Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Optimization Of Error Detection In Embedded Systems

By

Syed Muhammad Hassan

LIU-IDA/LITH-EX-A—10/053—SE
Final Thesis

Optimization Of Error Detection In Embedded Systems

by

Syed Muhammad Hassan

LIU-IDA/LITH-EX-A—10/053—SE

Supervisor: Adrian Lifa and Petru Eles
Examiner: Petru Eles (ESLAB)
ACKNOWLEDGMENT

I would like to thank my advisors Petru Eles, Zebo Peng and especially to Adrian Lifa who guided me and supported me during the work. I am also very thankful to my sisters and my parents who taught me that best kind of acquaintance to have is that which is erudite for its own sake.

They taught me that the largest task can be accomplished if it is done one step at a time. Without their love, prayers, encouragement and support it was really difficult to complete this work.
Abstract

This thesis deals with algorithms that optimize the implementation of the error detection technique for soft real-time and multimedia applications in order to minimize their average execution times. We aimed to design the algorithms such that with little hardware available we could achieve maximum time gain.

In the context of electronic systems implemented with modern semiconductor technologies transient faults have become more and more frequent. Factors like high complexity, smaller transistor sizes, higher operational frequencies and lower voltage levels have contributed to the increase in the rate of transient faults in modern electronic systems.

As error detection is needed, no matter what tolerance strategy is applied, the detection mechanisms are always present and active. Unfortunately, they are also a major source of time overhead. In order to reduce this overhead designers try to implement error detection in hardware, but this approach increases the overall cost of the system. In general there are three approaches to implement the error detection technique. One extreme implementation involves software only and another extreme implementation involves hardware only. But we focus on the mixed one which involves both hardware as well as software, in order to generate the best system performance with minimal costs.

To reduce the time overhead and to achieve maximum time gain, we place as much as possible of the checking expressions in hardware depending on the available resources. The decision is taken based on the frequency information obtained from an execution profile of the program.

To achieve our goal we have formulated the problem as a knapsack problem, for which we proposed two algorithms. The first one is a greedy approach and the second one finds the optimal solution using dynamic programming.

To compare the result for these two algorithms and evaluate their efficiency we have run a series of experiments considering applications with different number of detectors and checking expressions. We have also run our optimization on a real-life application (GSM
encoder) to see how these algorithms perform under real-life scenarios. The experimental evaluation has proved the efficiency of our algorithms under different scenarios.

Our optimization reduces the time overhead incurred by the error detection component in systems with tight resource constraints. The results presented in this thesis can be used as a foundation for future research in the area. For example, our algorithms could be extended to consider partially dynamically reconfigurable FPGAs or they could be extended so that they give probabilistic guarantees (and could be used in hard real-time systems).
# Table of Contents

1 Introduction ........................................................................................................................................ 1  
1.1. Purpose........................................................................................................................................ 1  
1.2. Motivation....................................................................................................................................... 1  
1.3. Scope ............................................................................................................................................ 2  
1.4. Contribution ................................................................................................................................. 2  
1.5. Thesis Overview ............................................................................................................................ 3  
2 Background ......................................................................................................................................... 5  
2.1. Error Detection Technique .......................................................................................................... 5  
2.1.1. Preliminaries ............................................................................................................................. 5  
2.1.2. Definitions ............................................................................................................................... 6  
2.1.3. Implementation ........................................................................................................................ 6  
2.2. Fault Model .................................................................................................................................... 10  
2.3. General Idea .................................................................................................................................. 11  
3 System Model ...................................................................................................................................... 13  
3.1. Application Model ....................................................................................................................... 13  
3.2. System Architecture .................................................................................................................. 15  
4 Problem Formulation ........................................................................................................................ 17  
4.1. Input ............................................................................................................................................. 17  
4.2. Output .......................................................................................................................................... 18  
5 Optimization Algorithm .................................................................................................................... 19  
5.1. Knapsack Problem ....................................................................................................................... 19  
5.1.1. Greedy Approach ...................................................................................................................... 20  
5.1.2. Dynamic Programming Approach .......................................................................................... 23
6 Experimental Evaluation ................................................................. 27
   6.1. Experimental Setting .................................................................. 27
   6.2. Applications With 20 Checking Expressions .................................. 29
   6.3. Applications With 100 Checking Expressions ............................... 32
   6.4. Applications With 400 Checking Expressions ............................... 35
   6.5. Case Study: GSM Encoder .......................................................... 38
7 Conclusion ......................................................................................... 41
References ............................................................................................ 44
Chapter 1

Introduction

1.1. Purpose
The purpose of this thesis is to optimize the implementation of the error detection for soft real-time and multimedia applications and our target is to minimize the average execution time. The main goal is to co-design the error detection mechanisms in an optimized way, such that the average execution time is minimized and the hardware cost constraints are met.

1.2. Motivation
The usage of electronic devices has increased tremendously in past few decades. Nowadays we are using more and more electronic device and our dependency upon them is growing fast, because the usage of these systems makes our work more efficient and our life more comfortable. In these modern systems the technology is shifting more towards the digital systems using nanometer technology. But there is always the chance of a system failure, since these systems always have some vulnerable points at which faults could occur. These faults could cause errors in the system. These errors could propagate in the system and cause system failure.

There are many factors which could introduce faults in the system and make our system fail. These faults could be permanent, transient or intermittent. Permanent faults cause long-term malfunctioning of components. Transient and intermittent faults appear for a short time. Causes of intermittent faults are within system boundaries, while causes of transient faults are
external to the system. The effects of transient and intermittent faults, even though they appear for a short time, can as devastating similar to the effects of permanent faults. They may corrupt data or lead to logic miscalculations, which can result in a fatal failure.

Factors like high complexity, smaller transistor sizes, higher operational frequencies and lower voltage levels have contributed to the increase in the rate of transient and intermittent faults in modern electronic systems [Con03]. Thus, in this thesis we focused on transient and intermittent faults (also known as soft errors).

In order to prevent the program from erroneous output or getting it crash, different techniques are used. These techniques detect, which may be caused by transient or intermittent faults. Unfortunately, when we include error detection techniques in a program this will increase its execution time. If an error is detected while executing the program, we could stop the error propagation and prevent the system from failure.

