize several metrics both in combination and independently.
- Design Tools
 - Tools to automate, evaluate, and analyze reconfigurable systems.

Static vs. Dynamic Reconfiguration investigation requires that **scheduling configurations and constraints** are accounted for so that devices can take advantage of applications which the hardware can continuously adapt to the application. Design Methodologies investigation is a large area and this report proposes that **Platform Based Design** can be used in this design environment. Multi-dimensional optimization is at its heart a design space exploration process in which multiple metrics are examined. Finally, the introduction of design tools are exactly what this investigation looks to address. Aspects of this tool will also address multi-dimensional optimization. The previous sections outlined the characterization of various devices. It is imperative that one understand these characterizations in order to understand how each has its own space within these challenges. This report will focus on the PSoC from Cypress Semiconductor and use the way in which it is characterized in order to address these challenges.

I am not saying that this investigation eliminates the need for future work in this area but more concretely that it explicitly looks to focus on the core areas which are key design challenges for reconfigurable logic. In particular it looks at each area and applies a solution uniquely suited for the Cypress Programmable System on a Chip as classified using the methods described in this Chapter.

1.3 Organization of Report

Following this introduction and motivation the project will now begin to develop. Firstly, in Chapter 2 an introduction to Cypress Semiconductor's Programmable System on a Chip (PSoC) will be given. This platform is the focus of this investigation. Next, the concepts of Platform Based Design (PBD) will follow in Chapter 3. This

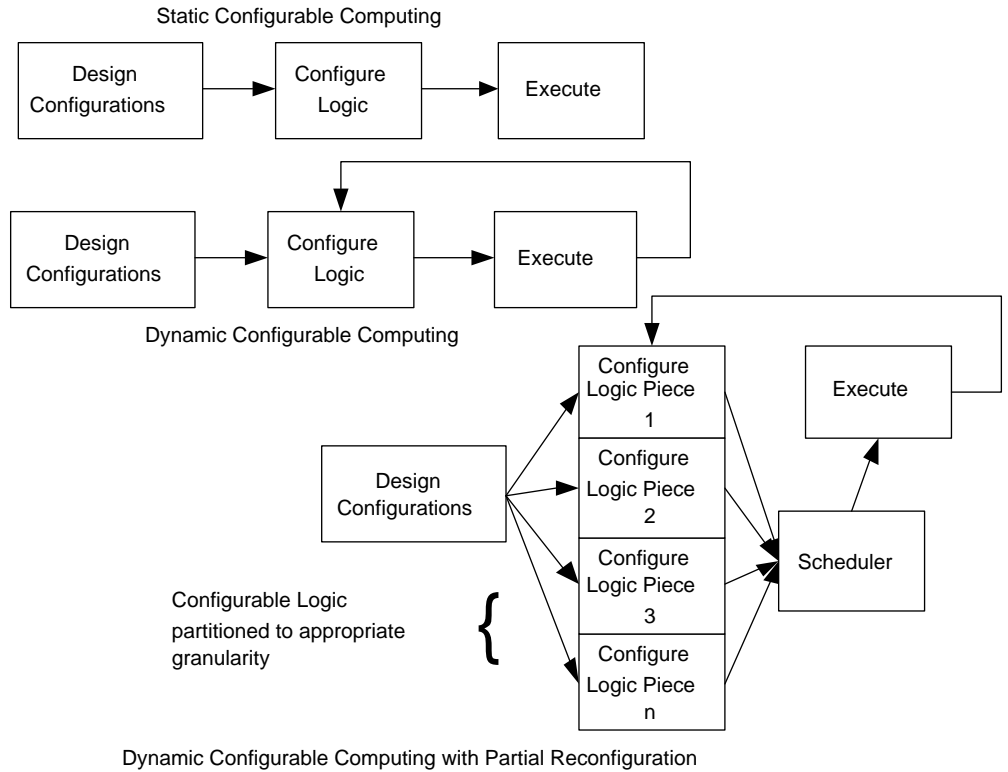


Figure 1.3: Temporal Reconfiguration Processes

provides the background regarding the overall methodology used in formulating the toolset flow. Boolean Constraint preliminaries will be covered in Chapter 4. This is the framework which provides the foundation for the “workhorse” of the toolset. These Chapters provide the specific background of the project. The actual project will begin with Chapter 5 which details the previous work done regarding reconfigurable architecture design space exploration. In addition this Chapter introduces the CAPE toolset. Chapter 6 will discuss the application space capture of PBD followed by the constraint space in Chapter 7, the platform space in Chapter 8, and the estimation space in Chapter 9. All of the material in this report is brought together by the case studies in Chapter 10 and finally the report ends with conclusions and future work in Chapters 11 and 12 respectively.

Chapter 2

Cypress Semiconductor's Programmable System on a Chip (PSoC)

Cypress Semiconductor's Programmable System on a Chip (PSoC) is a result of Cypress Semiconductor's establishment of its subsidiary, Cypress Microsystems, in the fourth quarter of 1999 [31]. Cypress is attempting to target embedded applications with a communications focus [29] with this acquisition. As a result of this effort, the PSoC was released November 13th, 2000. This is a unique entry into the reconfigurable market place due to its low cost (\$2-\$3.50¹) and the inclusion of analog reconfigurability. It is essentially a microcontroller core which can dynamically reconfigure itself to include various analog and digital peripherals. As mentioned in 1.2.2 it falls under the *SoC architecture* classification. Cypress provides the following company release regarding the target application space for the PSoC:

“As general purpose solutions, PSoC devices are targeted for implementation in embedded applications, including audio, wireless, handheld, data communications, Internet control, industrial, and consumer systems”
- Cypress Microsystems

The PSoC has been recognized in various publications and it was named “Innovation of the Year 2001” by EDN Magazine. The PSoC currently has 4 devices in its family and their characteristics are shown in table 2.1. It is highlighted since it is the focus of this investigation and this report's toolset is developed for it. It was chosen as the target device due to (1) its unique blend of analog and digital peripherals (2) its relatively straightforward architecture (3) its availability and Cypress' relationship with the Gigascale Systems Research Center (GSRC) and (4) most importantly it uses an IP block like granularity (classification) which is a key programming abstraction for this investigation. This last point is shown in its place in Table 1.5 and is more reason why this classification background was provided in Chapter 1.

2.1 Cypress and the GSRC

The relationship between Cypress and the GSRC dates back to at least Summer 2002. Sanjay Rekhi, a Cypress Engineer, was the acting liaison between the GSRC and the University of California Berkeley (UCB). He was interested in how UCB and Cypress could collaborate on various issues related to System Level Design. One of the goals of this collaboration was to examine how UCB's research efforts could be applied to industrial products. One of the themes of the GSRC is communication/component based design (manifested as Platform Based Design). This was a natural area in which to explore the possibilities of the PSoC device particularly due to the fact that it is toolset related. A proposal was made to attempt to see how the platform based design methodology could be applied

¹Reference <http://www.ebnonline.com>

to the PSoC and in January 2003 this project was started. Cypress provided the development kit which includes, an *In-Circuit-Emulator (ICE)* which provides debugging capabilities, a *Y-Programmer* to download configurations and code to device (supports DIP packages), a *POD, Pup, Foot* (Bar LED and DIP Foot, POD rev E), *Samples* (2 CY8C26443), and *Development Software* (Version 3.10). UCB in turn provided feedback on the results of the various projects carried out on the device. This report is the culmination of those efforts.

2.2 PSoC Architecture

The PSoC device is a combination of analog and digital reconfigurability along with an 8-bit Microcontroller and some dedicated peripherals. These peripherals include a watchdog and sleep timer, low voltage detection, and on-chip voltage reference. Figure 2.1 shows the overall layout of the device.

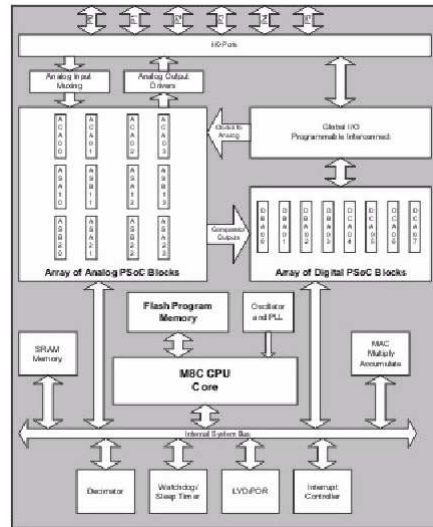


Figure 2.1: PSoC Architecture

Key to note is that this is a relatively simple device which in many ways resembles any low cost microcontroller. The arrays of the analog and digital blocks, along with the programmable interconnect, are the keys to programming the device. They are themselves fixed in the overall topology and interact with static peripherals via the M8C microcontroller.

As mentioned, there are 4 versions of the PSoC. Table 2.1 shows that the key differences are in I/O pins, packages, and program memory. All parts have the same number of analog and digital blocks and hence this key abstraction will scale for all devices and this investigation will apply to all current devices as well.

In order to put the PSoC in perspective when compared to the previous sections dealing with the classification methods it should be said that the following could classify the PSoC: **it is a SoC architecture with the granularity of User Modules**. It is run-time reconfigurable and supports partial reconfigurability. See Table 2.2 for the complete classification.

For the rest of this discussion the part referred to will be the **28 PDIP CY8C26443**. This is not only the part we have run experiments with but also has the maximum amounts of memory and reconfigurable blocks (only aspects visible to toolset in the future discussion). The next portion will discuss the **4 major areas** of the device: the *M8C microcontroller*, the *Digital Blocks*, the *Analog Blocks*, and the *User Modules*.

	CY8C25122	CY8C26233	CY8C26443	CY8C26643
Operation Frequency	93.7kHz-24Mhz	93.7kHz-24Mhz	93.7kHz-24Mhz	93.7kHz-24Mhz
Operating Voltage	3.0-5.25V	3.0-5.25V	3.0-5.25V	3.0-5.25V
Program Memory (KB)	4	8	16	16
Data Memory (Bytes)	256	256	256	256
Digital PSoC Blocks	8	8	8	8
Analog PSoC Blocks	12	12	12	12
I/O Pins	6	16	24	40/44
External SMP	No	Yes	Yes	Yes
Available Packages	8 PDIP	20 PDIP 20 SOIC 20 SSOP	28 PDIP 28 SOIC 28 SSOP	48 PDIP 48 SSOP 44 TQFP

Table 2.1: Available PSoC Parts

Classification	Type
Device Type	SoC Architecture
Granularity	Digital or Analog Blocks
Host Coupling	Same Chip; M8C μ Controller
Reconfiguration Methodology	Dynamic, Partial, Memory Mapped
Memory Organization	Large Blocks
Vertical Axis	ISA and Microarchitecture
SET instruction	No; requires series of memory operations

Table 2.2: PSoC Device Classification

2.2.1 M8C Microcontroller

The M8C Microcontroller is an 8-bit, 4 MIPS, Harvard Architecture Microcontroller. It allows for clock speeds up to 24 Mhz and as low as 937 kHz[30]. It is accumulator based and relies on a separate on-chip Multiply and Accumulate (MAC) unit.

The M8C has five internal registers which are displayed in Table 2.3.

Register	Purpose
Accumulator (A)	Holds the operand/result of arithmetic/logical operations
Index (X)	Used in various addressing modes; Useful for certain routines
Program Counter (PC)	16-bits for program execution
Stack Pointer (SP)	Used for stack operations in RAM
Flags (F)	Side effect of operations; Carry, Zero, Misc

Table 2.3: M8C Registers

All of the internal registers of the M8C are 8-bits except the program counter which is 16-bits. With the exception of the F register, the internal registers are not accessible via an explicit register address.

The M8C has three address spaces: *ROM*, *RAM*, and *Registers*. The ROM address space is accessed via its own address and data bus. The register space is used to configure the programmable blocks. The register space consists of two banks of 256 bytes each. These are primarily the registers used for reconfiguration. The RAM is broken into 256 byte pages. Figure 2.2 shows the address space.

Finally, the M8C has a total of seven instruction formats which use instruction lengths of one, two, and three bytes. There are a total of 256 instructions divided into 37 instruction types which support the following 10 addressing modes in table 2.4. Due to the statically scheduled, non-pipelined multicycle nature of the M8C, the cycles/instruction

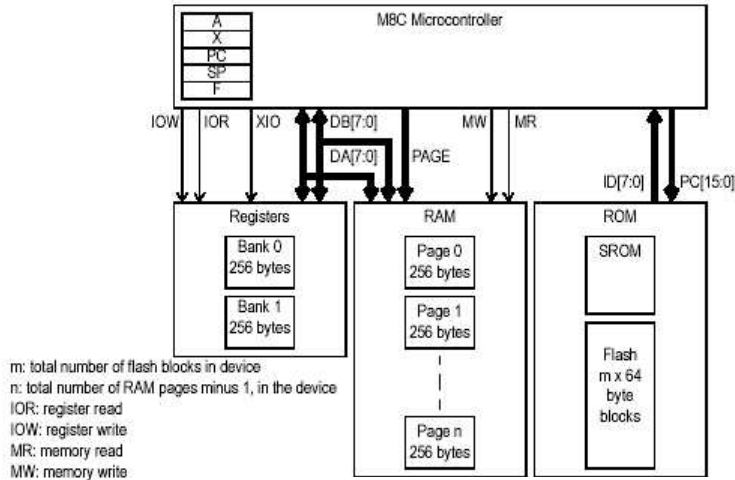


Figure 2.2: M8C Address Space

are static and are documented. *This will be useful later in performance estimation of various configurations.*

Source Immediate ADD A, 7	Source Direct ADD A, [7]	Source Indexed ADD A, [X+7]
Destination Direct ADD [7], A	Destination Indexed ADD[X+7], A	Destination Direct Source Immediate ADD [7], 7
Destination Indexed Source Immediate ADD [X+7], 7	Destination Direct Source Direct ADD [7], [7]	Source Indirect Post Increment MVI A, [8]
Destination Indirect Post Increment MVI [8], A		

Table 2.4: M8C Addressing Modes

For more information on the M8C microcontroller please refer to [30].

2.2.2 Digital Blocks

The first reconfigurable piece of logic is the “digital block”. The PSoC has eight digital blocks. A digital block is an 8-bit device. They are denoted as Digital Basic Type A (DBA) and Digital Communications Type A (DCA). The physical locations that they occupy are denoted similarly. Each of these digital blocks can be configured independently or used in combination. The type of block denotes what functionality that it can provide. The functionality that can be provided is: *Timer, Counter, Cyclic Redundancy Check (CRC), Pseudo Random Sequencer, and Deadband* on DBA and DCA blocks. In addition, *UART and Serial Peripheral Interface (SPI)* functions are available on the DCA blocks. Each of these is configured via its register space addressed by the M8C. The block is configured through the use of three types of configuration registers shown in Table 2.5.

In addition there are three data registers, **Data 0, Data 1, and Data2**. The function of these registers is dependent on the overall block functionality. Typically they may be used to store data or provide data.

Function Register	Select the block function and mode
Input Register	Select the data input and clock selection
Output Register	Select and enable the output

Table 2.5: Digital Block Configuration Registers

Finally, one control register, **Control 0** is also dependent on the block functionality and will adjust parameters unique to that user module.

The key point to bring out regarding the digital blocks are that they are full 8-bit devices which are connected in the topology shown in Figure 2.3 and that they are configured based on the values written into seven 8-bit registers which are in a register bank which can be accessed by the M8C microcontroller's register address space. Since they can be configured simply by writing values to the registers, they can be **dynamically reconfigured** at run-time so that the device can realize many different configurations. This leads to a notion of *configuration scheduling* which will be discussed later in this project.

Figure 2.3 shows the relationship between the digital blocks and the global input and output buses.

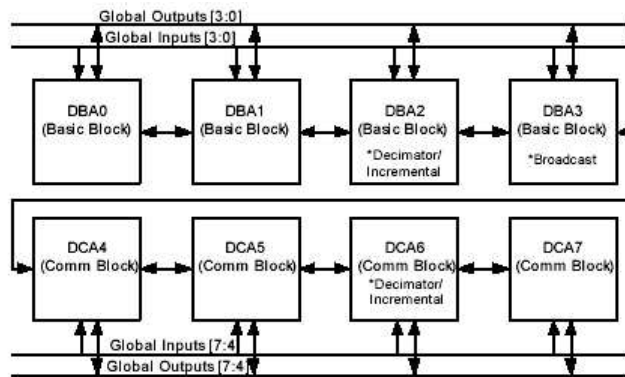


Figure 2.3: PSoC Digital Blocks

For more information see, [29]. Of note is what specific values cause what configurations and the process by which configuration registers are accessed.

2.2.3 Analog Blocks

Twelve analog PSoC blocks are available separately or combined with the digital PSoC Blocks. There are three types of analog PSoC Blocks: *Continuous Time Blocks (CT)*, *Type A and Type B Switched Capacitor (SC)* blocks. The CT blocks provide continuous time analog functions. SC blocks provide ADC and DAC analog functions.

The PSoC analog blocks are arranged as shown in Figure 2.4. Notice that there are three distinct outputs from each analog block. (1) The analog output bus (ABUS) which is an analog resource shared by all of the analog blocks in the column. (2) The comparator bus (CBUS) which is a digital resource that is shared by all the analog blocks in the column. (3) The output bus (OUT; plus GOUT and LOOUT in the CT blocks) which is an analog resource shared by all analog blocks in the column and connects to one of the analog output buffers to send a signal externally.

What is key to note is how the organization of the blocks dictates how they can interact. This interaction *constraint* will be key later in the discussion when configurations are created which adhere to these constraints.

Unlike the digital blocks, which are 8-bit peripherals in of themselves, the analog blocks are particular types of CT or SC blocks. This leads to a somewhat less coarse granularity. The actual circuit make-up of these types of blocks are shown in Figures 2.5, 2.6, and 2.7.

