Platform Based Reconfigurable Hardware Exploration via Boolean Constraints

Iyibo Jack
Summer Undergraduate Program in Engineering Research Berkeley 2003
Faculty Mentor: Alberto Sangiovanni-Vincentelli
Graduate Mentor: Douglas Densmore

1University of Washington 2University of California, Berkeley
Abstract—As technology advances, the need for having recon-
figurable devices becomes more of a necessity due to the growing
complexity of applications. Programmable devices like Cypress
Semiconductor’s Programmable System-on-a-Chip (PSoC) allow
its users the flexibility to program one device that meets many
of their application needs. Being that this technology is becom-
ing more wide spread, it is now time to research a higher level
of abstraction for programmable devices so that they can be even
more versatile and robust. This paper will demonstrate how the
programming of reconfigurable devices are affected when they are
constrained primarily on a system’s physical specifications. This
results in a Boolean constraint formulation which can be framed
as a classic CNF satisfiability problem.

I. INTRODUCTION

Time to market pressures, design complexity, and cost of
ownership (Non-Recurring Engineering costs) are all hurdles
that manufactures of electronic devices are facing now and will
continue to face in the future [1]. New design methodologies
though, are being researched and implemented to help alleviate
this current situation. Reconfigurable devices are being used
more frequently due to the complexity of applications in addi-
tion to the issues mentioned previously. One methodology is
to apply a Platform Based Design (PBD) approach to recon-
furbable devices. This approach will provide industry with a new
tool to assist in their design of applications.

In this paper, Cypress MicroSystems’s Programmable
System-on-a-Chip (PSoC) was used to explore how the pro-
gramming of reconfigurable devices are affected when they are
constrained primarily on a system’s physical peripheral speci-
fication. In accomplishing this, an Application Speciﬁcation Lan-
guage (ASL) was created. This language is parsed and put into a set
of robust data structures. These data structures are then ma-
nipulated to determine all of the possible PSoC device conﬁg-
urations based on ASL peripheral requirements. This represen-
tation is referred to as the unconstrained Platform Abstrac-
tion Text (RAW PAT). Constraints speciﬁed in ASL are added
to RAW PAT resulting in PAT. PAT is the platform abstraction
portion of PBD which will be used in design space exploration
of the PSoC.

II. OBJECTIVES

The goals for this project are as follows:

- Create an Application Speciﬁcation Language (ASL)
  This requires that we deﬁne both the semantics and syntax.
  In addition, construct graphical user interface (GUI) front end.

- Parse ASL and create an unconstrained Platform Abstrac-
tion Text (RAW PAT)
  In doing this an efﬁcient data structure should be made.
  These newly created data structure will be used in the Pars-
ing of ASL, and in the System, & Constraint libraries.

- Apply an Application Constraint Transform (ACT) us-
ing Boolean Constraints to augment RAW PAT
  This is the top down portion of Platform Based Design
  methodology.

- Result is a CNF formula that possesses the possible con-
  ﬁgurations of the device.

Using CNF satisfiability one can enumerate device conﬁ-
gruations and use one that best ts the users requirements.
This is the design exploration phase.

A. Organization of the Paper

The rest of the paper will be organized as follows: Section III
will provide background on the PSoC and its role in this inves-
tigation. Section IV gives background on the Platform Based
Design methodology. Sections V, VI, VII all provide infor-
mation regarding the methodology and tools developed for this
project. Sections VIII and IX provide conclusions and future
work respectively. Lastly section X is the place where people
that assisted in the project are recognized.

III. PSoC BACKGROUND

Cypress MicroSystems’s Programmable System-on-a-Chip
is a dynamically (at runtime) reconfigurable device [3]. It con-
ists of flash memory, SRAM memory, M8C 8-bit microcon-
troller, analog, and digital blocks. The changing of the conﬁ-
urations of the analog and digital blocks is the primary contrib-
tor to the reconﬁgurability of the device. Ampliﬁers and ﬁlters
are examples of analog block types while counters and timers
are examples of digital type blocks. The general orientation of
the PSoC block types are as follows:

- Analog Blocks - 12 Analog blocks; 72 registers; 8 bits
  each; There are three types of analog blocks. They are
  continuous time analog blocks and two sets of switched
  capacitor blocks ("A" and "B"). They can be positioned in
  locations labeled ACA(X), ASA(X), and ASB(X).

- Digital Blocks - 8 digital blocks; 56 registers; 8-bit each;
  These blocks are 8-bit peripherals and are conﬁgured via
  their function, input, output, data, and control registers.
  They can be positioned in locations labeled DCA(X) or
  DCA(X) which stand for Digital Basic (DB) and Digital
  Communication (DC).

Figure 1 [3] shows the microarchitecture of the PSoC. In par-
icular notice the presence of the analog and digital block arrays
as well as the M8C microcontroller which coordinates on-chip
activities.