In general, there are three different approaches to implement error detection. One is by implementing the error detection technique in hardware. This will increase the hardware cost of the system, but reduce the time overhead. The second approach involves software i.e. implementation of the error detection is done in software. Although there are no hardware costs incurred, the time overhead might be inacceptable because the performance of the program will be drastically degraded. The third approach involves both hardware as well as software i.e. implementation of the error detection is done with a mix of hardware and software components.

1.3. Scope

The primary focus of this research is to optimize error detection in such a way that the time overhead could be minimized, especially when the availability of hardware resources is limited. As mentioned earlier, we focus on optimizing the error detection mechanism for soft real-time and multimedia applications, for which it makes sense to minimize the average execution time.

1.4. Contribution

The main contributions of this thesis are:

• **An optimization has been proposed to implemented error detection**, based on a mixed approach, using greedy approach for the placement of the checking expression, considering only static reconfigurable FPGAs.

• **An optimization has been proposed to implemented error detection**, based on a mixed approach, using a dynamic programming approach for the placement of the checking expressions, considering only static reconfigurable FPGAs.
1.5. Thesis Overview

The thesis is structured as follows:

- **Chapter 2** presents background information about the error detection technique [[Pat07a], [Pat07b]] used in our work and we present the fundamentals behind this error detection approach.

- **Chapter 3** presents our hardware architecture and application model explaining them in detail.

- **Chapter 4** discusses our problem formulation. It gives details about the input parameters for our implementation and also discusses the output that we have generated.

- **Chapter 5** introduces our approaches for error detection optimization. We have formulated the problem as a knapsack problem, for which we proposed two algorithms. The first one is a greedy approach and the second one finds the optimal solution using dynamic programming.

- **Chapter 6** presents the experiments that we have performed on our approaches. We have also run our approaches on a real-life example, to study their performance.

- **Chapter 7**, finally, presents our conclusions.
Chapter 2

Background

The error detection technique used in our approach was first proposed by Pattabiraman et al. in [Pat07]. The following chapter will provide details about this specific error detection technique and how it works.

2.1. Error Detection Technique

2.1.1. Preliminaries

Current embedded systems have high performance requirements and tight cost constraints. At the same time, the applications running on these systems have fault tolerance requirements. But in order to tolerate faults, they have to be detected first. In this context researchers have proposed various error detection techniques [Blo06, Bol08, Hu09, Rei05]. In recent years, the concept of application-aware error detection has been proposed and implemented [Pat07a, Pat07b, Pat07c].

The concept of the application-aware error detection comes as an alternative to the traditional one-size-fits all approaches. Application-aware techniques try to make use of the knowledge about the application’s characteristics. This will produce a customized solution, tuned to better suit each application’s needs.

The authors of [Pat07c] report that it is possible to achieve 75%-80% error coverage for crashes by checking the five most critical variables in each function on a set of representative
benchmarks. Fault injection experiments conducted in [Pat05] showed that the above technique can reach a coverage of 99% by considering 25 critical variables for a perl interpreter.

Since this technique compares favourably with other traditional techniques (like for example full duplication\(^1\), we have chosen to use it in our work. The next section will describe how the technique works.

### 2.1.2. Definitions

Before describing how the technique works, let us define some of the concepts needed to understand the rest of the section.

**Backward Program Slice** of a variable at a program location is defined as the set of all program statements/instructions that can affect the value of the variable at that program location [Pat07b].

**Critical variable:** A program variable that exhibits high sensitivity to random data errors in the application is a critical variable. Placing checks on critical variables achieves high detection coverage [Pat07b].

**Checking expression:** A checking expression is a sequence of instructions that recomputes the critical variable, and is optimized aggressively and differently from the rest of the program code [Pat07b].

**Detector:** A detector is defined as the combination of the checking expression and the runtime monitoring [Pat07b].

### 2.1.3. Implementation

The implementation of this error detection technique assumes that the initial program is instrumented at compile time with instructions that later, at run-time, will detect errors. The rest of this section describes the steps performed to instrument an application with error detectors.

**Identification of critical variables**

The critical variables are identified by analyzing the dynamic execution of the program. The application is executed with representative inputs to obtain its dynamic execution profile, which is used to choose critical variables for detector placement. The critical variables are the

\(^1\) Full duplication assumes that the result for every instruction is compared to prevent the application from error propagation, which in turn results in high performance overheads [Pat07a].
variables with highest dynamic fanouts in the program. The errors that occur in these places could propagate to many locations and could cause a program failure [Pat07a].

**Computation of backward slice of the critical variables**

The backward slice is built traversing the static dependence graph starting from the instruction that computes the critical variable up to the beginning of the function. The slicing algorithm used is a static slicing technique that considers all possible dependences between instructions in the program regardless of program inputs [Pat07a].

The backward program slice is extracted for the identified critical variables. This is done by traversing the Static Dependence Graph (SDG) of the program, starting from the instruction that computes the critical variable (called critical instruction), going back until the beginning of the function is reached (at most). In order to derive the backward program slice, a backward traversal of the SDG is performed starting from the critical instruction, and continuing until one of the following conditions is met:

- **The beginning of the current function is reached** (only intra-procedural slices are considered). The rationale behind this decision is that it is sufficient to consider intra-procedural slices in the backward traversal because each function is considered separately for the detector placement analysis. As a consequence, slices do not include recomputation of function parameters across function boundaries. If a parameter is a critical variable, then a detector will be derived for it in the calling function.

- **A basic block is revisited in a loop** (loops are not recomputed). This issue is handled in the following way: if during the backward traversal a dependence within a loop is encountered, the loop is not recomputed in the checking expression. Instead, the check is broken into two separate checks, one placed on the critical variable, and one on the variable that affects the critical variable within the loop (to ensure that this variable is computed correctly, and can be used in the first checking expression). Hence, only acyclic paths are considered by the algorithm.

- **A dependence across loop iterations is encountered** (only previous loop iterations are considered when traversing loop-carried dependencies). This happens when the critical instruction occurs in-between the producer instruction of the dependence and the consumer instruction of the dependence. When a loop-carried dependence across two or more iterations is encountered, the dependence is truncated and the loop dependence is not included in the slice. This is done because recomputing critical variables across multiple loop iterations can involve loop unrolling or buffering intermediate values that are rewritten in the loop, which in
turn can complicate the design of the detector. Instead, the check is broken into two checks, one for the dependence-generating variable across multiple iterations and one for the critical variable.