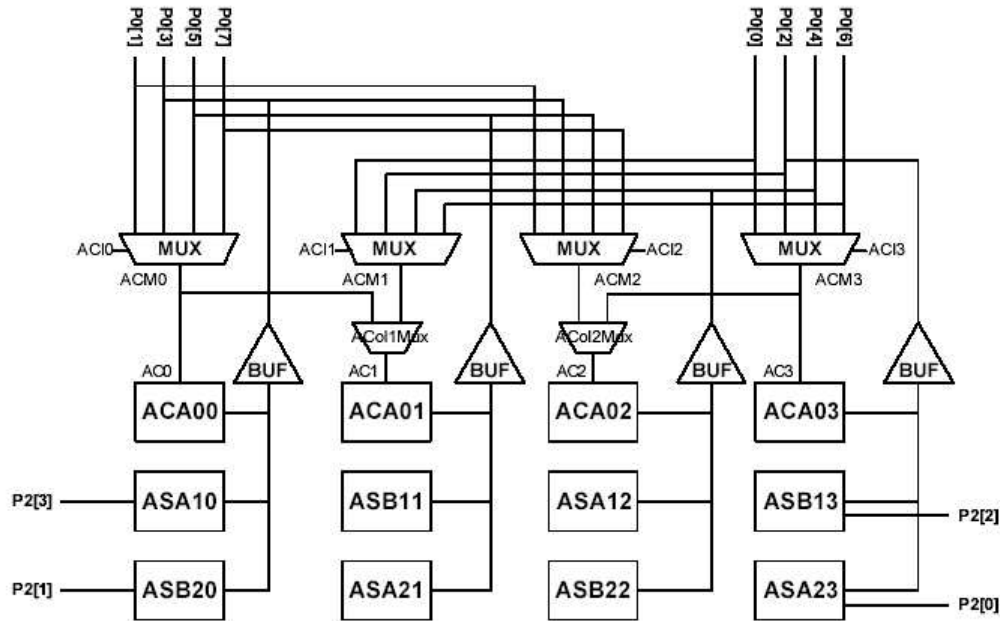


Figure 2.4: PSoC Analog Blocks Topology

Similarly to the digital blocks, there are registers that control the configuration of the analog blocks. These are shown in Table 2.2.3. Also, like the digital block registers, these can be changed at runtime, which will in turn provide a new set of analog configurations.

Register Type	Function
Analog Reference Control Register	Sets the analog reference for the whole device
Analog Column Clock Select Register	Determines which clock goes to which column
Analog Clock Select Register	Selects the source for ACLK0 and ACK1 signals
Analog Control Register	3 for CT, 4 for Sw
Analog Input Select Register	Muxing control
Analog Output Buffer Control Register	Sets output muxing
Analog Modulator Control Register	Select modulating signal for blocks

Table 2.6: Analog Block Configuration Registers

Again see [29] for more information on the analog blocks, specifics on register values, and potential user modules.

2.2.4 User Modules

The key programming abstraction of the PSoC is the *User Module*. It is not the individual blocks that the user manipulates but rather predefined user module. To be concise:

Definition 2.2.1 : A *User Module* is the collection of register settings which when loaded produces a specific functionality by utilizing analog blocks, digital blocks, or both. The module is parameterizable if certain register settings such as input or output are left to the user's discretion. A user module can consume the following resources, Digital

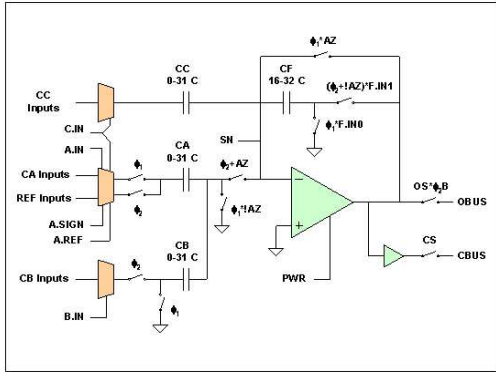


Figure 2.5: Analog Switched Cap Type A Block

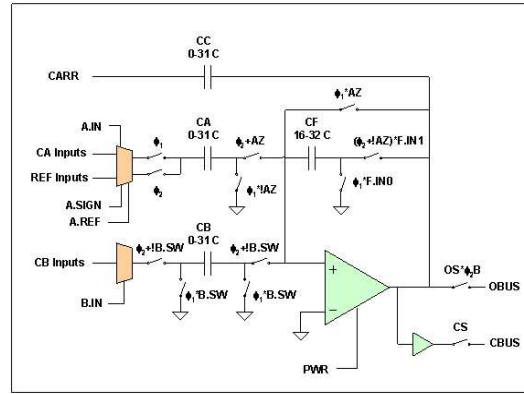


Figure 2.6: Analog Switched Cap Type B Block [29]

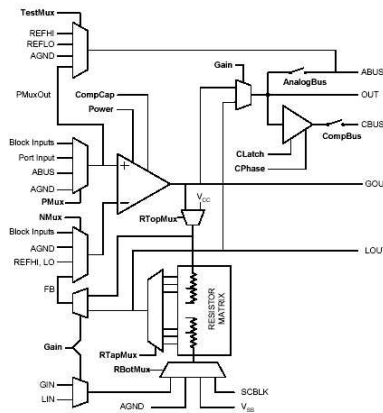


Figure 2.7: Analog Continuous Time Block

Blocks, Analog Blocks, RAM, and ROM. In addition, the user module has a fixed number of locations it can occupy on the device directly influenced by which reconfigurable blocks it uses.

Definition 2.2.2 : User Module and Configuration Relationships

Location Set (LS) = {CT, TypeA SC, TypeB SC, DBA, or DCA blocks}

Register Settings (RS) = {Bit level values in M8C register space}

Analog Blocks (AB) = {LS, RS}

Digital Blocks (DB) = {LS, RS}

User Modules (UM) = {AB, DB, RAM, ROM}

Configuration = {UM₁...UM_N}

User Module definitions are included with the Cypress Integrated Development Software and are updated with subsequent releases. Since a user module is essentially a register setting, one could create their own user modules by examining the block structures. However, the typical programming model is to use those provided by Cypress. Not only do they provide the configuration but also a datasheet and software API.

The peripherals are categorized in the following manner as shown in Table 2.7.

As mentioned, the user modules take up device resources. Table 2.8 shows a sampling of some user modules and how they utilize programmable blocks and how much memory they require.

ADCs	Amplifiers	Counters
DACs	Digital Comm	Filters
Misc Digital	MUXs	PWMs
Random Seq	Temperature	Timers

Table 2.7: Sample User Modules Categories

User Module	PSoC Blocks	Memory (Bytes)
12-bit ADC	2D, 1A	184 Flash 6 SRAM
Programmable Gain Amp	1A CT	32 Flash
8-bit Counter	1D	66 Flash
6-bit DAC	1A SwCap	47 Flash
16-bit CRC	2D	56 Flash
Two Pole Band Pass	2A SwCap	29 Flash
16-bit PWM	2D	115 Flash

Table 2.8: Sample PSoC User Modules

The user modules will be the foundation for the discussion to follow. How you place them, which ones you select, and how they interact, will determine a *configuration*. Naturally, how this is done is constrained by the device itself which creates a finite number of configurations. This can be further reduced by user input as will be described. However, before doing so we will examine the current programming environment and its shortcomings which should be addressed when creating tools for this device.

2.3 Integrated Development Environment

The PSoC development kit includes a Windows based development environment. This is used to assign user modules to the device, program the device (C or M8C assembly), and then debug the device (using the In-Circuit-Emulator (ICE)). Part (a) of Figure 2.8 shows the three main development areas. These are each important to the design of an application. However, notice that **there is no subsystem which automatically selects a configuration for the designer**. That must be done in the device editor subsystem manually. This investigation looks to improve on this area.

A design typically proceeds as shown in part (b) of Figure 2.8. The figure shows that the three subsystems each have their own purpose. In the Design Editor Subsystem there are three views: *User Module Selection View*, *User Module Placement View*, and *Pinout View*. Each of these has to do with setting the configuration registers (albeit in a user friendly graphical view). The Application Editor Subsystem is simply a project management system in which all application code can be edited and compiled. It also includes “automatically” the API for the included user modules. The Debugging Subsystem is where the design can be run and the code and device resources examined. This is very much like many hardware debuggers where the contents of M8C registers, Memory, and application Code can be traced through in the event that the device is not functioning as desired.

Figure 2.8 part (b) shows that the design progresses sequentially until the debugging subsystem at which point depending on the results of running the application, one might revisit the previous subsystems in order to fix the issue. Naturally these iterations should be kept at a minimum. *It is the **Device Editor Subsystem** iterations that hopefully, a configuration exploration could substantially reduce. More importantly it can determine configurations that could not easily be determined manually.* The tools in this report address **exactly these issues**.

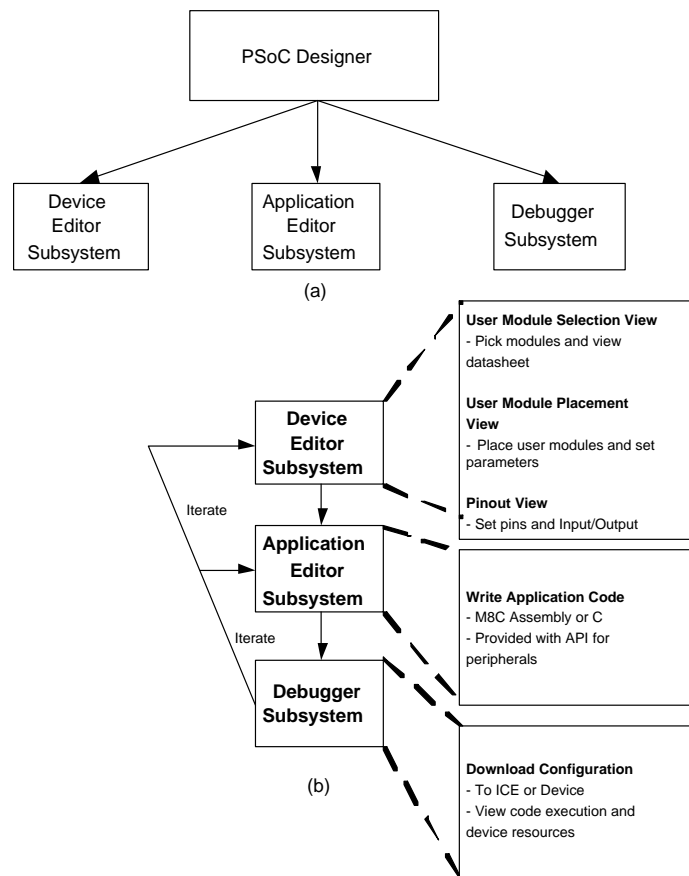


Figure 2.8: Design Subsystems and Development Flow

2.4 Third Party PSoC Tools

There is one distinct Third Party Software Package for the PSoC called *CodePalette* by Codetelligence [9]. This is a code generation package which creates support code for PSoC peripherals. This allows the designer to bypass writing low level assembly code and rather use higher level APIs to develop applications. It also allows for peripheral scheduling and power management. This tool **does not** help in the decision or development of a configuration instance and hence does not overlap with this investigation. It does show the need to raise the abstraction level of tools in order to deal with complexity and rapidly develop applications which is a similar philosophy to that in this report.

2.5 Configuration Exploration to Improve Performance

As mentioned previously, the selection of devices is manual in the Device Editor Subsystem. This requires that the user pick a configuration and then proceed through the rest of the flow. The configuration that is chosen can have an effect on the design in four areas: *Raw Resource*, *Topology*, *Performance*, *Scheduling*. These four metrics are how you can both **estimate and constrain** a configuration. Both of these two aspects are key in the exploration of configurations. See Figure 2.9 for a quick visualization of how the configuration space is partitioned depending on one or more of these concerns. Each of these are important areas to examine while deciding on a configuration. This is very difficult to do manually and ideally an automatic process would create and evaluate configurations and return the one suited for your criteria. *This is what the proposed toolset does!* In order to further demonstrate this concept

let's examine these four issues in more depth and provide examples of both the estimation and constraint possibilities within each.

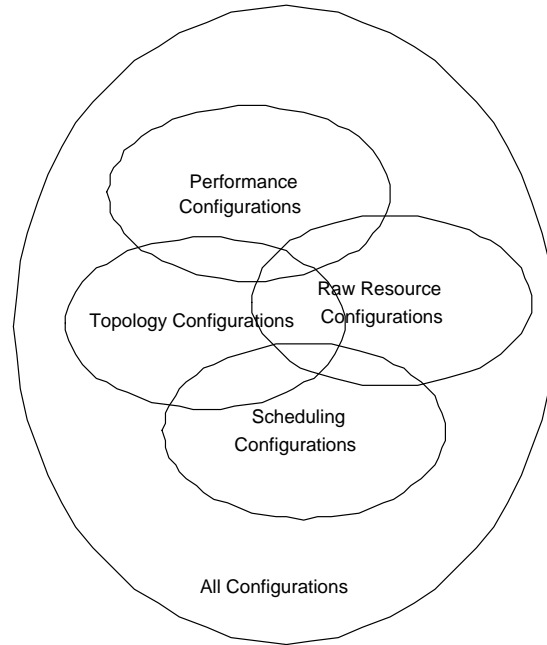


Figure 2.9: Interacting Configurations

Raw Resource

While developing a configuration, iterations may immediately result from the selection of user modules which cannot all occupy the same configuration at the same time or otherwise over utilize system resources. This would violate a *Raw Resource* constraint. For example:

$$PWM16 + Timer32 + Counter24 > 8DigitalBlocks \Rightarrow \text{(Raw Resource Constraint Violation)}$$

The main issue is that the relationship ² $\Sigma_{ReconfigBlocks} \geq \Sigma_{UserModule}ABPM + \Sigma_{UserModule}DBPM$ must be preserved. This is the **constraint** element of this aspect of a configuration. Naturally differing configurations will take up more Raw Resources. This is bounded by the constraint shown but within that constraint, there are many differing configurations naturally.

A raw resource **estimation** could be made quite easily by:

$\Sigma_{UserModule}ABPM + \Sigma_{UserModule}DBPM = TotalBlocksUsed$. A configuration could then be evaluated based on its resource usage. Depending on your application, this may lead to more desirable configurations.

Topology

A Raw Resource constraint, such as how many blocks something requires, can be caught by the existing PSoC software. However suppose you wanted to have more than 4 PWM16 devices using the same global output bus. This would not be possible due to the number of blocks consumed by the PWM16. The current software *cannot* currently catch this until you attempt to do it manually in the Module Placement View.

$$4PWM16 \rightarrow GlobalInputBus[0 : 3] \neq True \Rightarrow \text{(Topology Constraint Violation)}$$

²A(D)BPM - Analog(Digital) Blocks Per Module

It would not be until the next step in the flow (User Module Placement) where you would catch this. **This will cause an increased iteration.** Perhaps another configuration using PWM8 would be sufficient for your application. Since there was no specification as to the PWM bit width, this is a valid configuration.

$4PWM8 \rightarrow GlobalInputBus[0 : 3] = True \Rightarrow$ **(Topology Constraint Adherence)**

This demonstrates that again, various configurations can meet user imposed topology constraints which cannot be met by other configurations. Other topology constraint requirements can be in terms of resource usage such as clock source, input source, output source, etc.

Topology **estimation** could use metrics such as $\frac{usedConnections}{totalConnections} = \%ConnectionUtilization$ to give a feel for the communication of a particular configuration.

Performance

Another configuration classification could be in terms of the performance of the machine. For example the M8C can be run at a variety of clock speeds. The assembly code run on the processor has a fixed number of clock cycles. In addition, the peripherals run off a clock in order to function (i.e. counter). The code you create and the placement of peripherals to use various resources are key decisions. Decisions such as this are performance distinctions.

A trivial example of how code and clock speed could affect the performance is shown³:
 $10^6 Instructions * 5CPI * 1 \frac{Second}{1millioncycles} = 5seconds < 10^3 Instructions * 5CPI * 1 \frac{Second}{1cycles} = 5000seconds \Rightarrow$ **(Performance Estimation)**

Finally, an example of how device placement can **constrain** performance. Some blocks cannot be assigned to connect to all pins on the device. This could be expressed as: $ACA01input = Port0.2 \neq true \rightarrow$ Can't get input needed to perform certain requirement \Rightarrow **(Performance Constraint)**

You can **estimate** performance in a number of different ways. As will be talked about in more depth, section 5.1 the HySAM [6] system uses the metrics, *execution latency, execution throughput, precision of operands, and input and output bandwidth required*. The toolset described later will use similar metrics.

Scheduling

The final configuration classification is really inter-configuration scheduling. Since the PSoC is a dynamically reconfigurable device, an application can be made up of many different configurations each running at different times during the course of an execution. How to decide when and in which order this happens is a scheduling configuration issue. The time it takes to change configurations may be an important issue in real time systems where deadlines need to be met. For example if \rightarrow represents the change from one configuration to the next in a particular order the following condition may occur:

$[Config1 \rightarrow Config2 \rightarrow Config3](CycleCount) < [Config2 \rightarrow Config1 \rightarrow Config3](CycleCount) \Rightarrow$
(Scheduling Constraint)

A metric needs to accompany a configuration so that this analysis can be performed. This could be tied in with a partial reconfiguration framework where the order of configuration operation affects data sharing. **Estimation** is naturally a component of the framework equation above since cycle counts must be estimated in the first place in order to make such a comparison. Another estimation could be $\frac{configurationsChanged}{totalConfigurations} = \%PartialReconfiguration$

The focus of the configuration exploration will be how to exploit these four areas both in terms of their constraint and estimation portions. The constraints ensure that the configuration reflects the design space which is required for the application and the estimation allows the user to evaluate configurations within that space. This attempts to address the design challenge in Section 1.2.3 which refers to the challenge of multidimensional optimization.

The next Chapter will introduce the concepts of platform based design and how they rely on both on the constraint and performance estimation framework. This discussion will tie together all of the proceeding Chapters into one unified framework.

³CPI=Cycles Per Instruction

Chapter 3

Platform Based Design (PBD)

Platform Based Design can be interpreted as many ways as people you ask. This definition often varies based on three areas in which many designers and developers fall: *systems*, *semiconductor*, and *academic*. Each of these has their own motivation and views which serves to demonstrate that this is an evolving concept. This paper takes the academic view as proposed in [4] by Prof. Alberto Sangiovanni-Vincentelli at the University of California, Berkeley. This view is also being accepted in other frameworks such as [36]. This is founded on the two basic tenets taken verbatim from [4] below:

- The identification of design as a “meeting-in-the-middle process”, where successive refinements of specifications meet with abstractions of potential implementations.
- The identification of precisely defined layers where the refinement and abstraction process take place. The layers then support designs built upon them, isolating from lower-level details, but letting enough information transpire about lower levels of abstraction to allow design space exploration with a fairly accurate prediction of the properties of the final implementation. The information should be incorporated in appropriate parameters that annotate design choices at the present layer of abstraction. These layers of abstraction are called platforms to stress their role in the design process and their solidity.

The first bullet gets at the notion that this is both a “top down” and a “bottom up” process. This is shown in Figure 3.1. The “meet in the middle” is the result of the constrained application and the exported implementation information.

Figure 3.3¹ shows how Platform Based Design can be applied at a variety of levels. This is known as the “fractal nature” of design. PBD is not intended to apply at only one level but to continue at each stage of the design. This diagram shows how this methodology can adapt to any stage of the design while requiring similar constraint and estimation issues at each stage.