IV. PLATFORM BASED DESIGN OVERVIEW

Platform based design (PBD) [1] is a top down, bottom up,
m“meet in the middle” approach to designing applications. It in-
many ways resembles two funnels. A top funnel constraining
applications as they propagate to lower abstraction levels, while
a bottom funnel constrains implementation instances as they
propagate up it. An example of our top down approach of PBD
is demonstrated in ﬁgure 2. In this speciﬁc project we focused
primarily on the top portion (funnel/triangle) of PBD. This area
is referred to as the Application Space. The Application Space
is an abstract area in which the fundamental requirements of
an application can be deﬁned. From the Application Space,
ASL is implemented as a means to describe the requirements
of the application. Once ASL has been written ACT is applied
on ASL (top down constraints) thus resulting in PAT (platform
representation; “meeting in middle”).
V. ASL

There was a need to abstractly construct a way for the user to have the ability to specify designs without adding unnecessary bias which would lead the design to one particular implementation (PSoC configuration). To meet this need ASL was formed. ASL is a netlist type language that allows the user the ability to describe applications abstractly in terms of the peripheral requirements. In ASL, the designer has the ability to specify both performance and relationship constraints for peripherals. Being that the PSoC was used, we setup ASL so that it would be defined around peripherals (user modules) each related to each other in terms of the PSoC’s configuration page. A configuration page is one particular settings of the set of configuration registers. Because of this, ASL does not work at the level of the switched capacitors or logic gates but instead it recognizes ADCs, DACs, etc. There are 8 peripheral types that are defined in ASL: Counter, Timer, DAC, ADC, PWM, PSR, AMP, and Filters (ADCs are not fully defined in ASL since not all permutations were included). ASL is broken into 2 sections #Module and #Perf. #Module is followed by peripherals, which are connected to one another by it through input/outputs, shared clocks, or connected to the same bus are denoted by being encompassed by parenthesis “()”. #Perf has the syntax of: (<Peripheral name> <Peripheral performance type> <Peripheral performance>). In this project #Perf was not fully developed in detail. An example of ASL is shown in figure 3:

A. Parsing ASL

ASL was originally parsed by a program written in Perl but for this project it was redone using Java. 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.

1) #Module: 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 <Peripheral> per = new <Peripheral> and every peripheral read in is used in the constructor to make the new peripheral object. This new object is stored into a linkedlist. An arraylist is also created to store the keys of this object. These keys are necessary because the linkedlists are eventually added to a hashtable. This process of creating a new peripheral object and adding it to a linkedlist is O(n) where n is the number of linked peripherals. Adding these linkedlists to a hashtable has a run time of O(x) where x 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 peripherals on a specific line.

If a singular peripheral is read the same process as the linked peripherals is done except their is no need to strip the parenthesis off. All together after taking into account adding peripheral objects to a linkedlist (O(n)), adding linkedlists to a hashtable (O(x)), and having one major loop that is reading ASL from the file, O(ASL Lines) #Module has a runtime approximately of O(n^3) since n is the dominating factor.

See figure 4 for a visual representation of how #Module was added to the parsing data structure.
2) #Perf: #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 of `<peripheral name>`, `<peripheral type>`, and `<peripheral performance>`. The keys from the hashtable in the #Module section are used to properly index the corresponding lines in #Module. The linkedlists 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 linkedlist from the hashtable is \( \mathcal{O}(1) \) (constant). The process of adding the information to the element in the list is \( \mathcal{O}(n) \) 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 below in figure 5.

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

![Peripheral Object]

Peripheral Object

<table>
<thead>
<tr>
<th>Peripheral Name</th>
<th>Peripheral Performance Type</th>
<th>Peripheral Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

Fig. 5. #Perf Implementation

B. ASL Graphical User Interface (GUI)

A major reason for why ASL was parsed using Java instead of Perl was that making GUI’s in Java are significantly more natural than in most other languages. A GUI was decided to be part of this project because it provides the user a friendly environment. The GUI attached to the program gives the user a window where there are three button labeled File, Run, and Help respectively. The GUI also contains other options. A high level implementation of this is as follows:

```java
< JFrame frame = new JFrame(); >
// new JFrame frame
< Container cp = frame.getContentPane(); >
// Container cp initialized
< JButton button = new JButton("File"); >
// button labeled File made
< cp.add(button); >
< frame.pack; >
// sets the JFrame sized
< frame.setVisible(true); >
// lets JFrame be seen
```

If the File button is selected it opens up a file chooser menu where the user can select and open an ASL file. The Run button actually starts the parser and the rest of the program.

VI. RAW PAT

RAW PAT is acquired after ASL has been read, parsed, and put into a data structure. It is an unconstrained Platform Abstraction Text that list all possible configurations strictly from ASL peripheral. This includes configurations that can’t occur due to the ASL peripheral constraints. True PAT will take these constraints into account as will be discussed in section VII.

The result of RAW PAT is a CNF formula [2] such as \((A + B + C)(D + E + F)\). What each clause represents is a possible configuration choice for each peripheral specified in ASL. Therefore, in order to satisfy this function, one peripheral assignment must be made. For example, one could choose A and D or B and F, etc. The key to note is that the equation cannot be satisfied without selecting at least one literal per clause.