- **A memory operand is encountered** (memory dependencies are not considered). This is done because LLVM promotes most memory objects to registers. Since there is an unbounded number of virtual registers, the analysis does not have to be constrained by the actual number of physical registers available on the target machine. However, it can sometimes happen that it is not possible to promote a memory object to a register (e.g. pointer references to dynamically allocated data). In such cases, the load of the memory object is duplicated (provided that the load address is not modified along the control path).

It should be noted that the slice is specialized for each acyclic control path. Also, very important, the slice contains only the instructions that compute the critical variable along the specific program path. So, aggressive optimization can be employed (using the available compiler optimization infrastructure provided by LLVM).

Unfortunately, certain instructions cannot be recomputed in the checking expressions, because performing recomputation of those can alter the semantics of the program. Examples are mallocs, frees, function calls and returns. Omitting function calls and returns does not impact coverage because the detector placement analysis considers each function separately. Omitting mallocs and frees does not seem to impact coverage, except for allocation intensive programs [Pat07a].

An important observation would be that, by using this technique, diversification is achieved. Since the specific set of instructions corresponding to a slice is optimized separately from the rest of the program, this introduces a level of diversity in the recomputation of the critical variable. This diversity is valuable because it helps detecting errors in the program instructions that are interleaved with the critical variable recomputation. In other words, we might say that the probability of an error impacting the value produced by the instruction in both the original program and in the checking expression is very small. Therefore, common mode errors between the checking expression and the original program are highly reduced.

**Check derivation and Check insertion**

The specialized backward slice for each control path is optimized considering only the instructions on the corresponding path, to form the checking expression. The checking expression is inserted in the program before the use of the critical variable [Pat07a].
The program is also instrumented with instructions that keep track of the control paths followed at runtime so that the corresponding checking expression for that specific path will be executed.

**Runtime checking in hardware and software**

In this section we are going to illustrate how this error detection technique works using a simple example. In the Figure 2.1 we present a fragment of a C program. As we can see on the left side of the figure the original code is presented (with no shading) and on the right side the checking code is presented (with light shading). We can identify two possible control paths in the program shown in Figure 2.1 corresponding to the two branches. Each of these paths will calculate a different value of “t”. In the figure we see clearly that the checking expression changes depending on the path chosen.

![Figure 2.1: Code fragment for error detection [Pat07b].](image)

So, in order to employ error detection, we have to place different critical expressions in the program specific for each path. It is also needed to keep track of the control path at run time. We can notice in the figure that there are two different expressions depending on the path, expression \( t' = s + 5 \), for path 1 (where \( t' \) is temporary variable) and \( t' = x \times 2 + y \), for path 2. These expressions will re-compute the result for the critical variable and compare this with the original value computed in the program. At the right side of figure 2.1 the value of \( t \) (computed
by the original program) is compared with t’ (which is calculated by the checking expression). If
the results do not match this means that an error has occurred. In the case of error the
program will stop its execution and do recovery; otherwise it will continue its execution.

Performance overhead

The error detection technique that we have discussed has two main sources of runtime
performance overhead: path tracking and variable checking. Both path tracking and the
variable checking could be implemented either in software or in hardware. The implementation
of path tracking and checking in the hardware will increase the cost of the system, but
implementing both in software would increase the time overhead. Efficient and low overhead
implementation for the path tracking has been proposed [Pat07a].

Unfortunately implementing each checking expression to its own dedicated hardware
incurs excessive costs. In order to overcome this wasteful use of resources we propose to place
in hardware only those checking expressions that have the potential to provide highest
reduction of average execution time.

2.2. Fault Model

The fault model used in this technique covers errors in data values due to hardware
faults [Pat07a].

• Errors detected - due to:

Hardware faults: any transient error that would affect one of the following hardware
components:

1. Processor data path: data-value corruption because of errors in the functional units or
in the register file. For example, an ALU instruction is executed incorrectly inside a functional
unit, or the wrong memory address is computed for a load/store instruction, resulting in data
value corruption.

2. Processor control path: errors in the instruction decode and issue units that result in
the wrong instruction being executed. Either the wrong instruction is fetched, or a correct
instruction is decoded incorrectly, resulting in data value corruption.
3. Memory/Cache: soft errors in the main memory or cache, caused by, for example, electric interference or cosmic radiation\(^2\). As a result, the value will be incorrectly interpreted in the program.

It should be noted that this technique will detect the above categories of hardware errors only if they affect the computation of the critical variables, in either the program, or the checking expression. Common mode errors are handled by the diversity introduced in the checking expressions.

- **Errors not detected** - some examples:

  **Hardware errors**: permanent or persistent errors in some hardware components might not be detected. For example, if detectors are implemented in software, and the same, faulty, functional unit is used in the computation by both the original program, and the detector’s code. However, the probability for this to happen is very small.

### 2.3. General Idea

The error detection technique as we have discussed has to monitor many critical variables in order to prevent the application from failure. Putting entire detectors for critical variables in hardware will increase the hardware cost and this might be inacceptable. Similarly when we place all these detectors in software, this increases the time overhead. To maximize the efficiency and minimize the cost we need to use a mixed solution which involves both hardware and software. Using the above error detection technique, we have introduced two approaches to optimize the error detection techniques.

**Greedy Approach**

We can model our problem as a knapsack problem, and we propose two algorithms to solve it. The first one is based on a greedy heuristic. Using this approach we place some checking expressions in the hardware (FPGA) and other in the software. In this approach we assign the priorities to the critical variables based on time gain per hardware unit. We try to place in hardware those checking expression that will reduce the average execution time the most. We will discuss this approach in detail in section 5.1.1.

---

\(^2\) Other error detection techniques, such as ECC, are capable of detecting these errors, if deployed in memory and cache.
**Dynamic Programming Approach**

Since the greedy approach does not guarantee optimality, we also proposed an algorithm based on dynamic programming. Similar to the greedy approach we place those checking expressions in the hardware (FPGA) that will generate the biggest reduction of average execution time, while the others we keep in software. We will discuss this approach in detail in the section 5.1.2.
Chapter 3

System Model

This chapter details the hardware architecture that we considered in our approach, the application model we used, and also covers other assumptions we made in our work.

3.1. Application Model

In section 1.1 we have mentioned that we focus on the soft real-time applications and multimedia applications. The programs that we have considered for our system are structured. These structured programs are modeled as a control flow graph $G = (N, E)$ where $N$ is the set of all nodes and $E$ is the set of all edges.