Another important aspect of design is the *orthogonalization of concerns* [26]. One of the key concepts here is the separation of *functionality and architecture*. This is a natural consequence of PBD. These two aspects happen at all levels of abstraction. The functionality is captured in the more abstract model and then architectures can be independently explored in the implementation model. PBD will be used to explore reconfigurable architecture configurations by capturing the functionality at a coarse user module level and applying constraints and estimating the performance of the possible configurations adhering to those constraints.

3.1 Constraint Propagation

This is the “top down” portion in which the more abstract representation is reduced to an abstraction which has the required semantics for the progression of the methodology. The key is that the constraining only remove

¹Source A.R. Newton at the University of California, Berkeley

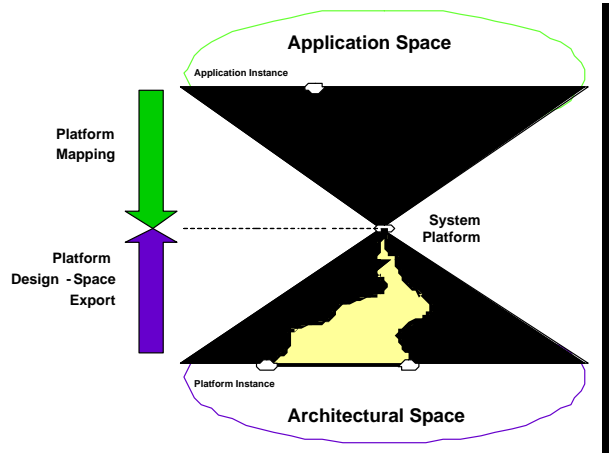


Figure 3.1: Platform Based Design Framework

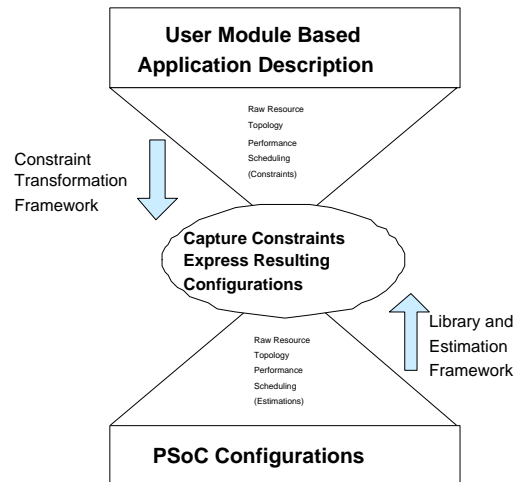


Figure 3.2: PBD with PSoC

the attributes which are required by the constraints and nothing more as not to bias the design toward one particular implementation. This effectively restricts the design space while retaining the key semantics of the application.

In this investigation's case this will be the set of all possible configurations for a given application, being reduced by the *raw resource, topology, performance, and scheduling constraints*.

3.2 Performance Estimation

This is the “bottom up” portion where the possible next layer uses the information regarding the platform to estimate what the performance at whatever the next abstraction level may be. This aspect allows for the reuse of existing modules and components. In the case of a reconfigurable architecture, the bottom up aspect represents the various configurations which the device can take in order to realize a design.

Again, for this investigation this will use the *Raw Resource, Topology, Performance, Scheduling estimations* in order to provide feedback on the particular configuration. This will be done by building a **library** which will have elements and their interconnections. Understanding how particular configurations interact with that library will provide these performance estimations.

3.3 Successive Platform Refinement

In order to perform design space exploration, PBD is fractal in nature as mentioned. This means that PBD happens at various abstraction levels in the design as shown in Figure 3.3. In order to move through this design space there is the notion of relationships between “*Abstract*” models and their “*Refined*” counterparts. Key to this relationship is that if properties hold in the abstract model then these same properties hold in their refined model. Holding these properties will prevent the need to check them again and hence result in a verification savings throughout each abstraction level. Any investigation using PBD should provide models which are intended to have this relationship. When developing the case studies in Chapter 10, the application descriptions are developed for both abstracted and refined views. For more information on successive platform refinement and PBD see [12].

3.4 PSoC and PBD

Exploring the PSoC in the PBD methodology can be shown in Figure 3.2. What this demonstrates is that the design will start with a User Module based application description. This was chosen for two reasons: **(1)** it is

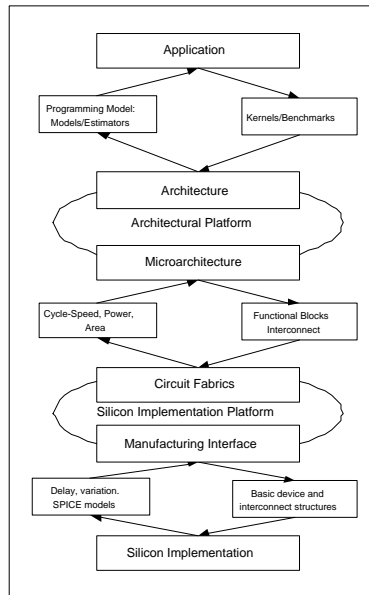


Figure 3.3: Platform Based Design Discipline

the appropriate granularity for this architecture classification, (2) it allows our constraint types to be easily applied. This description will be constrained, resulting in a more detailed representation. This will be the “platform” and will ultimately encode many possible configurations. This platform representation will be then analyzed with the key estimation metrics in the four areas mentioned. Once the estimations are made, the designer can choose amongst the configurations to meet their needs.

3.4.1 Configuration as Architecture Instance

The key insight regarding Platform Based Design and Reconfigurable Architectures is that the possible platform instances of a typical static design are each individual static architectures. However, with a reconfigurable device, each instance is simply another configuration. Therefore the platform is a constrained representation of the application which maintains characteristics of multiple configurations. The selection of one of those configurations and estimating its performance is the “bottom” up portion. The fact that reconfigurable architectures can be seen as multiple implementation instances is convenient for several reasons. Firstly, the performance estimation libraries only need to be created once. Subsequent configurations will be able to use the same metrics (as opposed to creating metrics applicable/relevant to many hardware components). Secondly, the creation of a platform can be more abstract since details can be implicit regarding implementation since all configurations will be on the same device. Finally, a rapid prototyping system can be developed where multiple configurations can actually be tested. The toolset in this report takes advantage of these characteristics.

Chapter 4

Boolean Constraints

This section provides a brief introduction and background regarding the use of Boolean constraint formulations. Essentially this requires that some system be represented as a Boolean equation or an equation using Boolean variables. The system is functional if the equation can be “satisfied” or in other words an assignment to the Boolean variables can be made in which the resulting formula evaluates to “true” or “1”. Systems are formulated in this way as to provide automatic methods to solve large constraint systems that would be much too unwieldy to do by hand. This is often used by the EDA community for Automatic Test Pattern Generation (ATPG) or Circuit Equivalence checking. What is of concern is to keep the complexity in P or to keep the problem size small. Since the problem is often NP-Complete, this formulation is used since it is easy to automate the creation of these problem instances (as in this discussion) and this automation outweighs the complexity concerns.

Boolean constraints in this report really mean that the constraint problem is *binary* in nature. All of the constraint formulations in this investigation assume two-valued logic, $\{0,1\}$. These problems arise in such instances as SAT (to be discussed) and colorability problems just to name two instances. Often the idea is to formulate a problem in which variables are representative of some quantity’s assignment (present(1)/not present(0)). The constraints serve to rule out the various illegal or undesirable assignments. The result is the *legal assignments* \subseteq *all assignments*. The following sections provide the background on the specific type of constraint formulation used in this investigation. In particular how it relates to the PSoC and ultimately how it is used to choose a configuration for the device.

4.1 Preliminaries

This section is a collection of definitions to formalize the discussion. These terms will be used in later discussions and are the foundation of the Boolean constraint formulation of the toolset.

Definition 4.1.1 A **literal** is a Boolean variable or its complement.

Definition 4.1.2 A **two level logic expression** refers to a Boolean formula which can be represented by a parallel series of same gates (AND, OR) followed by a second set of parallel same gates (AND, OR). The order in which these gates occur determine the type of two level circuit. Inverters are implicitly considered at the inputs which can technically make such circuits three level.

Example 4.1.1 $ABC+DEF \rightarrow$ (Two Level); $A(J(ABF+DEF)+GH) \rightarrow$ (Not two level)

Definition 4.1.3 A **completely specified boolean formula** f of N input variables, x_1, \dots, x_n and M output variables, f_1, \dots, f_m is a mapping $f : B^N \rightarrow B^M$ where $B^N = \{0,1\}^N$ and $B^M = \{0,1\}^M$. A function where $M=1$ is a **single output function**.

Definition 4.1.4 A **constraint** is a Boolean function $f: D^k \rightarrow \{0,1\}$ where k is a non-negative integer called **arity** of f .

Definition 4.1.5 Given a constraint $f: D^k \rightarrow \{0,1\}$ and (i_1, \dots, i_k) the pair $\langle f, (i_1, \dots, i_k) \rangle$ is referred to as an **application of the constraint** f to x_{i_1}, \dots, x_{i_k} .

Definition 4.1.6 Every assignment $\phi: \{x_1, \dots, x_n\} \rightarrow D$ naturally extends itself to any constraint application $C = \langle f, (i_1, \dots, i_k) \rangle$, we have

$$\phi(C) := f(\phi(x_{i_1}), \dots, \phi(x_{i_k}))$$

An assignment ϕ **satisfies a constraint application** C if $\phi(C) = 1$.

Definition 4.1.7 A constraint is said to be **0-valid** if $f(0, \dots, 0) = 1$ or **1-valid** if $f(1, \dots, 1) = 1$.

For more information see [13] and [10]. There are many other concepts which support these definitions and they can be found in most logic synthesis textbooks and tutorials.

4.2 Conjunctive Normal Form (CNF)

Constraint formulations often take a particular form when formed as a Boolean equation. This section details the key representation used in this discussion. CNF formula will be the form in which the Boolean constraints take for the configuration exploration in this report.

Definition 4.2.1 A **clause** is a disjunction (Sum; OR; Union) of literals.

Definition 4.2.2 **Conjunction Normal Form (CNF)** (often also called *Product of Sums (POS)*) of a Boolean equation is a two level logic representation which is a conjunction (product; AND; Intersection) of clauses. .

Example 4.2.1 $\phi = (A + B + C)(D + E + F)(\bar{A} + \bar{D})$ is a CNF formula with three clauses, 8 literals, and 6 variables. Notice that in order for a CNF formula to result in a logical “1”, each clause must result in a “1”. This results when $B=1, E=1$ and $D=0$ (one of several satisfying assignments)

In addition to CNF there are several other properties that require a brief definition.

Definition 4.2.3 A function f is **Weakly positive(weakly unate)** if f is expressible as an CNF formula having at most one negated (unnegated¹) variable in each clause.

Definition 4.2.4 A **2CNF** formula is one made up of clauses containing exactly 2 literals each.

Definition 4.2.5 A function f is **Bijunctive** if f is expressible as a 2CNF formula.

Definition 4.2.6 A function f **Affine** if f is expressible as a system of linear equations of the forms $v_1 \oplus \dots \oplus v_n = 0$ and $v_1 \oplus \dots \oplus v_n = 1$ where \oplus denotes the logical XOR function.

For more information see [22] and [10].

¹Such clauses are usually called Horn clauses

4.2.1 CNF Satisfiability

CNF Satisfiability or *CNF SAT* is the process of determining whether or not there is a satisfying assignment to the CNF formula. This is a highly studied problem for a variety of reasons. CNF SAT is a method of ATPG [28], Circuit Equivalence Checking [19], and obviously constraint solving. Once you have a constraint formulation, the item of key interest is what is the *solution* to the constraint application.

Much academic work has been done in this area resulting in many tools know as “SAT Solvers”. A sampling can be found in [41], [32], and [18]. These all use various techniques such as Davis-Putnam search and heuristic local search. What techniques they use and how they apply them are what separates the solvers. CNF based solvers are extremely popular and make this formulation intriguing for many problems.

Complexity

Naturally, the complexity of CNF SAT is a primary concern. What is unique about SAT is that *every problem in this class is either in P or NP-Complete*. This was shown in [10] and is well accepted in the EDA community. The problems can be divided simply by the conditions listed below. If the constraint set F satisfies one of the following conditions it is in P else it is *NP-Complete*.

1. Every constraint in F is 0-valid (1-valid)
2. Every constraint is bijunctive
3. Every constraint in F is weakly positive (weakly negative)
4. Every constraint in F is affine

This is shown in much more detail in [10]. The key to this discussion is to demonstrate that the **CNF formulation is crucial to its complexity**. For the most part however, CNF SAT will be NP-complete. This is not necessarily a major issue as we will see when we introduce the use of SAT solvers on particular problem formulations in Section 9.1. Amongst them are the solvers listed previously by their references. What will be of importance is the execution time of the toolset (which will be affected by the complexity of the problem formulation). **The actual running time target is < 1 minute**. This would be a substantial decrease over the current design flow. In order to document the formulation structure, the tool will provide statistics regarding the type and quantity of the clauses created which will give insight into the running time.

4.2.2 CNF SAT and PSoC

As will be shown in Section 8.1, the possible (or legal) configurations of the PSoC can be represented as a CNF formula. Once this formulation is created it will be necessary to find satisfying assignments to this formula which in turn represent possible configurations.

Example 4.2.2 Assume the variable set $\lambda = \{A, B, C, D, E, F\}$. Each variable could represent a particular configuration of the PSoC registers for one particular User Module. Assume again that the set is further broken into the subsets, $\theta = \{A, B, C\}$ and $\delta = \{D, E, F\}$. The subsets are constructed such that at least one element of θ must be selected and at least one element of δ must be selected. This could correspond to choosing one of a set of possible ADC configurations and another possible set of DAC configurations perhaps. A configuration in this case would require that one ADC was selected along with one DAC. This naturally could be expressed as:

$(A + B + C)(D + E + F)$ where the evaluation of the expression to “1” requires the selection (literal set to “1”) of literals in both clauses. This is trivially satisfied by $C=1$ and $D=1$ for instance.

Now if we add a constraint clause $(\overline{C} + \overline{D})$ to the previous equation the result is $(A + B + C)(D + E + F)(\overline{C} + \overline{D})$. Now we have effectively eliminated the selection of C or D from the possible configuration set. This is how invalid configurations can be specified. Satisfying assignments will now no longer include C and D as a result.

In order to use methodology with the PSoC (or any other reconfigurable device) it will require the following steps:

1. Create a formula which represents all possible configurations of a selected set of user modules.
 - The formulation of the CNF formula is key as mentioned previously. The user module requirements will come from user input. This will simply be unconstrained and should represent the selection of the required modules. The main work in this area is to take a description of the application and translate that into peripherals encoded as literals.
2. Add to that formula constraint clauses which represent Raw Resource, Topology, Performance, and Scheduling Constraints.
 - The constraints will both come from the user as well as the PSoC device properties when appropriate. Those specified by the user will be more flexible while those from the device will be rigid and device specific.
3. Enumerate a set of satisfying assignments (if they exist) via SAT solving techniques.
 - Ideally enumerate all solutions. In the event this is large, heuristically select a subset. If no solutions exist then the configuration can not be realized. The user will have to iterate back to the input description and re- specify the design.
4. Each satisfying assignment is a potential configuration to estimate in terms of the Raw Resource, Topology, Performance, and Scheduling Estimations.
 - The user will pick a configuration based on the estimations which will reflect their goals.

With this process outlined and the required background detailed, the remainder of the report will be dedicated to the specifics of how the toolset was developed and the previous work that influenced it.

Chapter 5

Project Overview

This Chapter will introduce previous work relating to the configuration exploration of reconfigurable devices. This will help to provide a context for the next area in the Chapter which is the introduction of the toolset created. This toolset, CAPE, makes use of the techniques outline thus far and is the basis for this report.

5.1 Previous Work

In order to begin this investigation it is important to examine what work has been done previously in the area of programming reconfigurable hardware and in particular how configuration decisions are made for a given application. This will provide a context for this work. The previous work essentially falls into the categories below. For more information and references see [6].

1. Architectures

- Devices which propose various ways of organizing and interfacing configurable logic. Various approaches to the classifications made in Section 1.2 fall into this category. Both commercial and academic offerings are applicable. [39] is a unique case in which an environment is created from existing FPGA architectures and corresponding libraries are built for it. This naturally falls also into the software tools category which is indicative of difficulty in characterizing these approaches.

2. Theoretical Models

- Models such as reconfigurable meshes and Virtual Hardware Operating Systems [7]. The SCORE computation model can also fit into this category [8]. These models are concerned with the scheduling and partitioning of applications onto reconfigurable architectures.

3. Applications

- Specialized configurable architectures for specific applications. These often replace ASIC designs and exploit optimization based on a specific input instance of the computation. They look to take advantage of characteristics such as spatial computation and deep pipelining.

4. Algorithmic Synthesis

- New techniques to schedule computations on dynamically reconfigurable machines. [20] looks at how one can partially reconfigure FGPAs in order to reduce reconfiguration overhead and improve application performance.

5. Software Tools

- Tools to address mapping techniques, run-time reconfiguration, compilation from high-level languages (such as C), simulation, and complete operating systems for dynamic logic. [35] is a tool for specifying and simulating dynamically reconfigurable systems. It is intended to co-exist with current tools and practices. This is useful in this investigation when examining how they estimate reconfiguration latency as having both *system* (t_{ack} , $t_{control}$, and t_{init}) and *circuit parameters* (t_{remove} and t_{load}). System parameters detail the communication aspects of reconfiguration while circuit parameters deal with the physical aspects.

This investigation is concerned primarily with **Theoretical Models and Software Tools** and hence we will focus in this area with an expanded look at both HySAM (*Theoretical Model*) and DEFACTO (*software tool*) both from K.Bondalapati et al. at the University of Southern California. These were chosen not only since they are in the two areas most relevant to this report, but also since they both come from the same source they ideally will concern themselves with similar issues and take an approach which can be related to one another with the same perspective. This will provide an alternate approach to similar issues as this investigation. With the conclusion of this discussion we will move into the particulars of the report and accompanying toolset.

5.1.1 HySAM

HySAM stands for *Hybrid System Architecture Model* and is introduced in [6]. This is a general model consisting of a von-Neuman style processor with an additional Configurable Logic Unit (CLU). In addition it has memory, a configuration cache, and an interconnection network for these devices. The models consist of two aspects: *declarative and generative*. The declarative aspect of a model specifies the parameterized model of the hybrid architecture. The generative aspect specifies the ways in which the declarative aspect can evolve based on a set of generative functions. In addition, the model partitions the *capabilities* of the hardware from the *implementations* (much like our discussion of separation of concerns in Chapter 3).