As a side note, it should be mentioned that each literal is actually a cube. For example, \(A = Timer16\#DBA01\#DBA02\). This actually will cause this to no longer be a true CNF formula. However, the literal representation is just an abstraction and in the future a look-up-table will store the true literal information.

A. System Library

In generating RAW PAT there was needed a way to make a library of all the possible types of peripherals and their locations. The library needed to be comprehensive but also not too costly with respects to time traversing it. It should also have the ability to add peripheral performance information for the bottom up portion of the toolset (to be implemented later). With these factors in mind the figure below depicts graphically how the library was designed. See figure 7. A hashtable has keys that represent the literal peripheral type. Here is essentially how this data structure was created:
<hashtable table = new hashtable()>
<table.put("DAC", makeList("DAC"))>
“makeList” is a method that stores all of the specific types of a peripheral in a linkedlist. The example above would have a makeList like :
<linkedList list = new LinkedList()>
<list.add(new blockinfoDAC6)>
<list.add(new blockinfoDAC8)>....... “blockinfo is an object which has the actual possible locations of the PSoC blocks. Linkedlists contain specific block positions and those positions are stored in an arraylist. Ex:
<linkedList list2.add(DBA00)>
<Arraylist array.add(list2)>

Class blockinfo calls blockentry which is an object that neatly contains the information that blockinfo possesses. The blown up square in figure 7 depicts this.

VII. PAT

PAT is the combination of RAW PAT plus the constraints from the ACT. ACT is formulated from the constraint library. RAW PAT is outputted in the form of clauses: \((A_1 + A_2 + A_3)(B_1 + B_2 + B_3)\) as mentioned previously. All that is needed to satisfy this expression is to have one value in each clause to be true. This yields many possible configurations because each combination represents a configuration. PAT though, has additional clauses that RAW PAT does not. When these constraint clauses are introduced it adds more additional conditions that limit configurations.

Ex: \(\text{PAT} = \text{RAW PAT} + \text{Constraint Clauses} \)

\((A_1 + A_2 + A_3)(B_1 + B_2 + B_3)(\overline{A_1} + B_2)(A_3 + B_2)\)

Now notice that the using A1 and B2 or A3 and B2 are no longer two valid selections of peripheral configurations. The constraint clauses effectively reduce the number of possible configurations. These clauses will represent the additional information in ASL provided by the "( )" and the #Perf sections.

These keywords generate constraint clauses for the CNF formulation. For example if two devices are associated in parenthesis such as \((\text{DAC ADC})\), then constraint clauses can be generated which require that if both peripherals are on separate buses they cannot be connected together. These constraint clauses are based on the development of a constraint library which is discussed in the next section.

The overall goal, will be the take this CNF formulation and find a satisfying assignment for it. The set of satisfying assignments are the possible configurations. Once the system library is augmented with performance information, each configuration can be evaluated automatically and therefore, all configurations can be exhaustively analyzed.

A. Constraint Library

In this project the constraint library was not fully developed for both the analog and digital blocks. Only the digital blocks were fully completed. The essence behind the constraint library was a data structure that can be used to show what types of connections/configurations cannot be made due to the systems physical constraints. For this project it was decided that only digital blocks on the same bus would be able to be connected to one another. The constraint library is no more than a class that has two arraylist which return linkedlists of the block locations that can occur.

VIII. CONCLUSION

This project exhaustively constructed a Boolean formulation of a PSoC constrained application. In having this approach, the maximum amount of information about the reconfigurability of the device is known. While this may have to be pruned heuristically for performance reasons, the key is that all the information is there which will provide maximum implementation flexibility. The implementation of the upper half tools for
design exploration with the PSoC was also accomplished which included the GUI, ASL, ASL parser, System, and Constraint libraries. Efficient data structures were also made in this project to increase the program’s operational speed. This provides a solid theoretical foundation on which to complete the rest of the toolset.

Fig. 9. Future Toolset Flow

IX. FUTURE WORK

Continuing research in the project would include completing the Performance Estimation of the PBD (see figure 9). PAT would have to be able to produce multiple configurations with varying performance levels. This would be done with a commercial CNF SAT solver. New heuristics will have to be made to reduce the size of the variables when trying to constrain RAW PAT to PAT. More complex constraints will also need to be developed so that #Perf portion of ASL will be put into consideration when trying to determine PAT. When this is all accomplished sample applications using the PBD will need to be constructed to actually apply new design techniques and measure actual effectiveness.

X. ACKNOWLEDGEMENTS

Without the efforts of many individuals this project would not have happened. I give thanks to the Superb-IT coordinators for giving me an opportunity to do research at one of the most premiere universities in the country. I am also thankful for the assistance the people at the DOP Center gave me, they were very accommodating. I am very appreciative for Professor Alberto Sangiovanni-Vincentelli for letting me to work in his group with his graduate students. A special thanks though, goes out to my graduate mentor Douglas Densmore. He was the one that worked with me the most and without him this project surely would not have been accomplished.

REFERENCES