*Definition:* A *control flow graph (CFG)* of a program is a directed graph $G = (N, E)$, where each node in $N$ corresponds to a straight-line sequence of operations and the set of edges $E$ corresponds to the possible flow of control within the program. $G$ captures potential execution paths and contains two distinguished nodes, *root* and *sink*, corresponding to the entry and the exit of the program.
Let us consider an example to elaborate the control flow graph and the checking expressions. Figure 3.1 shows an example of a control flow graph of a small program, consisting of 9 nodes and 10 edges.

Let’s suppose that node “7” contains a critical variable that needs to be checked. After applying the error detection technique that we discussed in section 2.1.3, we derived four checking expressions (C1, C2, C3 and C4) shown in the figure 3.1. Each of the checking expressions corresponds to an acyclic control path in the control flow graph shown in figure. Each checking expression has its overheads, like time overhead (execution time in both hardware and software) and area that it will occupy in hardware as shown in the table 1. Since node “7” is composed of four different checking expressions, its execution time depends on the path followed at runtime. This will determine which of the four mutually exclusive checking expressions will be executed.

Figure 3.1: Application Structure

Path C1 : 1,2,3,4,6,7
Path C2 : 1,2,3,5,6,7
Path C3 : 8,2,3,4,6,7
Path C4 : 8,2,3,5,6,7

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C4</td>
<td></td>
</tr>
<tr>
<td>1%</td>
<td>50%</td>
<td>20%</td>
<td>29%</td>
<td></td>
</tr>
</tbody>
</table>
Our system model considers that each edge in the control flow graph is characterized by the probability to follow that edge and this is obtained from a counter based execution profile of the application.

If we assume that all the checking expressions are implemented in software, the average time overhead for all the checking expressions will be calculated as:

\[
\text{Average time overhead}_{\text{sw}} = (1\% \times 25) + (50\% \times 21) + (20\% \times 18) + (29\% \times 32) = 23.63
\]

On the other hand, if we assume that the FPGA is big enough to accommodate all the checking expressions in the hardware, the average time overhead for all the checking expressions will be calculated as:

\[
\text{Average time overhead}_{\text{hw}} = (1\% \times 4) + (50\% \times 3) + (20\% \times 2) + (29\% \times 3) = 2.81
\]

<table>
<thead>
<tr>
<th>Checking Expression</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software: Time overhead</td>
<td>25</td>
<td>21</td>
<td>18</td>
<td>32</td>
</tr>
<tr>
<td>Hardware: area</td>
<td>20</td>
<td>15</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>Hardware: Time overhead</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.1

As mentioned before, by analyzing the control flow graph, we will try to obtain a performance close to the hardware only implementation but using as small hardware as possible. This is achieved by placing into FPGA only the most profitable checking expressions.

3.2. System Architecture

The applications we have considered run on an architecture that is composed of a central processing unit, a memory subsystem and a reconfigurable device (FPGA).

![Figure 3.2: Hardware Architecture](image)
The FPGA that we have modeled supports static reconfiguration. This means that the entire configuration memory has to be configured before the start-up of the system. After that, the modules on the FPGA cannot be reconfigured unless the system is turned off. This means that no run time, dynamic reconfiguration is possible. The FPGA is represented as a rectangular matrix of configurable logic blocks (CLBs). Each checking expression occupies a contiguous rectangular area of this matrix. Our model allows 2-dimensional as well as 1-dimensional placement and reconfiguration. In the case of 1-dimensional reconfiguration we reconfigure whole columns of CLBs. For the 2-dimensional case we allow reconfiguration of any rectangular shape and size that fits in the given FPGA.

Figure 3.2(b) illustrates our FPGA module. We show a 7x9 FPGA, on which we place module M1 (an example of a 1-dimensional placement) and module M2 (an example of a 2-dimensional placement).
Chapter 4

Problem Formulation

4.1. Input

As we have discussed in section 3.1 our application is a structured program, with a set of checking expressions placed at different points in the program. This set of checking expressions is denoted by $CE = \{C_1, C_2, C_3, \ldots, C_n\}$. Each checking expression has associated the following attributes:

- When the checking expression is implemented in software (SW), the value of time overhead ($TO_{SW}$) for this checking expression will be a positive integer value, given by the following function:

  $$TO_{SW} : CE \rightarrow \mathbb{Z}^+$$

- Similarly, when the checking expression is implemented in hardware (HW), there are two overheads that should be considered.
  - Time overhead ($TO_{HW}$) for hardware will be much less than the software time overhead; it’s a positive integer value, given by the following function:

    $$TO_{HW} : CE \rightarrow \mathbb{Z}^+$$

  - Area occupied by the checking expression in hardware (HW) will be a positive integer, given by the following function:

    $$AREA : CE \rightarrow \mathbb{Z}^+$$

    We consider that the area occupied by a checking expression represents the number of columns that are needed for that expression.
Each checking expression has associated a probability (PB), which refers to the probability that the checking expression’s path is followed at runtime. This probability is given by the following function:

\[ \text{PB: CE} \rightarrow [0:1] \]

Each checking expression has associated an iteration count (IC) value which denotes the number of times the expression gets execution inside a loop (if the case):

\[ \text{IC: CE} \rightarrow Z^+ \]

\[ \text{IC} (C) = \begin{cases} 
1, & \text{if checker is not inside a loop} \\
\text{average no. of loop iterations, if checker is executed inside a loop} & 
\end{cases} \]

4.2. Output

The set of checking expression implemented on the FPGA: \( \text{CE}_{\text{HW}} \subset \text{CE} \) such that the average execution time of the program is minimized. This implies that the average time gain is maximized subject to the hardware area constraints:

\[
\begin{align*}
\text{maximize} & \quad \sum_{C \in \text{CE}_{\text{HW}}} \{ \text{PB}(C) \left[ \text{TO}_{\text{SW}}(C) - \text{TO}_{\text{HW}}(C) \right] \} \\
\text{Such that} & \quad \sum_{C \subset \text{CE}_{\text{HW}}} \text{AREA}(C) \leq \text{FPGA area}.
\end{align*}
\]
Chapter 5

Optimization Algorithm

5.1. Knapsack Problem

The knapsack is one of the classical problems in optimization [Han10]. The knapsack problems are very common and they appear in many forms in economics, engineering, business and in any place where we deal with the allocation of a single scarce resource among multiple contenders for that resource. It has acquired the name “knapsack problem” because our common experience of packing luggage expresses something of the flavor of the problem: What should be chosen when space is limited?