Functions and Configurations

The input to the system is an application. This application is partitioned into tasks. These tasks are further decomposed into a sequence of CLU functions, F . A configuration denotes a specific structure and arrangement of the configurable logic which has a given functionality. The uninitialized configuration is C_0 , execution on the CPU is C_{cpu} , and subsequent configurations are C_j . Functions to configurations have many-to-many relationships. For example a configuration can contain both addition and logical OR. Configurations each have many different characteristics such as area, time, precision, etc.

Attributes

The HySAM model associates parameters with each function-configuration pair. These are:

- t - execution latency
- p - execution throughput
- π - precision of operands
- β_{in} and β_{out} - input and output bandwidth required.

The attributes can be extended to include additional variables that reflect other optimization criteria.

Generative Aspect

The generator G is a composition function, $G:C \times C \rightarrow C'$. It abstracts the process of composition of the configurations to generate other configurations. The physical process of building computing solutions is abstracted using the generators. The generators are functions which define the various forms of composition. This includes parallel composition, serial composition, and pipelining.

Execution Model

Finally, the input to the system is an application which is partitioned into tasks to be executed on the CPU and the Configurable Logic Unit. The application tasks to be executed are then decomposed into a sequence of CLU functions. This is converted to a task graph which is mapped to a sequence of configurations. Algorithmic optimization techniques are used to arrive at an optimal sequence of configurations.

5.1.2 DEFACTO

DEFACTO is a design environment for adaptive computing control presented in [5]. A user of DEFACTO develops an application in a high-level language such as C, possibly augmented with application-specific information. The system maps this application to an adaptive computing architecture that consists of multiple FPGAs as co-processors to a conventional GPP.

System-Level Compiler

The DEFACTO system level compiler has the following goals:

- Identify computations that have the potential to yield performance benefits by executing on a CCU.
- Partition computation and control among the GPP and the CCUs.
- Manage data storage and communication within and across multiple CCUs.
- Identify opportunities for configuration reuse in time and space.

The compiler analysis produces an annotated hierarchical source program representation which is called the Task Communication Graph (TCG). This specifies the parallel tasks and their associated communication. This is then refined to partition the communication and control among the GPP and CCUs.

The other aspects of the compiler are the identification of configurable computing computations, locality and communication requirements, and generating control for the partition.

Design Manager

The design manager provides an abstraction of the hardware details and guides the development of the final hardware design. This involves the computation and data mapping, an estimation tool, communication mapping, and storage mapping. This project is much more complex than can be described here and the reader is referred to [5] for more information.

5.2 CAPE

Taking both HySAM and DEFACTO as examples, a toolset should explore configuration combinations, provide feedback on those combinations, and identify opportunities for configuration reuse in time and space. The implementation in which those aspects and all of the previously described aspects of platform based design, reconfigurable logic (specifically the PSoC), and Boolean constraints meet is in the **CAPE Toolset**. CAPE stands for *Constraints, Applications, Performance, and Estimation* which reflects the components with which it is concerned. It is an environment which provides not only a specification “language” but also the engine which evaluates it. It is written primarily in the Java programming language but also makes use of PERL scripts and independently created SAT solvers. It consists of several interacting components:

- **Application Specification Language (ASL)** - This is the starting point for the design. Using a netlist-like syntax the designer can specify which user modules should make up an application, what primitive interactions they should have, and what are some basic performance requirements. This is described in Chapter 6.

- **Application Constraint Transform (ACT)** - This takes the application and transforms it into a representation of the requirements augmented with *raw resource, topology, performance, and scheduling* constraints. Chapter 7 has more information on how these constraints are formed.
- **Platform Abstraction Text (PAT)** - This is a text representation of the result of ACT applied to ASL. This is a CNF formula representing all possible feasible configurations. Chapter 8 has more on the PAT representation.
- **Platform Estimation Tool (PET)** - This is a combination of an existing SAT solver(s) and estimation metrics. This solves the CNF equation and provides information on the possible performances of the configurations.
- **Application Code Estimator (ACE)** - This is a companion estimation tool which evaluates PSoC M8C Assembly code and returns cycle counts for its execution. Chapter 9 describes this and PET.

In Figure 5.1 is how CAPE fits into the platform based design methodology:

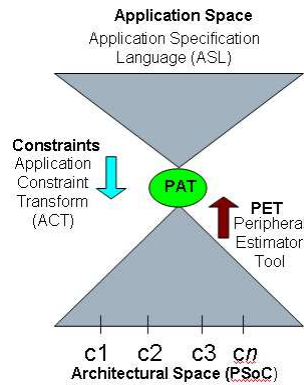


Figure 5.1: CAPE in Platform Based Design

As Figure 5.1 shows, the application space consists of ASL. This is the programming model for the designer and the point for all design iterations. The constraint space consists of ACT. This is the application of the constraint types mentioned throughout this report. PAT is the platform space and the “middle” of this toolset. The estimation space is both ACE and PET and provides information regarding the lowest level or architectural space which is the PSoC device itself in this investigation.

Design Process

Once one decides to use the CAPE toolset, the following procedure is used to produce and analyze a configuration. Note that the manual and automatic parts for both CAPE (C) and the provided IDE (I) are noted to compare where the savings in design effort is:

1. Decide on an application that one wishes to run on the PSoC (**Manual I and C**).
2. Decompose the application into which User Modules it will require to function properly (**Manual I and C**).
3. Determine which peripherals must communicate to each other (**Manual I and C**).
4. Write ASL description (**Manual C**).
 - The steps for the IDE would be to use each Subsystem as mentioned in Section 2.3. This would be much longer than writing ASL (5- 10×). Multiple iterations occur while trying to make legal configurations (**Manual I**).

5. Run *ACT* (**Automatic C; Nonexistent for I**).
6. Run *PET* (**Automatic C; Nonexistent for I**).
7. Decide on a configuration returned by *PET* based on the metrics reported back concerning potential performance (**Automatic C; Nonexistent I**).
8. Write Code for M8C (**Manual I and C**).
9. Run *ACE* (**Automatic C; Nonexistent I**).
10. Revise code if needed based on *ACE* results (**Manual I and C**).
 - Note that you will have no basis to rewrite code using only the IDE and therefore the performance of the code will not be known until you actually run it and determine whether or not it will meet the requirements.
11. Program configuration into chip with IDE tools. (**Manual I and C**).

The key in this process is that the configuration is generated automatically with CAPE and that no process which is currently automatic in CAPE is automated by the IDE. Furthermore, there is no automatic process in the IDE which is manual with CAPE.

5.2.1 CAPE GUI

All of the tools are packaged into a GUI as shown in figure 5.2.

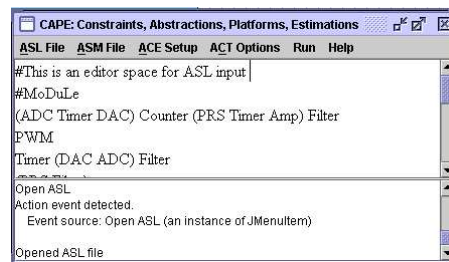


Figure 5.2: CAPE GUI

The GUI is a method to tie all the aspects of CAPE together in such a way that it is easy for the user to not only access the various tools but also provides a text editor for both ASL and M8C assembly files. It also provides a command window which shows which commands are executing and error messages when appropriate. There are several menu items:

1. **ASL File**. This allows you to open and save ASL files. This is the initial start to the design process. The initial text window in the GUI is the text editor for ASL.
2. **ASM File**. This allows you to open and save M8C assembly code files. In order to get the editor for this you must use the “open” command found under this menu. It is only the file you save with this menu that will be evaluated by the ACE tool.
3. **ACE Setup**. This allows the user to set the path to the Perl interpreter needed for ACE along with set the reference clock speed for the performance estimation.
4. **ACT Options**. This lets you select which SAT solver(s) you would like to use for ACT (ultimately used by PET). It also lets you select the set of constraints you want to consider. Currently only Chaff is fully integrated.

5. **Run.** This allows you to execute ACT, PET, and ACE as well as look at the results of each after they have been run.
6. **Help.** This opens text files which describe in more detail how to use the CAPE toolset.

The code for CAPE will be freely available at <http://www.cs.berkeley.edu/densmore/CAPE> . It provides basic documentation on how to use it. It requires both Java and Perl. More information on the tool should come from the help menu in the GUI. The following Chapters will introduce the detail regarding how each “space” in PBD manifests itself in CAPE and how the tool works at an algorithmic and implementation level.

Chapter 6

Application Space

The “starting point” in the Platform Based Design Methodology (or almost any design strategy) is the *application*. This is a top down ideology and the top half of the platform based design. What is key here is that the requirements be captured at this level sufficiently yet not over constrain the design. Also a minimal amount of structure should be imposed to prevent unnecessarily biasing the design. This is done to preserve the largest design space exploration process possible. CAPE uses ASL to capture the application requirements as described next.

6.1 CAPE: ASL

The *Application Specification Language* (ASL) was developed so that the designer can describe the application not only at a high level of abstraction but also in terms of the PSoC reconfiguration space. The PSoC consists of User Modules which can be realized on its analog and digital blocks. For example, the granularity of the design does not take place at the switched capacitor level but rather in terms of Filters, Timers, etc. ASL takes advantage of this by reducing these to a language consisting of **nine keywords**. They are *Timer, Counter, Filter, PRS, DAC, ADC, Amp, PWM, and Mux*. In addition “)” and “(” are also recognized and used to denote modules which require input or output from the other. Finally, ASL is broken into two sections. One is #Module where the previously mentioned text is inserted. Each line in #Module is considered a separate configuration in which the modules have no dependencies on any other configuration. The second section is #Perf in which the following format is used: `< device >< parameter >< value >`. The `< device >` can be any of the nine keywords that appeared in the #Module section. It will correspond sequentially to its #Module counterpart appearance. The parameters can be *Seconds, Clk(source), I(nput)Bus, or O(utput)Bus*. The `< value >` depends on the other two settings and is not strictly enforced (if it does not make “sense” it will be ignored). A “;” separates the configuration performance requirements. A sample snippet of ASL is shown in Figure 6.1.

Figure 6.1 has three configurations as shown by the three lines in the #Module section. The last two configurations consist of only one user module each. The first line has 4 user modules with 3 requiring a topological relationship. #Perf requirements apply sequentially for user modules with a “;” indicating which configurations they belong to. Note that in the example “//” are used for comments while no such construct actually exists in ASL.

6.1.1 Parsing ASL

In order to efficiently analyze ASL, a set of data structures were created to store the result of parsing the ASL. It begins with an *asl_parse* class with the following elements:

- Key Array - keeps keys to index into a hash table.
- Token Hash - hash table to hold the parsed tokens representing ASL user modules.

```

#Module
(ADC Timer DAC) Counter //ADC, Timer and DAC share inputs or outputs; Include
Counter
Filter //Second configuration
Timer //Third configuration
#Perf
Timer Seconds 20 //Timer period must be 20 seconds
ADC Clk ACLK1 //ADC Clk source; The values accepted depend on the
device
; //New configuration
; //New configuration
Timer Seconds 10 //Last timer period setting

```

Figure 6.1: ASL Example Code

The basic structure looks visually as shown in Figure 6.2. The key array simply holds double values which indicate the line of ASL with the integer part and the fractional part refers to what section of the line it is. The token hash holds linked lists of objects which reflect the information in ASL. A linked list length > 1 if “()” are used to indicate a topological relationship. Otherwise the list is length = 1. Each element in the linked list is a parse_peripheral object which holds the user module name, performance requirement (if one), and the performance requirement data (if one).

ASL is parsed sequentially and is fairly robust. It is not case sensitive and does not require keyword usage. However, the other tools in the chain will not function with invalid keywords. In addition the #Module declaration needs to precede the #Perf section.

The algorithm for parsing ASL was sequential and straightforward due to the fact that the syntax of the language was purposely design so that parsing would not be too difficult. What follows are sections detailing this process.

#Module Parsing

ASL is read in a line at a time from a file until the string “#Module” is read. When this occurs there are two types of statements which can follow it; a *single peripheral statement* <peripheral> or a *linked peripheral statement* (<peripheral><peripheral>). For linked peripherals they are stripped of their parenthesis “()”. A new object is created (parse_peripheral per = new parse_peripheral(< peripheral >)) and every peripheral read in is used in the constructor to make the new peripheral object. Each new object is stored into a *linked list*. An *array list* is also created to store the key of this object. This key is necessary because the linked list is eventually added to a hashtable at that key value. This process of creating new peripheral objects and adding them to a linked list is **O(n)** where *n* is the number of linked peripherals. Adding these linked lists to a hashtable has a run time of **O(m)** where *m* is the number of linked peripheral sets. Keys are in the format of “X.Y” where *X* represents the line number after the #Module and *Y* represents the number of peripheral groups on a specific line. The current key is updated as new lines and “()” are encountered.

If a singular peripheral is read the same process as the linked peripherals is performed except there is no need to strip the parenthesis off naturally. All together after taking into account adding peripheral objects to a linked list (O(n)), adding linked lists to a hashtable (O(m)), and having one major loop that is reading ASL from the file, **O(ASL Lines)**, #Module has a runtime approximately of $O(n \times m \times \text{ASL Lines})$ which at worst case is $O(n^3)$ since *n* is the dominating factor.

#Perf Parsing

#Perf is the only thing that can follow a #Module declaration (this is the semantics of ASL). As mentioned before, #Perf has the performance specifications of ASL. After the keyword #Perf is read the next line has the form

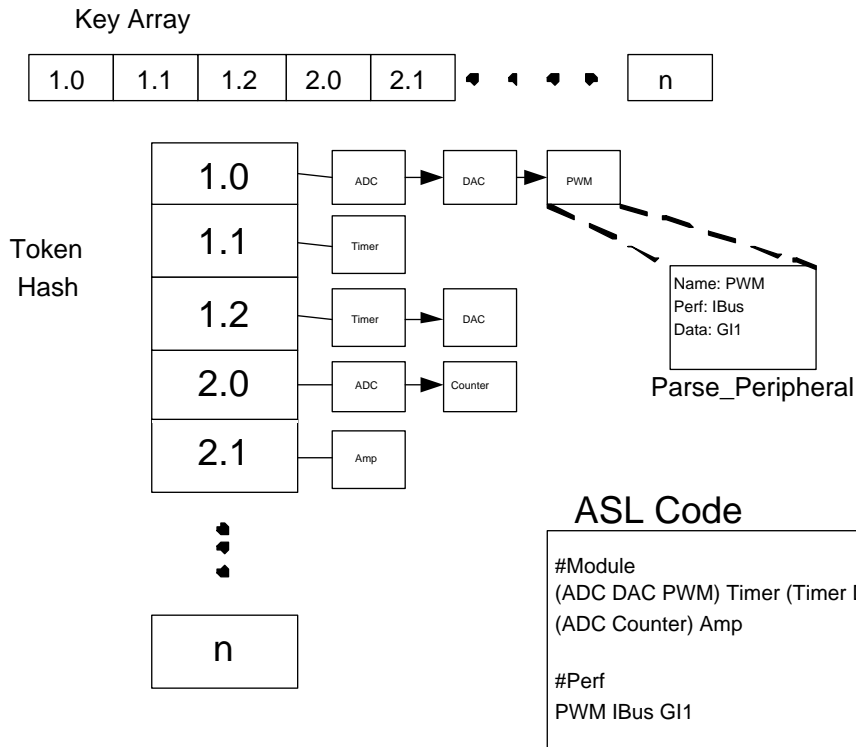


Figure 6.2: Data Structures to Hold Parsed ASL

of <peripheral name><peripheral type><peripheral performance>. The keys from the hashtable in the #Module section are used to properly index the corresponding lines in #Module. The linked lists are retrieved from the hashtable and then are iterated through to acquire the peripheral objects to modify with the #Perf information. The process of getting the linked list from the hashtable is $O(1)$ (constant). The process of adding the information to the element in the list is $O(n)$ where n is the size of the list. Once the peripheral object is accessed its class fields are set from the corresponding lines of #Perf. An example of this is illustrated in Figure 6.2.

Semicolons “;” in #Perf are designated to show new configuration pages. The parsing of #Perf has a runtime of $O(n^2)$ due to having to iterate through the linked list for each line, $O(\text{ASL Lines})$, before accessing the peripheral object in the linked list. Since the linked list will dominate the ASL line count, we can still approximately stand by the $O(n^2)$ complexity figure.

Chapter 7

Constraint Space

When looking to take the behavioral description of an application toward the implementation of that application in an actual electronic system, one must naturally examine what physical constraints will be placed on that application when implemented. While it is initially useful to examine an application for behavioral analysis without constraints, constraints are vital when the actual design must be mapped to hardware. Constraints can recognize the limitations/requirements of the environment in which the application will be used, the limitations of the physical resources which the application will utilize, or reflect implementation concerns such as cost and complexity. To ignore these constraints will lead to the development of systems which do not end up meeting the requirements of the application or in the worst case do not function at all.

Constraints in this investigation are what prevent not only the selection of a hardware configuration for a device from being trivial but also what allows a more rich and detailed configuration exploration process. The following sections detail the types of constraints considered in the toolset as well as how they manifest themselves as Boolean formula.

7.1 CAPE: ACT

The “top down” aspect of the Platform Based Design Methodology as implemented by CAPE is the *Application Constraint Transform* (ACT) portion of the toolset. This is where the ASL specification is transformed into a CNF formula. Depending on the constraint level selected, the CNF formula is modified to reflect the constraints specified in ASL. These constraints are in four categories. They are *raw resource*, *topology*, *performance*, and *scheduling*. The basic concept behind these areas was described in Section 2.5. Each of these constraint types is composed of several requirements to meet the overall constraint. Each requirement of the constraint will generate its own type of constraint clauses. For example, raw resource constraints must not only take into account the fact the devices cannot share physical locations but also must take into account the limited resources of the device. The next sections specify how these constraints are constructed and used to transform ASL into PAT. Before beginning that discussion, let us review what a literal constitutes in the CNF formulation for CAPE.

Example 7.1.1 *Literal Review*

Recall that literals in this system primarily represent two things: **1**. They represent the specific peripheral and **2**. they represent the locations that the peripheral would like to be placed. For example literal **A** might be represented as a string “DAC8DBA00” where “DAC8” is the peripheral and “DBA00” is the digital block location in which it can be placed. Locations may consist of more than one block depending on the peripheral. For example “DAC16DBA00DBA01” has both the specific location as well as **2** locations. The term “literal” will be used this way in the following discussion.

Each literal is initially created with the use of a library. As the asl parse structure is traversed, each generic peripheral is expanded to the set of all specific peripherals having that same type (i.e. PWM \rightarrow {PWM8, PWM16}).