Description: We have a set of items each with a cost and a value. We want to determine the items to include in a collection so that the total cost does not exceed some given cost and the total value is as large as possible.

There are two different types of problems: (i) “0-1 knapsack problem” (ii) “Fractional knapsack problem”

0-1 knapsack problem: In the 0-1 knapsack we are only allowed to put the whole item in the knapsack or not at all. We cannot split any item (for example, putting half of an item in the knapsack is not allowed).

Fractional knapsack problem: In fractional knapsack we can take fractional number of items, so we can split the items and put them in the knapsack.

In our case, since a checking expression cannot be split, our problem is an instance of the 0-1 knapsack problem.
Formal Description: We are given an instance of the knapsack problem with item set $N$, consisting of $n$ items $j$ with profit $p_j$ and weight $w_j$, and the capacity value $c$. (Usually, all these values are taken from the positive integer numbers.) Then the objective is to select a subset of $N$ such that the total profit of the selected items is maximized and the total weight does not exceed $c$ [Han10].

\[
\text{(KP) maximum } \sum_{j=1}^{n} p_j x_j \\
\text{Subject to } \sum_{j=1}^{n} w_j x_j \leq c, \\
x_j \in \{0,1\}, \quad j=1,\ldots,n.
\]

Informal Description: We have a number of $n$ checking expressions that we want to store into FPGA, and we have $W$ slices available for storage and we want to store those checking expressions in the available $W$ slices, so that the average time gain is maximized. The objective is to choose the set of checking expressions that fits in the FPGA and maximizes the time gain.$^3$

5.1.1. Greedy Approach

The greedy algorithm is easy to implement and it is quite efficient most of the times. It is used to solve many optimization problems. But it is also true that the greedy algorithm does not produce the optimal solution for the 0-1 knapsack problem. Although for fractional knapsack problem greedy finds the optimal solution [Han10], since our problem is an instance of 0-1 knapsack, the optimal solution is not guaranteed.

Description: In general an algorithmic approach is known to be “greedy” when it makes the decisions for each step based on what seems best at the current step. Moreover, once a decision is made, it is never revoked. It may seem that this approach is rather limited. However, many important problems have special features that allow correct solutions using this approach. Since we do not revoke our greedy decisions, that makes it simple and fast.

Let us next describe our proposed greedy algorithm to solve the problem defined in section 4.1. First of all we read all the checking expressions form the input file and put all the checking expression in the link list, as shown in Algorithm 1 in line 1 and 2. This list is later sorted on basis of time gain per hardware unit. The time gain per hardware unit is defined as the difference between the time overhead when the checker is implemented in software and the time overhead when the checker is implemented in hardware times the probability of following that path at runtime divided by the area needed by that checker (shown in line 4). After sorting the list, we place as many checkers that fits in the FPGA depending on the

---

$^3$ For each checking expression the average time gain defined as: $\text{Time Gain (C)} = PB(C) \left[ TO_{SW}(C) - TO_{HW}(C) \right]$
available space. Since the list is sorted in a descending order on basis of time gain per hardware unit therefore we will have the checker with highest gain per hardware unit at the top of the list.

Algorithm 1:

1. Read all checkers from file
2. Store them in Linklist checker_list;
3. For each checker c in checker_list do
   4. c.timegain_perHWunit = (c.PB [c.TO_{SW} − c.TO_{HW}] / c.AREA);
5. Do Sorting (checker_list); on basis of timegain_perHWunit
6. while ( FPGA_area is available )
   7. {
      8. Pick & remove first checker that fits into the FPGA ;
      9. Place that checker into the FPGA;
   10. }

<table>
<thead>
<tr>
<th>Checkers</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW timeOverhead (TO_{SW})</td>
<td>150</td>
<td>100</td>
<td>103</td>
<td>300</td>
</tr>
<tr>
<td>HWtimeOverhead (TO_{HW})</td>
<td>50</td>
<td>15</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>Area (slices)</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Probability (PB)</td>
<td>50%</td>
<td>20%</td>
<td>20%</td>
<td>10%</td>
</tr>
<tr>
<td>Gain =PB. (TO_{SW} - TO_{HW})</td>
<td>50</td>
<td>17</td>
<td>17.4</td>
<td>27</td>
</tr>
<tr>
<td>Avg. Gain/Area</td>
<td>10</td>
<td>8.5</td>
<td>8.7</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.1

Figure 5.1 (a): Link List Structure

Figure 5.1 (b): Unsorted Link List
Illustrative Example

The above figures give a clear picture of how the greedy algorithm works. Let us consider a program that contains four checkers as shown in table 5.1. These checkers are given as an input, then these checkers are stored in a link list as shown in figure 5.1 (b). After that the link list is rearranged according to the average time gain per hardware unit as described in Algorithm 1, (Figure 5.1(c)) then the “While” loop from Algorithm 1 (lines 6-7) is executed.

Whenever we need to decide for a new location on the FPGA where to place an error detection module we use the first-fit placing. In the 1 dimensional case this decision does not have such a big impact, as in the 2 dimensional case.

Let us consider an FPGA of size 9 slices. Our algorithm will first choose C1 for placement. Since the FPGA is initially empty, C1 is placed as shown in figure 5.2(a). At this point, C4 is the checker providing the highest time gain per hardware unit. Since its size (3 slices) smaller is than the available space on FPGA (4 slices), we chose C4 for placement. We cannot place C3 and C2 on FPGA because there is not enough space available for them. As the result, the output of our greedy algorithm is to place C1 and C4 into the FPGA, while keeping C3 and C2 in software. As a result, the average time gain obtained in this case is 77 time units. Unfortunately this result is suboptimal, as we shall show in section 5.1.2.

For simplicity reasons we have illustrated only the 1-dimensional placement in this example, but the placement function could also effectively handle 2-dimensional placements in FPGA.
5.1.2. Dynamic Programming Approach

As we have mentioned before the Greedy Approach is simple to implement but it does not guarantee to find the optimal solution. Dynamic programming is one of the methods for solving such problems and obtaining the optimal solution.