Each of those specific peripherals has a set of blocks which it can be placed on. It is the enumeration of these possibilities which creates the literals. The literal set only needs to be created once. The initial clauses created with these literals are all **1-Valid** clauses. No new literals will be created by the constraints (only the manipulation of existing ones). To use the correct terminology, the initial “literal” set actually establishes the variable set and the creating of the actual CNF formula along with its constraint clauses is the actual instantiation and appearance of the literals. Figure 7.1 shows how information regarding general peripherals, specific peripherals, and location information is arranged to create literals.

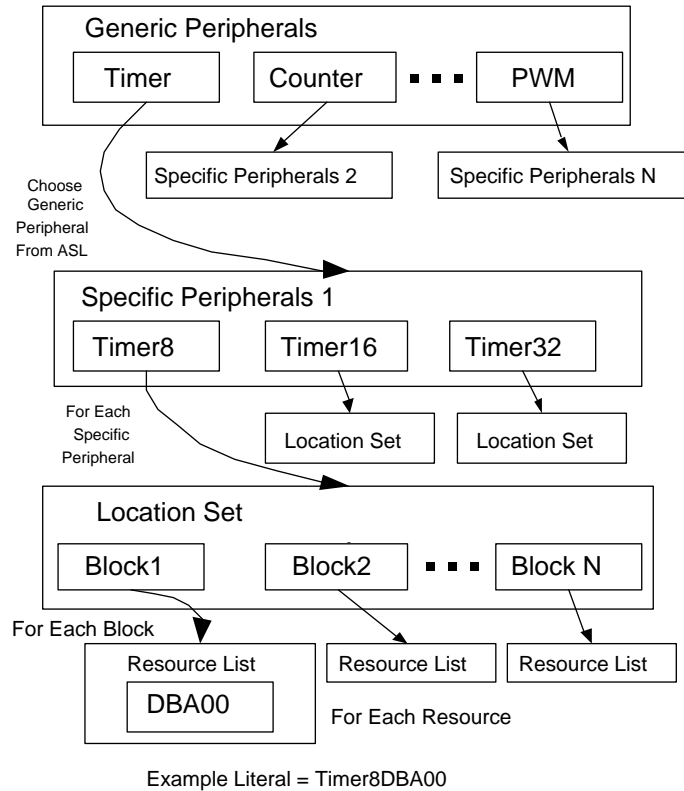


Figure 7.1: Peripheral Library Structures

7.1.1 Raw Resource

Raw Resource constraints deal with the physical resources of the device and the unique selection of one peripheral per device in ASL. While CAPE can be run with no constraints, this is the minimum constraint level which must be used to even produce a “legal” configuration. A “legal” configuration is one which can physically be implemented on the PSoC. A configuration without constraints is only useful in determining the theoretical “best” configuration.

Shared Locations

The primary issue concerning the Raw Resource constraints is the illegal sharing of reconfigurable blocks used by the User Modules. This occurs when the same block is used by two or more peripherals. The unconstrained representation of the configuration does not prevent this and therefore constraints must be added to prevent this.

Example 7.1.2 Shared Location Constraints Creation

Suppose **A** and **B** are literals in which the location component of the two is shared. This will require that in the event that one is chosen that the other is not chosen (or perhaps neither can be chosen). This will result in clauses of the form $(\bar{A} + \bar{B})$. This simply demonstrates that minimally for the clause to be satisfied, one of the literals must be a “0”. Applying DeMorgan’s law to this equation shows that the opposite or incorrect case is **A•B** which can only be satisfied (i.e. incorrect) when they are both set to “1”.

The creation of the shared location constraints is the *set of all pairs of literals which share physical locations*. These are a set of **2CNF, 0-Valid** clauses which are appended to the original unconstrained PAT representation. Figure 7.2 demonstrates this procedure.

Input: List(s) of literals which share locations. Lists are created in which conflicting location literals are gathered.

Output: Set of clauses representing legal positions to be appended to existing CNF formula.

$\forall i \in \text{SharedList}$ where i is a Literal

$\forall j \in \frac{\text{SharedList}}{i}$ where j is a Literal

Create clause $(\bar{i} + \bar{j})$

Figure 7.2: Creating Shared Location Constraints

This will run in $O(n^2)$ where n is the number of literals in the SharedList. This clause construction is also used to **prevent representing the same peripheral with two literals**. To do this the same algorithm is used but the input list(s) are literals which represent the same peripheral instance. It would be nice to separate this requirement out of the broad umbrella of Raw Resource constraints for the sake of design space exploration but technically is not over constraining the problem and is in fact a requirement.

Limited Resources

In addition to Raw Resource constraints concerning themselves with shared locations of peripherals, one must also take into consideration the fact that there are only a limited number of digital, analog, and memory resources for the PSoC. Therefore constraints must be added for these constraints also.

Example 7.1.3 Limited Resources Constraint Creation

Each literal can be queried as to how many digital or analog blocks it requires. In addition, it can also be determined how much memory it requires (thanks to a library lookup). This is also known for each PSoC User Module. Therefore, constraint clauses are added which prevent configurations which violate these requirements. This is done by greedily creating three lists which are collections of literals which when combined will violate a limited resource constraint.

For example if literal **A** requires 2 digital blocks, **B** requires 3 digital blocks, and **C** requires 4 digital blocks, then **A•B•C** cannot be valid since this totals more than the available 8 digital blocks. Therefore, the removal of at least one is required. DeMorgan’s law simply provides this as $(\bar{A} + \bar{B} + \bar{C})$. Due to the type of User Modules available for the PSoC there is actually not a configuration which can actually violate the memory requirement. Therefore checks for memory usage are added only for completion but not used in practice. Digital block consumption is between 1 and 4 blocks for module. This leads to clauses between 8 and 2 literals for digital constraints. Similarly modules can consume between 1 and 4 blocks for analog. This leads to clauses between 12 and 3 literals. Figure 7.3 demonstrates the procedure for making these constraints.

This will run in $O(n \times m)$ where n is the number of sublists and m is the number of literals in the sublist. A large part of this algorithm is actually the creating of the various sets in $List_i$. $List_i$ creation is done in $O(n^2)$ by looking at each literal and combining it greedily with other literals not of its user module type until it violates a constraint. The clauses created in this way are **0-Valid**. Some of these clauses may also be **2CNF**.

Input: List_{digital}, List_{analog}, List_{memory} which are lists themselves made of collections of literals which when combined violate limited resource constraints.

Output: Set of clauses representing configurations within the bounds of the resources.

\forall List_i where i \in digital, analog, memory

\forall Sublists_j \in List_i where sublists are greedily constructed literal sets

\forall Literals $k_1, k_2, k_n \in$ Sublist_j

Create clause $(\overline{k_1} + \overline{k_2} + \overline{k_n})$

Figure 7.3: Creating Limited Constraints

7.1.2 Topology

Topology constraints are derived from the designation that certain devices need access to the input or output of other user modules. This is denoted in the ASL file by “()” as described. From that, specific literals are determined to be potential selections to satisfy that requirement. From this, clauses are created which will require the satisfaction of this requirement by requiring certain literal combinations to be selected.

Strict Topology

These constraints can be used when you want to enforce a strict pair wise grouping of literals.

Example 7.1.4 Strict Topology Constraints Creation

Suppose **A** and **B** are specified syntactically in ASL to require them to be placed so that they can share inputs or outputs (i.e. via the “()” tokens). If this is true, CAPE will keep a list of all such literals which are members of each topological relationship. That will now require that clause sets of the form $(A + \overline{B})(\overline{A} + B)$. This requires that if you choose **A** you also choose **B** or vice versa (or neither). Again if DeMorgan’s law is applied to this it results in $\overline{A}B + \overline{B}A$ which shows the incorrect behavior as either one or the other. Figure 7.4 demonstrates this constraint creation.

Input: List of literals. This list is delimited by the literal “0” which denotes when literals referring the same device end. These are grouped such that literals **must be able to share and input or output connection**.

Output: Set of clauses representing the satisfaction of the topology constraint.

$\forall i \in$ Literals of user module p

$\forall j_1 \in$ Literals of user module k₁

$\forall j_n \in$ Literals of user module k_n

Create clause $(i + \overline{j_1})(\overline{i} + j_1)(j_n + \overline{j_1})(\overline{j_n} + j_1)(i + \overline{j_n})(\overline{i} + j_n)$

Figure 7.4: Creating Topological Constraints

This will run in $O(i \times j_1 \times j_n \times n)$ where the n is the number of user modules. This will be dominated by the literals containing a particular user module and the number of user modules. So if we say lit_{max} is the maximum number of literals for any of the user module this becomes $O(lit_{max}^3 \times n)$. This could be quite expensive. A “typical” design is quite reasonable however with user modules *using topology constraints* having an of average $n=2$ and $lit_{max} = 6$. If Topology constraints are added then the additional constraint clauses added are **2CNF** and those clauses all have only one negated variable which make them **Weakly Positive**.

Relaxed Topology

The strict topology formulation is actually not currently used in CAPE and can be found to be **too constrained**. Knowing this, the example previously where is A is chosen the B is chosen will not allow the fact that there are many other literals which can be combined with A and satisfy the topology requirement. Relaxed Topology Constraints will allow this.

Example 7.1.5 Relaxed Topology Constraints

In actuality there are many literals which may satisfy a topological constraint. For example if ASL specified (PWM ADC) then there will be a set of literals created for each peripheral. For example $\{A, B, C\}$ and $\{D, E, F\}$ respectively. If you take these literals and examine which ones which are in certain locations you can further break this into groups. For example $\{\{A, B\}, \{D, E\}\}$ and $\{\{C\}, \{F\}\}$. Now the clauses which should be generated should be: $(A + \bar{A})(\bar{A} + D + E)(C + \bar{C})(\bar{C} + F)$. This is continued for literals B, D, E, and F. The first clause gives the option of choosing or not choosing the literal. The second clause says that if that literal is chosen you must choose at least one of the others which can topologically satisfy the requirement (whereas the non-relaxed version required that you only had one choice). Figure 7.5 demonstrates this procedure.

Input: List of literals. This list is delimited by the literal ‘0’ which denotes when literals referring the same device end. These are grouped such that literals **must be able to share and input or output connection**.

Output: Set of clauses representing the satisfaction of the topology constraint.

$\forall i \in$ Literals of user module p

$\forall b_n \in$ Literals of user module k_1 in the same group as i

$\forall j_n \in$ Literals of user module k_n in the same group as i

Create clause $(i + \bar{i})(\bar{i} + b_1 + \dots + b_n)(\bar{i} + j_1 + \dots + j_n)$

Figure 7.5: Creating Relaxed Topological Constraints

The run time of the heart of this procedure will be $O(i \times b_n \times j_n)$. This is essentially $O(n^3)$ and this process will create the trivial 2CNF clause, $(i + \bar{i})$, and its corresponding **Weakly Positive** clauses.

7.1.3 Performance

Performance constraints are requirements explicitly assigned to particular peripherals in ASL in the *#Perf section*. They are unique since they each are manifested in different ways via a configuration. Recall all that can be manipulated by a configuration is the place in which you place a peripheral and the peripheral that is chosen itself. Depending on the performance constraint specified and the User Module it is applied to, this may or may not actually manifest itself into an actual constraint for PAT. These constraints are denoted by the keywords *Seconds*, *Clk*, *IBus*, and *OBus*. They are used as follows:

- $\langle device \rangle Seconds \langle double \rangle$ - This is used to determine the one of two things **(1)** the sampling rate of a peripheral **(2)** the timer comparison value. It is up to the user to ensure that the “device” paired with this parameter can realize one of these two scenarios.
- $\langle device \rangle Clk\{GO\{0-7\}, GI\{0-7\}, ACLK1, ACLK2\}$ - This is used to determine the clocking source for the peripheral. These clock sources are values ones which influence both location and how many peripherals can take advantage of them.
- $\langle device \rangle IBus\{GI\{0-7\}, ACI\{0-3\}\}$ - This denotes which bus input data should come from. Again, the user needs to make sure that the device actually could use that input (typically this is the $GI\{0-7\}$ for digital and the $ACI\{0-3\}$ for analog).
- $\langle device \rangle OBus\{GO\{0-7\}, AO\{0-3\}\}$ - This denotes which bus output data should go to. Like all other cases, this too must be matched to an appropriate device.

Each of these constraints are formulated in very much the same way as previous constraints. Seconds has one structure while the other three share the other structure. Figure 7.6 refers to Performance Type1 constraints (for “Seconds-like” constraints) while Figure 7.7 refers to Performance Type2 constraints (for resource usage constraints).

Example 7.1.6 Performance Type 1 (P1) Constraints

Suppose that literals **A** and **B** both have a “Seconds” constraint. In addition, they have been grouped into different “domains”. A domain is a grouping where the literal requirements are of the same order of magnitude. This

reflects the fact that they are similar (i.e. 100 seconds and 200 seconds; assumes that the same clock source could produce these values for a timer). Furthermore, they are grouped into “regions” depending on their location. A clause $(\bar{A} + \bar{B})$ will be created if A and B are in the same region but different domains. This clause will prevent the same region supplying two different domain requirements.

Input: List of literals which have share this constraint (i.e. all have P1 requirements).

Output: Set of clauses representing the satisfaction of the P1 constraint and the value for the user module register in order to realize that constraint (i.e. the value for the comparison register in a counter).

$\forall i \in$ Literals who are associated with a P1 constraint create a list with associated magnitude domains, $List_N$.

$\forall k \in$ Literals in $List_N$ organize into corresponding regions $\{D1, D2, A1, A2, A3, A4\}$ determined by the location information of the literal.

if $(N > 1)$

$\forall l \in$ Regions $\in List_z l$

$\forall j \in$ Literals of $List_z l$ where z is initially 1

for region $m = 1$

$\forall p \in$ Literals of $List_{z+1} m$

Create clause $(\bar{j} + \bar{p})$

Figure 7.6: Creating Performance Type 1 Constraints

This will run in $O(j \times p \times domains)$ where j and p are the number of literals using the constraints in a domain and “domains” is the number of seconds domains that exist. These are **0-Valid, 2CNF** clauses.

Example 7.1.7 Performance Type 2 (P2) Constraints

These constraints types are very straightforward. Assume that in #Perf a peripheral is denoted to need to use a particular clock source. All that has to be done is identify all the literals not in a location which can use that clock source and provide a single literal negated clause to prevent its selection.

Input: List of literals which are associated with P2 constraints.

Output: Set of clauses representing the satisfaction of the P2 constraint.

$\forall c \in$ #Perf Constraint in ASL

$\forall j \in$ Literals which do not match c

Create clause (\bar{j})

Figure 7.7: Creating Performance Type 2 Constraints

This will run in $O(n \times c)$ where n is the number of literals related to a constraint and c is the number of performance constraints specified. The clauses that it creates only have one literal so they are trivial to evaluate.

7.1.4 Scheduling

The final type of constraints are those regarding scheduling. Scheduling constraints refer to the ability of a reconfigurable device to be dynamically loaded with a configuration at runtime. The desire to have an explicit alternate configuration is denoted by the separation of lines in the ASL #Module section. If the user wishes to have an alternate configuration they simply specify that by starting a new line in ASL. The scheduling constraints will simply create a schedule in which the changing of peripherals and device settings is minimized therefore speeding up the application execution time.

The scheduling constraints are different from the other constraints in that they do not augment the CNF formula with clauses. Rather the information is formulated as a Linear Programming (LP) problem and used to generate a schedule that the designer will use when actually programming the device according to the results of the other constraints.

[24] and [33] provide several examples of formulating an Integer Linear Programming (ILP) problem with 0-1 integer variables. This investigation is based on that work. In particular it is the combination of *Time Constrained*

and *Resource Constrained Scheduling*. This results in *Feasible Scheduling*. The formulation is as follows in Figure 7.8.

Example 7.1.8 Scheduling Constraint Formulation

There are several items that the constraint formulation must be concerned with. Firstly, each literal (i.e. peripheral setting) must only be used in one configuration. This is captured by constraint (1). Secondly, during a configuration, no two literals can occupy the same location. This is captured by constraint (2). Finally, the dependency between literals must manifest itself in an ordering of the literals and their resulting configuration slot schedule. This is the case where one literal must be scheduled before another. This is captured by constraint (3).

Variable Definitions

L_{tk} are integer variables which denote the number of particular locations. Since each location for the PSoC is unique this value = 1 in all cases.

$x_{i,j}$ are 0-1 integer variables associated with literal o_i . $x_{i,j} = 1$ if o_i is scheduled in configuration j otherwise $x_{i,j} = 0$.

Linear equations

$$\sum_{j=0} x_{i,j} = 1, \text{ for } 1 \leq i \leq n; \text{ (1)}$$

$$\sum_{o_i \in \text{location } k} x_{i,j} \leq L_{tk}, \text{ for } 1 \leq j \leq s, 1 \leq k \leq m; \text{ (2)}$$

$$\sum_{j=0} (j * x_{i,j}) - \sum_{j=0} (j * x_{i',j}) \leq -1 \text{ for all } o_i \rightarrow o_{i'}; \text{ (3)}$$

Figure 7.8: Scheduling Constraint Formulation

The solution to this system of equations will result in an assignment of each $x_{i,j}$ variable to a 1 or 0. Each literal has a set of variables, each corresponding to a different “time slot” where the total time slots are the number of configurations specified in ASL. All of the variables but one for each literal will be “0”. The nonzero entry determines the overall order of the literals in the configurations. The user can then program the device to realize this schedule.

This concludes the discussion of constraint formulations and the next Chapter will discuss the platform space in this investigation.

Chapter 8

Platform Space

As mentioned the “meet in the middle” aspect of Platform Based Design is in fact the *Platform*. The platform space should be the representation of the constrained application with enough semantic information to represent multiple implementation instances. This representation should only reflect restrictions imposed by the constraints and be at such an abstraction level as to allow the exploration of this platform as various implementation instances. The key is that it *not commit to a particular implementation* but rather be simultaneously a representation of them all. Classical examples of this could be at various abstraction levels. For example, consider an RTL description of a system. This has the potential to be considered a platform if it is written in a behavioral fashion not committing to a particular topology. It will encompass the many ways in which the design can be realized. These designs will then be realized depending on the synthesis path chosen (i.e. what technology library chosen). While one can make the case that all models given a particular perspective are platforms, the key is that it should encompass some transformation from a higher abstraction level while remaining unbiased regarding its implementation. CAPE is reliant on an encoding which captures the intent of ASL and the constraints of ACT. This is PAT as described in Section 8.1.