We want to place on FPGA those checking expressions, which provide maximum time gain. We cannot place parts of a checking expression; it is the whole checking expression or nothing. There are two types of knapsack problems as we have discussed before. Greedy would provide optimal solution if the checkers would be divisible. Otherwise it does not guarantee optimality. Since the item (checking expression) is indivisible we solve the problem with *dynamic programming*.

**The idea:** Compute the solutions to the sub-problems *once* and store the solutions in a table, so that they can be reused (repeatedly) later.

Dynamic programming is a programming technique just like divide and conquer. With Dynamic programming we solve the problem gradually i.e. divide the problem in small sub problems and gradually build the solution to the original size of the problem. If we can work out using smaller capacity knapsack using fewer items then we can start with arbitrary small knapsack then slowly build to the original problem size. To do this we need to have two things:

- Firstly we need some base cases where we absolutely know the answer already. In the knapsack problem we could do this using two base cases. Firstly, if the knapsack capacity is zero then obviously the best solution is going to put zero items in the knapsack because none of the items fit in the knapsack. The other possible base case is that if we do not have any items then obviously the best solution is going to put no item in because we do not have any items (shown in the algorithm 2 in lines 5 and 6).
- Secondly we need some rules as well as our base case to build up the knapsack to the size of the original problem.

In order to solve any problem using dynamic programming we need to perform a series of steps:

**Step1:** *Structure:* Characterize the structure of an optimal solution.

- Decompose the problem into smaller problems, and find a relation between the structure of the optimal solution of the original problem and the solutions of the smaller problems.

**Step2:** *Principle of Optimality:* Recursively define the value of an optimal solution.
- Express the solution of the original problem in terms of optimal solutions for smaller problems.

**Step 3:** *Bottom-up computation:* Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

**Step 4:** *Construction of optimal solution:* Construct an optimal solution from computed information.

Steps 3 and 4 may often be combined.

---

**Algorithm 2:**

```plaintext
1 Read all checkers from file
2 Store in Linklist checker_list;
3 For each checker c in checker_list do
4   c.timegain_perHWunit = (c.PB [c.TO_SW - c.TO_HW] / c.AREA);
5 for i = 1 to total-checkers
6   Z[ 0 , i ] ← 0
7 for i = 1 to FPGA-area
8   Z[ i , 0 ] ← 0
9 for i = 1 to total-checkers
10   for j = 1 to FPGA-area
11     if (FPGA-area < checker(i).area)
12       { Copy the previous solution from Z[ j , i-1 ] to Z[ j , i ]
13       }
14     else if (FPGA-area >= checker(i).area)
15       { if (Previous solution of Z[ j , i-1 ] >= ( Z[j- checker(i).area][i-1] + checker(i).gain)
16         Copy the previous solution from Z[ j , i-1 ] to Z[ j , i ]
17         else
18             current solution = previous solution + checker(i).gain
19             Copy current solution in Z[ j , i ]
20         }
21     }
22 }
```
In the next paragraph we will explain how we solved our knapsack problem using dynamic programming (Algorithm 2). The recursion in the Algorithm 2 is known as Bellman recursion. There is an “if” condition in line 11 which expresses that we consider a knapsack which is too small to contain this item (current checker in the list). Hence, this item could not change the optimal solution. So we copy the previous solution (line 18). If the item (current checker) does fit into the knapsack, there are two possible choices shown in line 17 and 19. Either the current checker is not packed into the knapsack and the previous solution \( Z[i, j] \) remains unchanged or the current checker of the list is added into the knapsack and adds its gain to the solution value but decreases the capacity. The remaining capacity should be filled with checkers that generate as much gain as possible. The best possible solution value for this reduced capacity is given by \( Z[j – \text{checker}(i)\_area][i-1] \) shown in line 17. Taking the maximum of these choices between line 17 and 19, we yield the optimal solution [Han10].

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>17.4</td>
<td>17.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{2}</td>
<td>{3}</td>
<td>{3}</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>17.4</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{2}</td>
<td>{3}</td>
<td>{4}</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>34.4</td>
<td>34.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{2}</td>
<td>{2,3}</td>
<td>{2,3}</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>50</td>
<td>67</td>
<td>67.4</td>
<td>67.4</td>
</tr>
<tr>
<td></td>
<td>{1}</td>
<td>{1,2}</td>
<td>{1,3}</td>
<td>{1,3}</td>
<td>{1,3}</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>50</td>
<td>67</td>
<td>67.4</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>{1}</td>
<td>{1,2}</td>
<td>{1,3}</td>
<td>{1,3}</td>
<td>{1,4}</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>50</td>
<td>67</td>
<td>84.4</td>
<td>84.4</td>
</tr>
<tr>
<td></td>
<td>{1}</td>
<td>{1,2}</td>
<td>{1,2,3}</td>
<td>{1,2,3}</td>
<td>{1,2,3}</td>
</tr>
</tbody>
</table>

**Table 5.2:** The Optimal Solution values computed by DP
Illustrative Example

In this subsection we will elaborate more on the dynamic programming approach and illustrate how Algorithm 2 works using an example. The four checkers shown in the table 5.1 are given as input. Similar to the Algorithm 1 these checkers are stored in the link list. We construct a two dimensional array $Z$ of size $Z_{\text{FPGA_area, total-checker}}$. The array $Z$ will store the maximum computed gain of any subset of checkers of size at most FPGA size as shown in table 5.2. As discussed before we start with the small problem size considering, zero checkers at the start as shown in table 5.2, and gradually we increase the problem size up to the total number of checkers. The vertical dimension of the array is indexed between 0 and the size of the FPGA.

We compute the gain for each entry in the array as follows. If the computed gain is less than the previous gain we copy the previous solution otherwise we choose the best solution between including the current checker in the current solution and keeping the best solution obtained without considering this checker.

Whenever a new gain is added to the previous solution in the table, we actually went through the line 19 to 21 in the Algorithm 2, for example we look at the table 5.2 at location row 7 and column 2 the new gain is added into the previous solution ($Z[j\text{-checker}(i)\text{.AREA}][i-1]$) which is row 5 and column 1 at this point, we keep track of the checkers separately according to the gain in table.

After computing all the entries in this array, the array entry $Z[\text{max, max}]$ (i.e. the last entry of the array) will contain the maximum computed gain of checking expressions that can fit into the FPGA, i.e. the solution to our problem. In table 5.2 the last entry is 84.4 which expresses the gain obtained when placing checkers $\{1,2,3\}$ into the FPGA. As we can observe, the dynamic programming approach generated the optimal solution which is 8.77% better than the solution generated by our greedy algorithm.
Chapter 6

Experimental Evaluation

6.1. Experimental Setting

In order to evaluate the algorithms we have performed several experiments on synthetic examples.

We have generated different applications for our algorithms. These applications consist of different number of detectors like 5, 25 and 100. For each detector in these applications we randomly generate, between 2 and 6 checking expressions. Thus the first problem size contains roughly 20 checking expressions, the second problem size contains roughly 100 checking expressions and the third contains 400 checking expressions. For each problem size we randomly generated 10 applications. Each detector consists of a number of checking expressions with different parameters like:

- A global probability to execute that checking expression. This can be computed as the product of probabilities of all edges in that path.
- The execution time if this checker is implemented in software.
- The execution time and the necessary FPGA area, if this checker is implemented in hardware.

The values for each of these parameters are randomly generated. Different probabilities were assigned to each of the checking expressions. The time overhead for the software
implementation is higher than the hardware one, which was assigned a value depending on the area required for that particular checking expression.

In both approaches we have varied the size of the FPGA available for the placement of the detectors. We sum all the hardware areas for all the checking expressions for a certain application:

\[
\text{Maximum Hardware} = \sum_{i=1}^{\text{card}(CE)} \text{AREA} (C_i),
\]

Where card (CE) is the size of the set CE, containing all the checking expressions in an application.

Then we generated problem instances by considering the size of the FPGA corresponding to different fractions of maximum hardware, like 5%, 10%, 15%, 30%, 45%, 60% and 75%. We have assumed 1-dimensional placement and reconfiguration for the FPGA, i.e. the checkers in the hardware will occupy a certain number of adjacent columns.

These test cases are run on a machine with Core 2 Duo CPU frequency 2.00 GHz, 2 GB of Ram and running Windows 7.
6.2. Applications With 20 Checking Expressions

The above figures show the experimental evaluation for the Greedy Approach and the Dynamic Programming Approach for the test case with five detectors (and 20 checking expressions). In figures 6.1 (a) and (b) we plot gains (that we have achieved by placing the checking expressions in hardware) for each hardware fraction. As we can notice, as we increase the availability of the hardware for checking expressions the gain increases fast in the beginning and then saturates. The graphs in both the figures 6.1 (a) and (b) follow the same trend. The gain with the Dynamic Programming Approach is slightly more than the one with the Greedy Approach.
In figure 6.2 the graph shows the percentage of gain for varied hardware, that we managed to obtain from the maximum gain possible (that corresponding to 100% hardware fraction). We first computed the maximum gain as:

$$\text{Max\_Gain} = \sum_{i=1}^{\text{card}(CE)} \{PB(C_i) \cdot [TO_{SW}(C_i) - TO_{HW}(C_i)]\}$$

Then we computed the gain resulted after running our optimization for each problem:

$$\text{Gain} = \sum_{i=1}^{\text{card}(CE_{HW})} \{PB(C_i) \cdot [TO_{SW}(C_i) - TO_{HW}(C_i)]\}$$

In figure 6.2 we plot the average (over the ten applications) percentage of gain (PG) obtained:

$$\text{(PG)} = \{\frac{\sum_{i=1}^{10}(\text{Gain}/\text{Max\_Gain})}{10}\}$$

We can observe that with as few as 10% hardware fraction we obtain more than 45% of the maximum gain, while with 60% hardware fraction we already obtain more than 90% of the maximum gain. This is possible because our algorithms first place into the FPGA the checking expressions that generate a big reduction of average execution time. As a result, the remaining checking expressions will either have a low probability to be executed or will generate a small time gain.

As we can see the difference between the Greedy Approach and Dynamic Programming Approach are marginal. This might be due to the nature of the problem: the granularity of the checking expressions is relatively small, so the solutions generated by the Greedy Approach are quite good.
Figure 6.3: Average Running Time

The figure 6.3 shows the diagram for average running time over the ten applications for both the approaches. The graph plots the execution time for varied hardware. The curve marked with squares represents the Dynamic Programming Approach while the curve marked with diamonds represents the Greedy Approach. As we can notice the execution time for the Dynamic Programming is growing at a faster rate.
6.3. Applications With 100 Checking Expressions

The above figures show the experimental evaluation for the Greedy Approach and the Dynamic Programming Approach for twenty five detectors (100 checking expressions). The graphs in the above figure follow the same trend line as in figure 6.1 (a) and (b). The difference that we can notice between the figure 6.1 and 6.4 is that in figure 6.4 the points are closer to the mean while in figure 6.1 the points are more scattered. The average gain has also increased slightly in the figure 6.4.
Figure 6.5

Similar to figure 6.2, figure 6.5 shows the percentage of gain (PG) that we managed to obtain for each hardware fraction. The same comments apply here, as in figure 6.2.

Figure 6.6: Average Running Time

The figure 6.6 shows the graph for average running time over the ten applications considered both approaches. The graph plots the execution time for varied hardware. As we can notice the curve which represents the Dynamic Programming Approach is growing much
faster than the curve which represents the Greedy Approach and follows the linear trend. The execution time for Dynamic Programming is approximately 20 times higher than the Greedy Approach. Considering that Dynamic Programming algorithms have a pseudo-polynomial behaviour [Han10], these results were expected.
6.4. Applications With 400 Checking Expressions

The above figures show the experimental evaluation for the Greedy Approach and the Dynamic Programming Approach for hundred detectors which have around 400 checking expressions. Figure 6.7(a) represents the gain for the Greedy Approach and it follows the same trend as in figure 6.1. We can notice that the experimental evaluation for the Dynamic Approach is incomplete. Since the number of checking expressions is large the Dynamic Programming Approach consumes a lot of memory. Therefore we could not complete the experimental evaluation for the Dynamic Programming Approach as we run out of memory. Another interesting aspect visible in figure 6.7(a) and (b) is that the points are even closer to the mean than in the previous figures.
The figure 6.8 is similar to the previous figures 6.5 and 6.2, shows the percentage of gain (PG) that we managed to obtain from the maximum possible, for each hardware fraction. The same comments apply here, as in figure 6.5 and 6.2.
The figure 6.9 shows the graph for average running time for both the approaches. The graph plots between the execution time for varied hardware. We can notice that the curve which represents the Dynamic Programming Approach is very steep, while the curve which represents the Greedy Approach has polynomial behaviour. The execution time for Dynamic Programming is much higher than the Greedy Approach. These observations, correlated to the fact that with both approaches we obtain similar performance, suggests that in many cases the Greedy Approach is to be preferred.
6.5. Case Study: GSM Encoder

We have also tested our approaches on a real-life example. This real-life example is a GSM encoder which implements the European GSM 06.10 provisional standard for full rate speech transcoding.