8.1 CAPE: PAT

In CAPE, *Platform Abstraction Text* (PAT) is the representation of all possible configurations which can satisfy the ASL description. This is the platform in this investigation. This is naturally the result ACT and is CNF formula built during ACT and is shown in the DIMACS [23] format. DIMACS was chosen since it is not only a very common expression of a CNF formula but more practically due to the fact that it is the input format for three major SAT solvers (SATO, Chaff, BerkMin). PAT results from the original unconstrained CNF formula augmented with whatever level of constraints that the user has chosen in ACT. The CAPE GUI allows for the viewing to the CNF file through the menu options. Viewing this file gives some insight into how the various constraint clause structures look and how they are augmented together. An example is below.

Example 8.1.1 *DIMACS Format:*

$(A+B)(C+D)(\bar{A} + \bar{D})$ would be:

1 2 0

3 4 0

-1 -4 0

This shows that each literal has a unique integer representation and that each clause is terminated with a zero. The negation of a variable is shown with the “-” sign.

Figure 8.1 is a picture which demonstrates visually how PAT works into the CAPE flow for a trivial example. To see larger PAT examples please see the Appendix.

PAT could have easily been “bypassed” and never explicitly separated from ACT. However, this would prevent the modularity explicitly required by PBD. Separating out PAT allows for a totally independent estimation

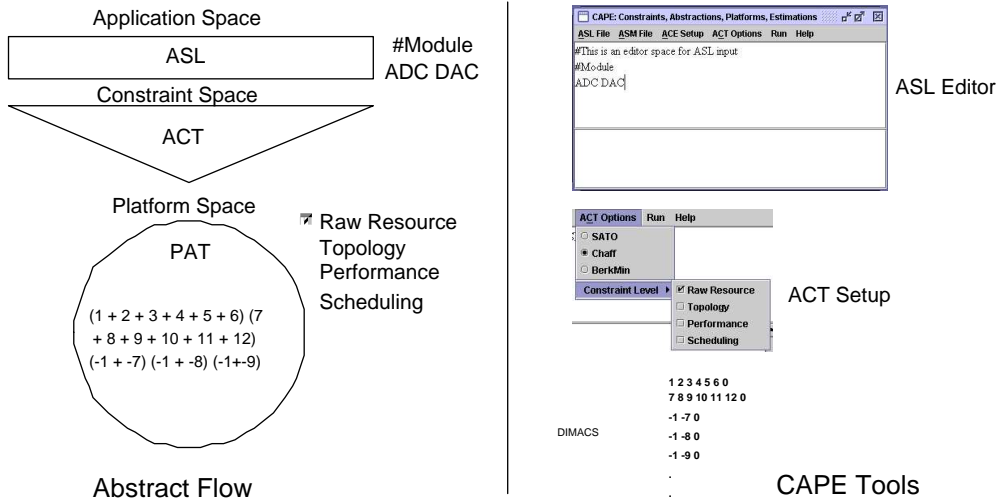


Figure 8.1: PAT in the CAPE Flow

process if one was so inclined. Additionally it provides insight into how the constraints are shaping the eventual mapping process. PAT can be moved up and down in abstraction level if need be. PAT could be more abstract by not being DIMACS but rather a transformation of ASL (such as the parsed data structure) or a graph based representation capturing the structure of ACT. A lower level abstraction than the current PAT could be the a more granular literal encoding which leads to a larger CNF formula. While the current version of PAT seems natural for this toolset, the point is that the platform target is not fixed and can adapt to the needs of the methodology.

In addition to PAT being a DIMACS formula, it is also implicitly the encoding of the literals. A literal actually tells the general peripheral represented by that literal object, the specific peripheral, its location, performance requirement(s), and resource usage. Therefore, $PAT \Rightarrow \{\text{DIMACS Code}, \{\text{Literal Objects}\}\}$. Naturally after PAT has been created it should be fed to an appropriate solver and used to provide an estimation of the performance of the configuration. That is described in Chapter 9.

Chapter 9

Estimation Space

The estimation space is the set of possible transformations of the platform into a set of particular implementation instances and then reporting back on the performance estimations were that platform to take one of those implementation instances. This is the “bottom up” aspect of PBD as shown in Figure 3.2. This process ultimately is how the final design decisions will be made regarding which configuration is chosen. The performance estimation must be generic enough to reflect the information provided by the abstract platform but yet accurate enough to provide meaningful information to the designer. The more encompassing the encoding of the platform, the better/more comprehensive the estimation process will be. The tradeoff between the complexity of the platform and the amount of performance information you would like is the major design decision for the toolset. Key is the efficient use of the platform. Therefore the estimation stage should take full advantage of the information provided by the platform.

9.1 CAPE: PET

The *Peripheral Estimation Tool* (PET) is the final stage of the CAPE toolset design process. The input to PET is PAT. PAT is the CNF formulation of ASL as discussed in 8.1. PET provides an analysis of the configuration. This is transforming PAT into various configurations for the PSoC and then estimating the performance and analyzing the problem structure. PET works as follows:

1. Pipe PAT to a SAT solver
 - Ideally several solvers could be selected from. At this point only Chaff is fully integrated. This is the way in which PAT becomes a concrete PSoC configuration.
2. Determine which positive literals are part of the solution to the SAT problem. Positive valued literals are the selection of the peripheral at that location.
3. Decode the literal to obtain information regarding the performance. Literals are a data structure which hold multiple pieces of information as described in earlier Chapters.
4. Use literal performance to analyze the performance of the collection of peripherals.
5. Solve the SAT problem again preventing the solution just received.
 - This involves the use of blocking clauses and is not integrated yet. This is how multiple PSoC configurations for one ASL description are generated. This is discussed later in the report.

Literal encoding contains much information. It tells what the generic peripheral type is (as specified in ASL; PWM, Timer, etc), it tells the specific peripheral this literal encodes (PWM8, Timer8, etc), where it is placed (DBA00,

DCA01, etc), how many blocks it takes up (analog and digital), the performance requirements specified in ASL, and the bit-level precision of the device.

Once the SAT solver has returned a solution (collection of literals), PET uses the literal information and there are several types of results PET can return.

1. Cumulative Values

- These are values that can be added up (or subtracted) from information in each literal. They accumulate and can be generated simply by a single traversal of the literal list in $O(n)$ time. Those returned by PET currently are **Digital Blocks Used**, **Analog Blocks Used**, **RAM Used**, **ROM Used**, and **Input and Output Connections**. Input and output connections are those made explicitly. These occur from performance constraints.

2. Upper/Lower Bound Values

- These values are the lowest or highest value found among the literals. These can be computed in a single traversal of literals in $O(n)$ time. The one example of this type of PET result currently returned by PET is **Precision**. This is the lower bound of all the bit-level precision values of each literal in a configuration. The rationale for this metric is the application is only as precise as the least precise element. An example of an upper bound metric could be **Latency** where the longest delay through a peripheral dominates.

3. Relational Values

- These are values that occur from both the type and location of the peripherals which make up the configuration. These are not only the most complex of results to calculate but also the most likely to deviate from the actual system performance. There are currently no metrics in PET which use this methodology. However some that are considered are **Throughput** and **Input and Output Bandwidth**. Since these have not been implemented there is not a running time estimate at this time.

Naturally the constraints selection set will influence the results of PET. Digital, Analog, RAM, and ROM resources used are a combination of all the constraints since these are simply based on the configuration returned. The input and output connections can only be made explicit by Performance constraints and will have the value zero unless a peripheral is specified. Finally precision is also a combination of all resources and depending on the resultant configuration.

In addition to the performance based results, PET also obtains the structural data of PAT. This is returned so that we can pinpoint possible ASL structures which are producing overly constrained PAT instances and also to observe potential optimization areas. The structural information includes the **Total Variables**, **Total Literals**, **Total Peripherals**, **Total Clauses**, **Initial Clauses**, **2CNF**, **1CNF**, **Horn**, **Weakly Positive**, **0-Valid**, and **1-Valid Clauses**.

9.2 CAPE: ACE

To extend this estimation framework into an implementation for the PSoC architecture instance, a tool was created called the *Assembly Code Estimator* (ACE). ACE is a tool developed to take M8C assembly language and provide the *performance estimation* in terms of how many cycles will be consumed for a particular part of the code. This is done as a static assessment of the M8C's 256 instructions broken into 37 instruction types. This is a CISC like architecture with 10 addressing modes. This architecture was described in section 2.2.1. The estimation task is easy to do since the M8C is a statically scheduled, multi-cycle datapath which only issues one instruction at a time. Examples of the M8C assembly code can be found in [30] and a sample is shown in figure 9.1.

The user is able designate which portions of code to be analyzed and then ACE will return how many cycles the M8C will spend processing this. This is done by annotating the assembly code with **;ACE Start** where one wishes to begin counting the cycles and **;ACE End** where the end of the cycle count should occur. The tool also does a simple calculation of how many seconds this will take provided a clock speed.

ACE can be used from the CAPE GUI. This is done by:

```
ADC A, expr ;Add with Carry Immediate Addressing
ASL [expr] ;Arithmetic Shift Left Destination Direct
MOV reg[expr], expr ;Move Destination Direct Source Immediate
```

Figure 9.1: Sample M8C assembly

1. ACE Setup → Set the path needed by Perl interpreter (ACE is a perl script).
2. ACE Setup → Set the clock speed that the system will run.
 - While you can set this at anything you desire, technically the clock speeds of the PSoC are 93.7kHz to 24MHZ. The actual steps provided by Cypress' tools are 24, 12, 6, 3, 1.5 Mhz and 750, 185.5, 93.7 kHz.
3. ASM File → Open the ASM file you want to analyze.
4. Add the ;ACE Start and ;ACE End comments to your file.
5. Run → Execute Tool → ACE
6. Run → Results Files → ACE Results

If one were to examine the code in figure 9.1 one would find that it takes **19 cycles** (4+7+8). Once combined with a clock cycle such as 24 MHZ this would give you an execution time of 7.916×10^{-7} seconds. If this was an unacceptable time then the instruction sequence may be changed to give a more desirable time. This falls into the category of *performance constraints*. These are typically real-time requirements in the case of execution but they could also indicate that sample rates may not be met or that code is simply inefficient. The design would then go through an iteration in the attempt to remedy this situation.

Chapter 10

Case Studies

In order to demonstrate the CAPE methodology, two case study applications will put to use the CAPE toolset and design flow and the results of these experiments will be documented and analyzed. These applications should make use of both the digital and analog capabilities of the PSoC as well as represent non-trivial applications with implications to various aspects of computing such as digital communication and signal processing. The two applications chosen are *μ-Law Companding and Pulse Coded Modulation (PCM)*. This examination will entail the following aspects:

1. A written description of the application. This will serve as background to the reader unfamiliar with these applications and provide an example of what high level information may be available to a designer when starting the process of formulating ASL. Key to this area is the identification of what high level components will be needed.
2. A ASL description of the application is developed. This simply will be the abstract description that will be fed to CAPE. Two ASL descriptions will be provided. This is done in order to demonstrate that since ASL provides a coarse granularity of User Modules, there is not a canonical representation of an application but rather a *successive refinement* of the descriptions. This refinement is representative of platform based design [12].
3. PET results of CAPE derived from PAT set with various ACT constraints will be gathered. Simply the raw results of the tools as well as insight and analysis will be provided. CAPE allows for constraint levels to be selected. Results with successively more constraints “turned on” will be shown regarding Raw Resource, Topology, and Performance. Scheduling constraints are not used since the applications did not utilize multiple configurations at run-time. Scheduling constraints will be discussed only theoretically. Additional information will be provided in the Appendix regarding the running of the experiments.
4. A comparison between naive initial configuration attempt (or published solutions) will follow. In the case of *μ-Law Companding* there is a published solution in an Cypress Application note [16] which will serve as a comparison. In the case of Pulse Coded Modulation, a naive implementation will be created using the most primitive components placed greedily on the device.
5. General conclusions on the method will be the final piece in each section. Finally this will comment on the findings and make suggestions regarding how the gathered information can be used to generate better designs.

The results were all done by writing ASL in the CAPE text editor window, selecting the Chaff SAT solver from the drop down menu, setting the appropriate combination of constraints from the constraint selection menu, running ACT, and then running PET both from the GUI execution menu. With the exception of ACE, this was a total exercise of the CAPE toolset.

10.1 μ -Law Comanding

The first application to be examined is μ -Law *Comanding*. This is a well accepted application for implementation on the PSoC and is documented in [16]. This application is used to convert an analog voice band signal and produce a digitized compressed value. An expanding DAC is developed which restores the compressed digital value back to the analog value.

This application arises in telephony where applications are increasingly becoming digital. μ -Law is a technique of data compression and expansions which allows for a greater dynamic range given the same signal bandwidth.

There are two parts which can be naturally separated due to the processing steps of this application:

- μ -Law Compressor
 - This requires a PreAmp, ADC, and Serial Transmitter (UART)
 - * This will be represented by an AMP and ADC in the ASL description. The UART will not be in ASL at this time due to the expressiveness of the language.

See Figure 10.1 for a picture of the compressor from [16].

- μ -Law Expander
 - This requires a Serial Receiver (UART), Four Pole Switched Cap Filter (Two filters), Two Pole Sallen Key Filter (Amp and resistors), and DAC.
 - * This will be represented by two FILTER objects, a DAC object, and AMP object in the ASL description. The UART will again be omitted due to ASL expressiveness limitations.

See Figure 10.2 for the picture of the expander from [16].

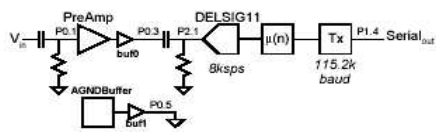


Figure 10.1: μ -Law Compressor

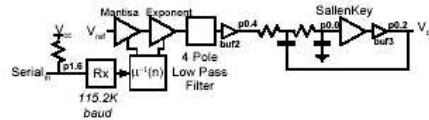


Figure 10.2: μ -Law Expander

10.1.1 ASL description

When designing this application the first thing to decide is if one wishes to put both the expander and the compressor on the same configuration. In practice the two sections would be on different chips but combining them takes unique advantage of the PSoC capabilities and will be used in this investigation. As for putting them in different configurations (as denoted in ASL by separate #Module lines), they will not be different configurations since (1) Scheduling constraints are not investigated in this report and (2) all the peripherals will fit in one configuration. To incur the overhead with dynamically switching configurations is not advantageous in any event and should be avoided unless necessary to get all the peripherals into the application or if the application switches rarely and remains in each configuration for a long time relative to the overall execution time.

There are two ASL descriptions provided. One is considered “*Abstract*” and the other “*Refined*” this is meant to demonstrate the concept of successive platform refinement in Platform Based Design from Section 3.4. The abstract ASL description reflects the expander and compressor with a basic performance requirement for the compressor ADC clock source. The two Filters of the expander are tied together topologically as are the ADC and AMP of the compressor. However the expander DAC and AMP are not required to explicitly communicate with the filters. Figure 10.3 shows this.

```

#Abstract ASL for uLaw Application
#Module
(FILTER FILTER) DAC AMP (ADC AMP)

#Perf
ADC Clk ACLK2
;

```

Figure 10.3: Abstract ASL for μ -Law Companding

The second or refined ASL requires more explicit topological constraints on the expander components and requires that the AMP and Filter be using the same input and output buses respectively which will allow for communication between the two. This is shown in Figure 10.4.

```

#Refined ASL for uLaw Application
#Module
(FILTER FILTER DAC AMP) (ADC AMP)

#Perf
ADC Clk ACLK2
AMP Ibus ACIO
FILTER Obus A01
;

```

Figure 10.4: Refined ASL for μ -Law Companding

As mentioned these are not the only ways to express these applications but they represent the key issues (peripheral selection and placement) that CAPE looks to address. The UART is eliminated since in this version of CAPE, UART is not a recognized keyword and does not have the corresponding library elements in which to make literals representing the object for ACT. However this is an extension which simply requires effort (not a theoretical change) to remedy.

10.1.2 PET Results

This section will detail and analyze the results of PET. ACT was run with the following constraint combinations: No Constraints (**None**), Raw Resource (**R**), Topology (**T**), Performance (**P**), **RT**, **RP**, **TP**, and **RTP**. PET information was detailed in Section 9.1. “Best” configurations are rated as those which are satisfiable (pass the SAT solver), valid (could be realized on the PSoC), and have the most desirable PET estimations as described in Section 9.1. Both the ASL descriptions were run with the constraint settings outlined here.

Best and Satisfiability Analysis

The “Best” configuration for both ASL modules was with the Raw Resource and Performance constraints applied (RP). *All application of constraints were “satisfiable”*. As expected only the results obtained with the Raw Resource constraints applied were valid configurations for implementation on the actual PSoC device. Table 10.1 shows the results regarding which specific peripherals and locations were in the “Best” configuration.

In Table 10.1, BPF = Band Pass Filter, LPF = Low Pass Filter and the other names refer to specific peripherals in the PSoC user module library. The keys to notice are that (1) This produces valid results which could actually be used on the PSoC device! (2) The configurations are different for each ASL description. This demonstrates the design space exploration opportunities. With this information a designer could select and place the User Modules confidently

Abstract μ Law	All SAT					
Constraints	Filter	Filter	DAC	AMP	ADC	AMP
RP	BPF2	LPF2	DAC6	AMPINV1	DELSIG11	AMPINV1
	ASB20 ASA21	ASB10 ASA11	ASA10	ACA01	ASA12	ACA00
Refined μ Law	All SAT					
Constraints	Filter	Filter	DAC	AMP	ADC	AMP
RP	BPF2	BPF2	DAC6	AMPINV1	DELSIG11	PGA
	ASB22 ASA23	ASA21 ASB20	ASA10	ACA00	ASA12	ACA03

Table 10.1: μ -Law Best and SAT Analysis

on the actual device for this application without fear of an design iteration due to incorrect performance or lack of resources.

Performance Estimation Numbers

Table 10.2 shows the PET performance numbers for each “Best” configuration. They both allow for 6-bits of precision, use 8 analog blocks, 1 digital block, 9 bytes of RAM, and 319 bytes of Flash ROM. The difference in the Input and Output Connection counts come from the addition of the performance requirements in the ASL for the refined version. The lesson learned here is that **both configurations are essentially equal** from a performance standpoint. This demonstrates that the addition of performance constraints did not make for a “better” performing configuration in this case. **However** the performance constraints do lead to an alternate mapping which may be required if those performance constraints are required for correct functionality!

Perf. Metric	Ab μ Law	Ref μ Law
Precision	6 bits	6 bits
Input Connects	0	1
Output Connects	0	1
Total DB	1	1
Total AB	8	8
RAM	9 bytes	9 bytes
ROM	319 bytes	319 bytes

Table 10.2: μ -Law Performance Estimation Comparison

Structural Analysis

Tables 10.3 and 10.4 show the structural (and execution time) aspects of ACT which PET returns. These are shown to demonstrate which constraints produce which clauses and how the different combination of constraints result in different structures in ACT. Values in **bold** show the earliest (least amount of constraints) high and low values. The earliest is important since there is not reason to continue to constrain the problem if the same result could have been obtained with fewer constraints.