We instrumented the whole application with 56 checkers, corresponding to the 19 most critical variables, according to the technique described in section 2.1.3. The execution times were derived using the MPARM cycle accurate simulator, considering an ARM processor with an operational frequency of 40 MHz. The checking modules were synthesized for an XC5VLX50 Virtex-5 device, using the Xilinx ISE WebPack.

The control flow graph for each task, as well as the profiling information was generated using the LLVM suite [Lat04] as follows: llvm-gcc was first used to generate LLVM bytecode from the C files. The opt tool was then used to instrument the bytecode with edge and basic block profiling instructions. The bytecode was next run using lli, and then the execution profile was generated using llvm-prof. Finally, optanalyze was used to print the control flow graph to .dot files. We run the profiling considering several audio files (.au) as input. The results of this step revealed that many checkers (35 out of the 56) were placed in loops and executed on average as much as 375354 times, which suggests that it is important to place them in HW.

After finding the parameters that were require (as mentioned in section 4.1), we run the optimization algorithms. The results obtained are shown in figures 6.10 to 6.12.

![Figure 6.10(a): Greedy Approach](image1)

![Figure 6.10(b): DP Approach](image2)
The figures 6.10 (a) and (b) show the experimental evaluation for the Greedy Approach and the Dynamic Programming Approach for the GSM encoder which has 56 checking expressions.

Figure 6.10(a) represents the gain for the Greedy Approach and the graph is similar to the graph in figure 6.5. Since the application has roughly the same size. As we can notice, the gain increases as we increase the hardware area and after a certain point it stabilizes and gets saturated. In the above figure 6.10(a), after 30% of hardware area the gain starts to saturate.

The figure 6.10(b) represents the gain in percentage for the dynamic programming approach and this figure is similar to figure 6.10(a). The only difference is that the gain for small fractions of available hardware in the dynamic programming approach is more than with the greedy approach.

In figure 6.11 the graph is similar to figure 6.2 and figure 6.5, which shows the percentage of gain (PG) that we managed to obtain for each hardware fraction. The same comments apply here, as for figure 6.2.
Figure 6.12 shows the graph for average running time for both approaches. The graph plots the execution time for varied hardware fractions. The execution time for Dynamic Programming is higher than the Greedy Approach. At the beginning with small fraction of hardware, the running time of the Dynamic Programming is twice as big compared to the Greedy Approach, but as we increase the hardware availability the difference between the running time of the two approaches also increases and at highest point the running time for Dynamic Programming is nine times higher than the Greedy Approach. As mentioned earlier, these results are expected since Dynamic Programming has pseudo-polynomial behaviour (so the running time increases as we increase the size of the FPGA and the number of checking expressions). The result for average execution of GSM encoder is similar to our previous series of test with 25 detectors.
Chapter 7

Conclusion

In this chapter, we will discuss the overall thesis and conclude based on the outcomes of our experiments. In this thesis we have presented two approaches for optimization of the error detection implementation. These algorithms are proposed in order to minimize the average execution time for soft real-time and multimedia applications.

The applications that we have considered run on a hardware architecture consisting of a single node composed of a processor and a memory subsystem. The node also has attached a reconfigurable device (e.g. FPGA).

In our work, we have used an error detection technique developed by Pattabiraman et al., we consider the alternative implementation; for error detection: software only, mixed (using both software and hardware) and hardware only. Each of the proposed methods has different time and cost overheads. For hardware only, the hardware cost of the system increases, but the time overhead is reduced. For software only, although there is no hardware costs incurred, the time overhead might be unacceptable. We have focused on the mixed approach and tried to maximize the gain with limited hardware by placing as much checking expressions as possible depending on the resource constraints and based on profiling information.
To achieve the maximum gain and to use the hardware as efficiently as possible, we have formulated the problem as a knapsack problem, for which we proposed two algorithms. The first one is a greedy approach and the second one finds the optimal solution using dynamic programming.

The greedy approach sorts the checking expressions on the basis of the input parameters provided like probability, time gain and necessary hardware area. All of these input parameters are discussed in detail in section 4.1. The greedy approach sorts the checking expressions’ list and rearranges the checking expressions according to their time gain per hardware unit. By using this approach we place in the hardware (FPGA) some high priority critical variables that potentially give us the highest reduction of average execution time, while the rest we keep in the software. The algorithm provides high quality solutions that are discussed in detail in chapter 6.

The dynamic programming approach is a bit more complex. The dynamic programming technique solves the problem gradually i.e. divides the problem to small sub-problems and gradually builds the solution to the original problem. This technique finds the optimal solution at the expense of much higher execution time and memory consumption than the greedy approach. The technique and the algorithm are discussed in section 5.1.2.

To evaluate our algorithms and to find the performance of the two approaches we have considered a series of applications with different number of detectors and checking expressions. We have also run our optimization on a real-life application (GSM encoder) to see how these algorithms perform under real-life scenarios. A common result that we have come across is that both approaches generate significant gain with relatively small hardware available. The experiments also show that after 30% of the hardware the gain starts to saturate.

The experiments revealed, especially in the case of the GSM encoder, that when small hardware is available for the placement of the checking expressions, the results of the dynamic programming approach are better and outperform those generated by the greedy approach. As we increase the hardware availability, the difference between the gains produced by the two approaches reduces. Since the dynamic programming is complex the average running time for this approach is higher, and memory consumption is also significant.

As we see from the results of our experiments, the running time for the greedy approach was better than the dynamic programming approach and the gain for both the approaches were quite similar especially when we increase the hardware availability over 30%.
To conclude, the work that has been presented in this thesis can be used as a foundation for future research in the area of design optimization of embedded systems. The implementation could be extended for distributed architectures. Since our implementation is for single program another extension could be made for multiple programs running on a processor.
References