The key highlights from these two tables of results are that **7.3 seconds** was the longest execution time (which is mainly I/O of the tool; not SAT solving). This is very reasonable. Naturally all cases have the same number of variables since this is based on the number of peripheral location instances that are possible. The only valid configuration will have 6 total peripherals (one for each ASL object). This only occurs when *R* constraints are applied. The initial clauses are simply the clauses for each peripheral in ASL (i.e. the unconstrained total clauses also). No

Constraint Level	None	R	T	P	RT	RP	TP	RTP
<i>Execution Time(mSec)</i>	1032	2523	682	300	1332	2464	501	1352
<i>Total Variables</i>	108	108	108	108	108	108	108	108
<i>Total Literals</i>	108	4542	108	108	4542	4552	118	4552
<i>Total Peripherals</i>	108	6	108	98	6	6	98	6
<i>Total Clauses</i>	6	2223	6	16	2223	2233	16	2233
<i>Initial Clauses</i>	6	6	6	6	6	6	6	
<i>2CNF</i>	0	2217	0	0	2217	2217	0	2217
<i>1CNF</i>	0	0	0	10	0	10	10	10
<i>Horn</i>	0	0	0	0	0	0	0	0
<i>Weakly Positive</i>	0	0	0	0	0	0	0	0
<i>OValid</i>	0	2217	0	0	2217	2217	0	2217
<i>IValid</i>	6	6	6	6	6	6	6	6

Table 10.3: Abstract μ -Law Structural Analysis

Constraint Level	None	R	T	P	RT	RP	TP	RTP
<i>Execution Time(mSec)</i>	561	7331	491	440	3534	3994	801	3173
<i>Total Variables</i>	108	108	108	108	108	108	108	108
<i>Total Literals</i>	108	4542	108	136	4542	4570	136	4570
<i>Total Peripherals</i>	108	6	108	80	6	6	80	6
<i>Total Clauses</i>	6	2223	6	34	2223	2251	34	2251
<i>Initial Clauses</i>	6	6	6	6	6	6	6	6
<i>2CNF</i>	0	2217	0	0	2217	2217	0	2217
<i>1CNF</i>	0	0	0	28	0	28	28	28
<i>Horn</i>	0	0	0	0	0	0	0	0
<i>Weakly Positive</i>	0	0	0	0	0	0	0	0
<i>OValid</i>	0	2217	0	0	2217	2217	0	2217
<i>IValid</i>	6	6	6	6	6	6	6	6

Table 10.4: Refined μ -Law Structural Analysis

Horn clauses are produced by construction. Notice that **Topology constrains whether alone or combined do not contribute to the problem**. You can see this by looking at RT vs. R, P vs. TP, and None vs. T.

Abstract and Refined Comparison

This section just highlights the structural results of the last section for each ASL description. The only significant difference (Input and Output Connects) comes from the refined ASL's extract performance constraints. The results are in Table 10.5.

10.1.3 Comparison between reference configuration

The reference configuration is the configuration described in [16]. This configuration is in Table 10.6 and again shows what the specific peripherals were and where they were placed.

Compared to the CAPE generated configurations the reference design requires one more analog block (9 vs. 8) but this results in a higher precision (8 vs. 6). They both use 1 digital block. The reference design has a total of 9 bytes of RAM (just like CAPE results) but uses 84 more bytes of ROM (403 vs. 319). The main point is that the reference design uses a DAC8 where the CAPE configuration has a DAC6. If the precision was required this could be

Struct. Metric	Ab μ Law	Ref μ Law
Total Variables	108	108
Max Literals	4552	4570
Max Clauses	2233	2251
“Best Config”	RP	RP
Digital Block Range	0-8	0-8
Analog Block Range	8-162	8-162
Input Connect Range	0	1-16
Output Connect Range	0	1-8

Table 10.5: Abstract vs. Refined Comparison for μ -Law

Reference μ -Law						
	Filter	Filter	DAC	AMP	ADC	AMP
	LPF2	LPF2	DAC8	PGA	DELSIG11	PGA
	ASB13	ASB22	ASA21	ACA01	ASB20	ACA00
	ASA12	ASA23	ASB11			

Table 10.6: Reference μ -Law Configuration

specified in an extension of ASL which calls for #Perf specification of precision. This would be a Performance Type 2 constraint and easily implemented.

10.1.4 General Conclusions

The primary conclusion to the μ -Law case study is that the “Abstract” and “Refined” ASL descriptions were the same from a performance standpoint as measured by PET. This means one of two things: (1) They truly are the same from a performance standpoint or (2) PET metrics are not in place which can differentiate them. I believe the truth to be somewhere in the middle. Granted PET does not measure everything that it could, but what it does not measure does not directly translate into performance. Additionally, we only are looking at single runs for each constraint setting for each description. If multiple configurations were generated for each ASL and constraint combination, I would expect more diversity.

The DAC6 vs. DAC8 selection was the key difference between CAPE and the reference design. CAPE uses the DAC6 since the SAT solver determinism seems to assign the lowest integers first (i.e. the first literals made). Due to the way in which literals are constructed, the lower precision devices are the lower integer values. Multiple runs of the SAT solver preventing the last solution would fix this. Alternately, if a heuristic of always trying to get the highest precision was to be implemented, this would just require that lower literals have the highest precision and this would be avoided.

Finally, the bottom line is that both ASL descriptions were satisfiable for all constraint levels and produced designs close to the white paper reference in a fraction of the time. This definitely is a “proof-of-concept” and shows that this approach is very reasonable.

10.2 Pulse Coded Modulation

The second application area that was examined was *Pulse Coded Modulation* (PCM). The following discussion is from [34]. Pulse Coded Modulation refers to a system in which the standard values of a quantized wave are indicated by a series of coded pulses. When these pulses are decoded, they indicate the standard values of the original quantized wave. These codes may be binary, in which the symbol for each quantized element will consist of pulses and spaces: ternary, where the code for each element consists of any one of three distinct kinds of values (such as

positive pulses, negative pulses, and spaces); or n-ary, in which the code for each element consists of any number (n) of distinct values. This discussion will be based on the binary PCM system.

The entire range of amplitude (frequency or phase) values of the analog wave can be arbitrarily divided into a series of standard values. Each pulse of a pulse train takes the standard value nearest its actual value when modulated. The modulating wave can be faithfully reproduced. The amplitude range has been divided into standard values. Each pulse is given whatever standard value is nearest its actual instantaneous value. The greater the number of standard levels used, the more closely the quantized wave approximates the original. This is also made evident by the fact that an infinite number of standard levels exactly duplicates the conditions of nonquantization (the original analog waveform).

For the purpose of decomposing this application there are the following areas:

- Holding Circuit
- Pulse Duration Modulator - **Represented in ASL**
- Gating Circuit
- Clock Pulse Generator - **Represented in ASL**
- Binary Counter - **Represented in ASL**
- Electronic Commutator

10.2.1 ASL description

The ASL descriptions of PCM will describe the Pulse Duration Modulator (ADC DAC), the Clock Pulse Generator (PWM), and the Binary Counter (Timer). These are all in one configuration (hence the one #Module line). The performance constraints will allow the same sampling rates for the DAC and ADC as expressed by the same clock source. The abstract ASL for this is shown in Figure 10.5.

```
#Abstract ASL for PCM application
#Module
(ADC DAC) Timer PWM

#Perf
ADC Clk AClk1
DAC Clk AClk1
;
```

Figure 10.5: Abstract ASL for PCM

The refined ASL in Figure 10.6 simply builds on the abstract by adding a topology constraint and performance constraints for those peripherals involved with the Binary Counter (using a timer) and the Clock Pulse Generator (PWM). The performance constraints will require that these peripherals are now on the same bus for input and output communication.

10.2.2 PET Results

The PET results are based on the same methodology as that for μ -Law Companding described in Section 10.1.2.

```

#Refined ASL for PCM application
#Module
(ADC DAC) (Timer PWM)

#Perf
ADC Clk AClk1
DAC Clk AClk1
Timer Ibus GI1
PWM Obus GO1
;

```

Figure 10.6: Refined ASL for PCM

Best and Satisfiability Analysis

Table 10.7 shows the “Best” configurations for the two ASL descriptions. To note is that for the refined PCM ASL description, TP and RTP were **NOT** satisfiable. This comes from the interaction between the Topology and Performance constraints (notice that RT or T by themselves are satisfiable as are the others). The differences in the configurations involve the ADC selection and placements of other peripherals to accommodate it. Notice also the constraint setting for the “Abstract Best” was RR while the “Refined Best” was RT.

Abstract PCM	All SAT			
Constraints	ADC	DAC	TIMER	PWM
RR	DEL11	DAC6	Timer8	PWM8
	ASB20	ASA10	DBA00	DCA07
Refined PCM	TP and RTP NOT SAT			
Constraints	ADC	DAC	TIMER	PWM
RT	SAR6	DAC6	Timer8	PWM8
	DBA02 DBA03 DBA04	ASA10	DBA01	DBA00

Table 10.7: PCM Best and SAT Analysis

Performance Estimation Numbers

Like μ -Law there is little performance estimation differences between the two descriptions. This again indicates that peripheral selection is not as impactful concerning the metrics in this experiment. Table 10.8 shows the results.

Structural Analysis

Again the structural values for both PCM experiments are shown in Tables 10.9 and 10.10. **Bold** values are the high and low values with at the “earliest” constraint setting as before.

The longest execution time for any of these results is **13.1 seconds**. This is the longest of any experiment but is very reasonable. In addition, notice that Table 10.10 shows the appearance of up to 120 weakly positive clauses. This is the first appearance of these types of clauses. The initial clauses are for each ASL user module (as for μ Law) and the total variables are the same across all experiments as expected.

Perf. Metric	Ab PCM	Ref PCM
Precision	6 bits	6 bits
Input Connects	0	1
Output Connects	0	1
Total DB	3	2
Total AB	2	2
RAM	9 bytes	0 bytes
ROM	249 bytes	249 bytes

Table 10.8: PCM Performance Estimation Comparison

Constraint Level	None	R	T	P	RT	RP	TP	RTP
<i>Execution Time(mSec)</i>	1222	13139	541	681	9434	9935	300	9314
<i>Total Variables</i>	114	114	114	114	114	114	114	114
<i>Total Literals</i>	114	6722	114	138	6722	6746	138	6746
<i>Total Peripherals</i>	114	4	114	90	4	4	90	4
<i>Total Clauses</i>	4	3308	4	28	3308	3332	28	3332
<i>Initial Clauses</i>	4	4	4	4	4	4	4	4
<i>2CNF</i>	0	3304	0	0	3304	3304	0	3304
<i>1CNF</i>	0	0	0	24	0	24	24	24
<i>Horn</i>	0	0	0	0	0	0	0	0
<i>Weakly Positive</i>	0	0	0	0	0	0	0	0
<i>OValid</i>	0	3304	0	0	3304	3304	0	3304
<i>IValid</i>	4	4	4	4	4	4	4	4

Table 10.9: Abstract PCM Structural Analysis

Abstract and Refined Comparison

Table 10.11 shows a quick comparison between the ASL descriptions and their PET results. The increase in the literals for the Ref PCM is from the topology constraint clauses. Notice that the RT configuration is better for the refined as opposed to the R configuration for the abstract. Again this shows the topology clause effect.

10.2.3 Comparison between naive configuration

A naive configuration was generated for this application by Greedily selecting the first User Module for each generic peripheral and then placing it in the first location assigned to it by the IDE. The results of this method are shown in Table 10.12.

Compared to the CAPE generated configurations the naive design requires the same number of analog blocks (2 vs. 2) and they have the precision (6 vs. 6). The naive uses more digital blocks (4 vs. 2). The naive design has a total of 6 bytes of RAM whereas the CAPE results require none. In addition the CAPE configuration uses 153 less bytes of ROM (402 vs. 249). It appears that the CAPE generated solution is superior (as expected to the naive configuration) in all cases (or they are the same).

10.2.4 General Conclusions

Comparing the “Abstract Best” and “Refined Best” again was not very meaningful regarding the metrics generated by PET. However, notice that unlike μ -Law, the best configurations were generated under different constraint settings.

The comparison between the naive and CAPE configurations shows that the CAPE solution saves ROM, RAM, and block resources. The precision is the same. This is again since the naive algorithm greedily selected the

							Not SAT	Not SAT
Constraint Level	None	R	T	P	RT	RP	TP	RTP
<i>Execution Time(mSec)</i>	520	7285	810	360	5723	5261	1134	5727
<i>Total Variables</i>	114	114	114	114	114	114	114	114
<i>Total Literals</i>	114	6722	1802	160	8410	6768	1848	8456
<i>Total Peripherals</i>	114	4	114	68	4	4	0	0
<i>Total Clauses</i>	4	3308	184	50	3488	3354	230	3534
<i>Initial Clauses</i>	4	4	4	4	4	4	4	4
<i>2CNF</i>	0	3304	60	0	3364	3304	60	3364
<i>1CNF</i>	0	0	0	46	0	46	46	46
<i>Horn</i>	0	0	0	0	0	0	0	0
<i>Weakly Positive</i>	0	0	120	0	120	0	120	120
<i>OValid</i>	0	3304	0	0	3304	3304	0	3304
<i>IValid</i>	4	4	4	4	4	4	4	4

Table 10.10: Refined PCM Structural Analysis

Struct. Metric	Ab PCM	Ref PCM
Total Variables	114	114
Max Literals	6746	8410
Max Clauses	3332	3488
“Best Config”	R	RT
Digital Block Range	3-122	3-122
Analog Block Range	2-90	2-90
Input Connect Range	0	1-26
Output Connect Range	0	1-28

Table 10.11: PCM Abstract vs. Refined Comparison

lowest precision device much like how the SAT solver grabs the lowest precision literal to assign first. This could be remedied as described in the μ -Law conclusions. There seems to be a stronger suggestion that the CAPE configuration is “better” than the naive configuration.

Key to note with the PCM results is that of all 4 ASL descriptions (2 PCM, 2 μ -Law), only the refined PCM ASL description produced results in which the Topology constraints generated clauses. This is quickly seen by the fact that *weakly positive clauses* occurred. Also note that the two case which were not SAT (TP and RTP) involved Topology generated clauses. This indicates that naturally the addition of constraints increases the chances that the ASL description cannot be realized.

Like the μ -Law experiments, the PCM experiments again showed that this design flow generates valid configurations that could be used by a designer in a fraction of the time it takes to do the same with the IDE provided by Cypress. In addition they are superior to naive configurations for the same application.

Naive PCM				
	ADC	DAC	TIMER	PWM
	ADCINC12	DAC6	Timer8	PWM8
	DBA00 DBA01 ASA10	ASB20	DBA03	DBA02

Table 10.12: Naive PCM Configuration

Chapter 11

Conclusions

This chapter will describe the conclusions drawn regarding the theoretical framework, the tool development, and the experimental results presented in this report. These are the three major areas in which this investigation gains insight into the process of using Boolean constraints to develop a mapping from specification to implementation for reconfigurable hardware devices.

11.1 Theoretical Conclusions

These are conclusions based simply on the theoretical foundation of CAPE and have to do with the assumptions and algorithms that CAPE is based on. Some of the limitations are fairly trivial while others would change quite a few aspects of the approach.

11.1.1 General Limitations

Naturally the general limitation starts from both the “top” and the “bottom” of the design process. “Top down” limitations result from the expressiveness of ASL. Currently ASL is only concerned with expressing User Modules per configuration, some relationship between peripherals, and some performance constraints. ASL could be extended to have more explicit performance requirements detailing the requirement of specific peripherals, allowing nested “()” relationships, and locking in the configuration execution order. The performance requirement modifications would be fairly easy to implement while the other two aspects would change drastically the semantics and parsing of ASL. More investigation would have to be performed to see if those changes are warranted.

The “bottom up” theoretical issues involve how to accurately model the complex hardware relationships which make up the performance estimation of a particular device. In PET these are called relational metrics and were not implemented for this report. How to model the many different performance interactions is non-trivial. For example to take into account bus loading, power consumption, and context switching overhead requires a much more detailed model and literal encoding scheme. A concern is that in order to accurately do this, more effort is required than actually experimenting with different configurations manually. CAPE has always been concerned with rapid prototyping at the expense of some estimation accuracy.

Finally, based on the examination of the CNF formula produced by ACT, the problem will be NP-Complete. While this is discouraging from a complexity standpoint, this seems to be of no consequence in practice, as the execution times are quite reasonable.

11.1.2 Horn Clauses

No Horn clauses were developed in this investigation. This is not surprising in of itself since the clause creation is formulaic and will not generate random clause structures. The fact that no constraints were developed as Horn clauses simply means that the hope to formulate the problem consisting of Horn clauses in order to attempt to

move the complexity from NP is not an encouraging proposition. Horn clause tracking could be moved out of PET until such clauses are actually going to be created (if that ever occurs).

11.1.3 Constraint Types and Granularity

It would be nice to have constraint levels within the constraint categories themselves. For example for each constraint clause type in each constraint category, there could be a setting just for that. This would increase the axes of design space exploration. Recall in the constraint formulation for Raw Resource constraints there were two types of concerns: one which prevents the over use of resources and the other which prevents the sharing of resources. Currently with Raw Resource constraints in effect, both of those requirements are enforced. The ability to separate them would allow more control over the configurations.

11.2 Tool Development Conclusions

These are simply conclusions based on the completion of the toolset for this report.

11.2.1 Basic PAT

PAT is based on the DIMACs format simply since it works for the SAT solvers used in this exploration. However, alternate PAT constructions might help to make the tool more portable to other environments. Perhaps an internal data structure might be more useful not only for producing alternate formats but also for PET to analyze for more complex metrics.

11.2.2 Basic PET

The current PET results do not include any relational effects as mentioned. A criticism of the current PET results could be that they are trivial. If more information was encoded in the literals then PET could be made more robust. An initial metric which could be added is %Connection Utilization. This would measure how many of the potential pin and bus connections are being used. This could give some indication of the congestion of the device or how much room for expansion is possible. This also indicates how I/O driven the configuration is. This could translate implicitly into a performance estimation by identifying a bottle neck in the system.

11.3 Experimental Results Conclusions

These conclusions are based on the observation of the results in the two case study experiments.

11.3.1 Execution Time

The execution times for the experiments were from **300 milliseconds to 13.139 seconds**. This was the application of ACT and PET. This involved not only the running of the Chaff SAT solver but also writing the DIMACS file (ACT results) and additionally writing the output of the SAT solver (PET results). In addition there are parsing scripts (PERL based) as well as the general control based Java code to set up ACT. I estimate that the run time is I/O limited and not a result of the CNF formulation. In any event 13 seconds is more than acceptable and trivial for the usage scenario (interactive use by a single designer). Again this is with the formulation being NP-Complete.

11.3.2 Clause Generation

As the structural results demonstrate **all the clauses are either 2CNF, 1CNF, Weakly Positive, 0valid, or 1valid**. This is a function of how the constraints were written in CAPE and do not mean that all possible constraints will fit into these categories. There is not a formulation in which all the clauses are of the same type which would put the problem in P . However the regularity and ease of clause classification is promising.

11.3.3 Constraints Mixing and Valid Configurations

Running the tools without Raw Resource constraints would **by default produce configurations which could not be realized on the PSoC**. However running without Raw Resource constraints was useful for determining an upper bound on the implementation and show how much of an improvement the application of constraints resulted in. The ability to mix constraints provided 8 axes of design space exploration which was very rich and sufficient for this type of mapping problem.

11.3.4 SAT Solver Determinism

The **Chaff SAT solver is deterministic**. Therefore repeated applications of it would produce the same configurations. Therefore no experimental results were obtained for multiple runs of a single ASL description and all the corresponding configurations which would result. This could have been done with a *blocking or prevent* clause. This clause would have been as follows:

Example 11.3.1 *If a solution to $(A + B)(C + D)$ is $A = 1$ and $C = 1$ then the addition of the blocking clause $(\bar{A} + \bar{C})$ will prevent $A=1$ and $C=1$ from being a future solution. Continuing to add blocking clauses until the problem is UNSAT will generate a list of configurations.*

Blocking clauses would bypass the difficulty of the solver determinism. I could have also integrated the SATO and BerkMin SAT solvers.

A major issue of the SAT solver Chaff, appears to be the ordering of assigning literals. Lower literal values (i.e. the literal represented by the lower integer value) seem to be set first in Chaff. The lower literal values in CAPE will use the least amount of resources and have the lowest precision. Therefore the “best” configurations tend to minimally use resources. A direct result of this is the fact that 6-bits of precision is the result for all the configurations.

11.3.5 Results

The results for the case studies in general showed the following: **(1)** The abstract and refined ASL specifications did not produce configurations with very different performance characteristics. **(2)** The abstract and refined configurations produced did adhere to the ASL specification which implies information not captured by PET performance metrics. **(3)** CAPE configurations were at least as good if not better than the reference or naive configurations. **(4)** The structural analysis showed the interaction between constraint types and the clauses that they generate in such a way that provided insights during unsatisfiable instances of the problem formulation. All of these areas are very promising for CAPE as a methodology.

11.3.6 It Works!

The most important result of the experimental results is that **CAPE works!** Not only were the configurations that it produced valid and legitimate for the ASL descriptions but it also caught user errors that I introduced while writing ASL descriptions. There were invalid topology requests and invalid performance requests that I attempted to make and actually needed the tool to tell me were unreasonable.

In addition, it gives a **productivity increase**. I conservatively estimate the PSoC development kit takes approximately $\frac{20\text{minutes}}{\text{iteration}}$ where iterations are introduced due to the selection of too many user modules in the “Module Selection View” and the inability to place them in the “Module Placement View”. CAPE takes approximately **5 minutes** with no iterations if ASL is specified correctly.

A concern could be voiced that the user effort will be in “reverse engineering” ASL in order to get a satisfiable result. In the experimental investigation, this was not true. In fact, if ASL was not satisfied it typically indicated that what was asked for was not possible. The current level of constraints was appropriately conservative to prevent user effort in ASL development. However if more constraint types were added, it removes some of the granularity of ASL and will make it harder to be abstract in ASL and get a satisfiable result. An ideal automation would be to successively remove constraints until the instance is satisfiable in the event that the initial result is not.

Chapter 12

Future Work

By no means does the conclusion of this current investigation mean that there is not more work to be done regarding CAPE or investigating reconfigurable architecture configurations. As with all research, the collection of one set of data points inevitably leads to yet another set of unanswered questions. In fact, in the Chapter dealing with the conclusions of this report, several items that are potential future work issues were introduced. This section will detail what the **immediate future work** should consist of and suggest where appropriate steps can be taken to such an end.

12.1 Tool Work

The development of CAPE was based on developing a proof-of-concept framework. However, if the tool were to be used industrially it would need to be fully flushed out and rigorously tested. This section will describe what enhancements would need to immediately occur to fully realize CAPE's potential.

12.1.1 Completion of the CAPE Toolset

This includes providing the remaining keywords for all the PSoC User Modules. Currently only nine peripheral keywords are recognized specific peripherals as described in Section 6.1. There should be a keyword for each User Module type which currently is available. This would include adding DTMF Dialer, CRC, MUX, SPISlave, SPIMaster, RX8, TX8, UART, and FlashTemp minimally. Updating these keywords would have to correspond to new releases of the PSoC development software as Cypress creates more User Modules in their library.

In order to support the additional keywords, the library which describes the possible location of peripherals would have to be augmented. For each peripheral keyword added, the corresponding specific peripheral instance would have to be added along with the locations that are valid for those peripherals. This would be required for creating the literal encoding as described in Section 7.1.

Naturally, adding the blocking clause and iterative SAT solving should also be added. Adding and creating the blocking clause is trivial. The main work is concerned with the file I/O (setting up the input to the SAT solver repeatedly). In addition, there should be an extension to the GUI which allows you to specify a number of configuration generation iterations the user would like to perform (or until exhausted; i.e. UNSAT).

Currently, topological constraints really only concern the digital I/O and analog I/O. Ideally a much more robust criteria of what topological blocks should consist of would be developed to include MUX utilization and on chip level I/O.

Finally, the scheduling formulation should be added. This would require developing ASL descriptions which use multiple #Module lines and the integration of an ILP solver. Metrics would have to be developed to evaluate one scheduler compared to another.

12.1.2 CAPE Extensions

One quick and easy extension of CAPE is in #Perf to allow for explicit literal prevention or requirement. Also, it would be nice to specify a particular precision threshold. This would require all peripherals to meet this requirement. These would both be Performance Type 2 constraints.

Adding the other SAT solvers, SATO and BerkMin, to CAPE would be interesting to see how they differ both in configuration selection and run time as compared to Chaff.

Finally, minor enhancements could be made to the GUI such as a progress bar, batch mode and script support, as well as aesthetic improvements.

12.1.3 ACE Investigation

Notably absent from the results was any information regarding the results of ACE. This was due to the fact that no M8C assembly code was written for either application. Future work will entail the actual development of code to run on the resulting configuration and then an analysis of its running time based on the cycle count returned from ACE. Code could be written for both case studies and also actual implementations could be created to check the actual performance versus the estimated results.

12.1.4 Testing

The completion of testing would require that once CAPE provided a configuration, actually program the device and (1) make sure that the application functions and (2) compare the observed performance with the actual performance. This could be done for the two case studies provided in this report. In addition, testing needs to be done for corner cases of ASL parsing such as strange input, unrecognized keywords, spelling, capitalization, etc.

12.2 Theory Work

Outside of the CAPE toolset there are some theoretical issues that could be explored in the future.

12.2.1 Literal Encoding

Key would be to investigate what the encoding of a literal can reasonably include and use for PET analysis and ACT problem formulation. The focus now is on the specific peripheral instance and where it goes. There are several extensions of this. One is to give an area estimate of the peripheral or a power consumption value. Also literals could be made more coarse to represent entire configurations or more granular to recognize the individual configuration registers.

12.2.2 Generalizing to Reconfigurable Computing

Finally a discussion should include how to extend this framework to the reconfigurable computing community in general. In general, any reconfigurable device has some primitive which is under the control of the user to program. What is necessary is to decide upon how those primitives make up a configuration. Once it is determined how to combine those primitives into a configuration, the next step is how to encode those primitives into a Boolean variable. Typically this variable will be present/not present. For example in an FPGA there could be a variable for each LUT and their 4-input function. So if there were 3 LUTs for example there would be 48 variables. A configuration would then be to select one variable which represents the setting of a LUT. Additionally, interconnect and memory resources would also need an encoding. The main concerns for this formulation naturally are: (1) A large number of literals may result, (2) a large number of clauses may be created, or (3) very irregular clause structures could be present. The effort in the problem formulation will be to avoid as many of these issues as possible. If that can be accomplished, the additional problem of accurate performance estimation is still a major obstacle. High level metrics for narrowing down an application to a set of configurations is most likely to be successful in this area.

Chapter 13

Appendix

This section just provides some extra material to show what the designer deals with when running CAPE. There is little to no explanation here. For more details the reader should refer to the appropriate sections in the body of the report. The documentation within the toolset itself should also be referenced for specific information regarding running the tool.

This is the output of the Chaff SAT solver run during PET. The literal integers without a “-” sign represents the selected configuration. A script is used to parse out those literals for PET to interpret.

```
Z-Chaff Version: ZChaff 2003.7.1
Solving temp/act_results.txt .....
2225 Clauses are true, Verify Solution successful. Instance satisfiable
1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 14 -15 -16 17 -18 -19 -20 -21 -22 -23
-24 -25 -26 -27 -28 -29 -30 -31 -32 -33 -34 -35 -36 -37 -38 -39 -40 -41 -42 -43
-44 -45 -46 -47 -48 -49 -50 -51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63
64 -65 -66 -67 -68 -69 -70 -71 -72 -73 -74 -75 -76 -77 -78 79 -80 -81 -82 -83 -
84 -85 -86 -87 -88 -89 -90 -91 -92 93 -94 -95 -96 -97 -98 -99 -100 -101 -102 -
103 -104 -105 -106 -107 -108
Max Decision Level 96
Num. of Decisions 134
Num. of Variables 108
Original Num Clauses 2223
Original Num Literals 4542
Added Conflict Clauses 2
Added Conflict Literals 19
Deleted Unrelevant clause 0
Deleted Unrelevant literals 0
Number of Implication 162
Total Run Time 0
```

SAT

The following shows the information provided after running PET. This corresponds to the results of Chaff shown previously. The literal information is for the literals selected and the end of the report provides the structural and performance information. This would be done for each case study experiment in the report. The ASL is already provided in the body of the report.

```
PET Results
New and improved PET
```

Literal Int = 1
General Periph = FILTER
Specific Periph = BPF2
Location 0 = ASB20
Location 1 = ASA21
Digital Blocks = 0
Analog Blocks = 2
Ram = 0
Rom = 29
Latency = 2.0
Throughput = 2.0
Precision = -1
Type1 = false
Type2 = false
Req =

Literal Int = 14
General Periph = FILTER
Specific Periph = LPF2
Location 0 = ASB10
Location 1 = ASA11
Digital Blocks = 0
Analog Blocks = 2
Ram = 0
Rom = 29
Latency = 2.0
Throughput = 2.0
Precision = -1
Type1 = false
Type2 = false
Req =

Literal Int = 17
General Periph = DAC
Specific Periph = DAC6
Location 0 = ASA10
Digital Blocks = 0
Analog Blocks = 1
Ram = 0
Rom = 47
Latency = 2.0
Throughput = 2.0
Precision = 6
Type1 = false

Type2 = false
Req =

Literal Int = 64
General Periph = AMP
Specific Periph = AMPINV1
Location 0 = ACA01
Digital Blocks = 0
Analog Blocks = 1
Ram = 0
Rom = 32
Latency = 2.0
Throughput = 2.0
Precision = -1
Type1 = false
Type2 = false
Req =

Literal Int = 79
General Periph = ADC
Specific Periph = SAR6
Location 0 = DBA00
Location 1 = DBA01
Location 2 = DBA02
Digital Blocks = 0
Analog Blocks = 1
Ram = 0
Rom = 56
Latency = 2.0
Throughput = 2.0
Precision = 6
Type1 = false
Type2 = true
Req = ACLK2

Literal Int = 93
General Periph = AMP
Specific Periph = AMPINV1
Location 0 = ACA00
Digital Blocks = 0
Analog Blocks = 1
Ram = 0
Rom = 32
Latency = 2.0
Throughput = 2.0

Precision = -1
Type1 = false
Type2 = false
Req =

Execution Time = 4597 milliseconds
Total Variables = 108
Total Literal Appearances = 4542
Total Peripherals Used = 6
Total Clauses = 2223
Initial Clauses = 6
2CNF = 2217
1CNF = 0
Horn = 0
Weakly Positive = 0
Ovalid = 2217
1valid = 6

Execution Latency = 0
Execution Throughput = 0
Precision of Operands = 6
Input Bandwidth = 0
Output Bandwidth = 0
Total Digital Blocks (MAX config) = 0
Total Analog Blocks (MAX config) = 8
Total ROM Memory Used (MAX config) = 225
Total RAM Memory Used (MAX config) = 0

Bibliography

- [1] Luciano Lavagno Alberto Sangiovanni-Vincentelli, Marco Sgroi. Formal models for communication-based design. In *Proceedings of CONCUR '00*, August 2000.
- [2] Altera. *Altera FGPAs*. World Wide Web, <http://www.altera.com>, 2004.
- [3] Anadigm. *Anadigm FPAs*. World Wide Web, <http://www.anadigm.com/>, 2004.
- [4] A.Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign*, February 2002.
- [5] Kiran Bondalapati, Pedro C. Diniz, Phillip Duncan, John Granacki, Mary Hall, Rajeev Jain, and Heidi Ziegler. DEFACTO: A design environment for adaptive computing technology. In *IPPS/SPDP Workshops*, pages 570–578, 1999.
- [6] Kiran Bondalapati and Viktor K. Prasanna. Reconfigurable computing systems.
- [7] G. Brebner. A virtual hardware operating system for the xilinx xc6200. In *International Workshop on Field Programmable Logic*, September 1997.
- [8] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, Yury Markovskiy, John Wawrzyniek, and Andre DeHon. Stream computations organized for reconfigurable execution (score): Introduction and tutorial. In *Proceedings of the 10th International Conference on Field Programmable Logic and Applications*, August 2000.
- [9] Codetelligence. *Codetelligence CodePalette*. World Wide Web, <http://www.codetelligence.com/products.htm>, 2004.
- [10] Nadia Creignou, Miki Hermann, and R. Pichler. Complexity of constraint satisfaction problems.
- [11] Andre DeHon. The density advantage of configurable computing. In *IEEE Computer*, April 2000.
- [12] Douglas Densmore, Sanjay Rekh, and Alberto Sangiovanni-Vincentelli. Microarchitecture development via metropolis successive platform refinement. In *Design Automation and Test Europe (DATE)*, February 2004.
- [13] Srinivas Devadas, Abhijit Ghosh, and Kurt Kuetzer. *Logic Synthesis*, chapter 3: Two Level Combinational Circuits. McGraw Hill, 1994.
- [14] Merriam-Webster Online Dictionary. *configuration*. World Wide Web, <http://www.merriam-webster.com> (1 Aug. 2003), 2002.
- [15] Merriam-Webster Online Dictionary. *reconfigured*. World Wide Web, <http://www.merriam-webster.com> (12 Aug. 2003), 2002.
- [16] David Van Ess. Logarithmic signal companding. In *Cypress Microsystems App Note AN2095*, 2003.
- [17] International Technology Roadmap for Semiconductors. *2002 Update ITRS*. <http://public.itrs.net/>, 2002.
- [18] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.

- [19] E. Goldberg, M. Prasad, and R. Brayton. Using sat for combinational equivalence checking, 2000.
- [20] J. Hadley and B. Hutchings. Design methodologies for partially reconfigured systems. In Peter Athanas and Kenneth L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 78–84, Los Alamitos, CA, 1995. IEEE Computer Society Press.
- [21] Reiner W. Hartenstein and Herbert Grünbacher, editors. *Field-Programmable Logic and Applications, The Roadmap to Reconfigurable Computing, 10th International Workshop, FPL 2000, Villach, Austria, August 27-30, 2000, Proceedings*, volume 1896 of *Lecture Notes in Computer Science*. Springer, 2000.
- [22] Soha Hassoun and Tsutomu Sasao, editors. *Logic Synthesis and Verification*, chapter 12 SAT and ATPG: Algorithms for Boolean Decision Problems. Kluwer Academic Publishers, 2002.
- [23] Holger H. Hoos and Thomas Sttze. Satisfiability suggested format. In *I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000*, pages 283–292, 2000.
- [24] Cheng-Tsung Hwang, Jiahn-Hung Lee, and Yu-Chin Hsu. A formal approach to the scheduling problem in high level synthesis. In *IEEE Transactions on Computer-Aided Design*, volume 10, 1991.
- [25] Intel. *Intel Flash Memory*. World Wide Web, <http://www.intel.com/design/flash/>, 2004.
- [26] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. In *IEEE trans on Computer-Aided Design*, volume 19, December 2000.
- [27] Kurt Kuetzer. Programmable platforms will rule. *EETimes*, September 2002.
- [28] Tracy Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, 1992.
- [29] Cypress Microsystems. *CY8C25xxx/26xxx Device Data Sheet Version 3.21*. <http://www.cypressmicro.com/pdf/8C25KDataSheet.pdf>, November 2002.
- [30] Cypress Microsystems. *PSoC Designer: Assembly Language User Guide*. <http://www.cypressmicro.com/pdf/AssemblerUserGuide.pdf>, October 2002.
- [31] Cypress Microsystems. *Cypress Microsystems Home Page*. World Wide Web, <http://www.cypressmicro.com/corporate/corporate.htm>, 2004.
- [32] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
- [33] Ralf Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*, chapter 4: Hardware/Software Partitioning. Kluwer Academic Publishers, 1998.
- [34] Integrated Publishing. Communications pulse modulators. <http://www.tpub.com/neets/book12/491.htm>.
- [35] David Robinson, Gordon McGregor, and Patrick Lysaght. New CAD framework extends simulation of dynamically reconfigurable logic. In Reiner W. Hartenstein and Andres Keevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm*, pages 1–8. Springer-Verlag, Berlin, / 1998.
- [36] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in forsyde. In *IEEE Transactions on CAD*, January 2004.
- [37] Patrick Schaumont, Ingrid Verbauwhede, Majid Sarrafzadeh, and Kurt Keutzer. A quick safari through the reconfigurable jungle. In *Design Automation Conference*, June 2001.

- [38] M. Sima, S. Vassiliadis, S. Cotofana, J. van Eijndhoven, and K. Vissers. A taxonomy of custom computing machines, 2000.
- [39] D. Smith and D. Bhatia. RACE: Reconfigurable and adaptive computing environment. In Reiner W. Hartenstein and Manfred Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, pages 87–95. Springer-Verlag, Berlin, 1996.
- [40] Xilinx. *Virtex II Pro Platform FPGA Handbook*, ug120 (v2.0) edition, October 2002.
- [41] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97)*, volume 1249 of *LNAI*, pages 272–275, 1997.