9-2011

On Co-Optimization Of Constrained Satisfiability Problems For Hardware Software Applications

Kunal Ganeshpure
University of Massachusetts - Amherst

ON CO-OPTIMIZATION OF CONSTRAINED SATISFIABILITY PROBLEMS FOR HARDWARE SOFTWARE APPLICATIONS

A Thesis Presented

by

KUNAL GANESHPURE

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2011

Electrical and Computer Engineering Department

ON CO-OPTIMIZATION OF CONSTRAINED SATISFIABILITY PROBLEMS FOR HARDWARE SOFTWARE APPLICATIONS

A Thesis Presented

by

KUNAL GANESHPURE

Approved as to style and content by:

Sandip Kundu, Chair

Maciej Ciesielski, Member

Israel Koren, Member

Prashant Shenoy, Member

C. V. Hollot, Department Head
Electrical and Computer Engineering Department

My Parents

ACKNOWLEDGMENTS

I owe my deepest gratitude to my adviser Prof. Sandip Kundu for his support and mentorship, and for giving me an opportunity to address problems at various levels of design abstractions during the course of my graduate studies at the University of Massachusetts Amherst. This work would have been impossible without his help and guidance. I really appreciate the insightful comments and suggestions from my dissertation committee members Prof. Maciej Ciesielski, Prof. Israel Koren and Prof. Prashant Shenoy.

It is a pleasure to thank the members of the VLSI CAD Lab at the University of Massachusetts Amherst for their time and effort for engaging in in-depth technical discussions with me. I would also like to thank my friends for their support and help during the course of my PhD. I am sincerely thankful to members of the Electrical and Computer Engineering Department at University of Massachusetts Amherst for their help.

I am grateful to my parents for their constant support and encouragement with helping me to reach this goal.

I am thankful to Semiconductor Research Corporation for their funding during the course of my doctoral work.

ABSTRACT

ON CO-OPTIMIZATION OF CONSTRAINED SATISFIABILITY PROBLEMS FOR HARDWARE SOFTWARE APPLICATIONS

SEPTEMBER 2011

KUNAL GANESHPURE

B.E., THE MAHARAJA SAYAJIRAO UNIVERSITY OF BARODA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Sandip Kundu

Manufacturing technology has permitted an exponential growth in transistor count and density. However, making efficient use of the available transistors in the design has become exceedingly difficult. Standard design flow involves synthesis, verification, placement and routing followed by final tape out of the design. Due to the presence of various undesirable effects like capacitive crosstalk, supply noise, high temperatures, etc., verification/validation of the design has become a challenging problem. Therefore, having a good design convergence may not be possible within the target time, due to a need for a large number of design iterations.

Capacitive crosstalk is one of the major causes of design convergence problems in deep sub-micron era. With scaling, the number of crosstalk violations has been increasing because of reduced inter-wire distances. Consequently only the most severe

crosstalk faults are fixed pre-silicon while the rest are tested post-silicon. Testing for capacitive crosstalk involves generation of input patterns which can be applied post-silicon to the integrated circuit and comparison of the output response. These patterns are generated at the gate/*Register Transfer Level* (RTL) of abstraction using *Automatic Test Pattern Generation* (ATPG) tools. In this dissertation, an *Integer Linear Programming* (ILP) based ATPG technique for maximizing crosstalk induced delay increase at the victim net, for multiple aggressor crosstalk faults, is presented. Moreover, various solutions for pattern generation considering both zero as well as unit delay models is also proposed.

With voltage scaling, power supply switching noise has become one of the leading causes of signal integrity related failures in deep sub-micron designs. Hence, during power supply network design and analysis of power supply switching noise, computation of peak supply current is an essential step. Traditional peak current estimation approaches involve addition of peak current associated with all the CMOS gates which are switching in a combinational circuit. Consequently, this approach does not take the Boolean and temporal relationships of the circuit into account. This work presents an ILP based technique for generation of an input pattern pair which maximizes switching supply currents for a combinational circuit in the presence of integer gate delays. The input pattern pair generated using the above approach can be applied post-silicon for power droop testing.

With high level of integration, *Multi-Processor Systems on Chip* (MPSoC) feature multiple processor cores and accelerators on the same die, so as to exploit the instruction level parallelism in the application. For hardware-software co-design, application programming model is based on a *Task Graph*, which represents task dependencies and execution/transfer times for various threads and processes within an application. Mapping an application to an MPSoC traditionally involves representing it in the form of a task graph and employing static scheduling in order to minimize the sched-

ule length. Dynamic system behavior is not taken into consideration during static scheduling, while dynamic scheduling requires the knowledge of task graph at run-time. A run-time task graph extraction heuristic to facilitate dynamic scheduling is also presented here. A novel game theory based approach uses this extracted task graph to perform run-time scheduling in order to minimize total schedule length.

With increase in transistor density, power density has gone up substantially. This has lead to generation of regions with very high temperature called *Hotspots*. Hotspots lead to reliability and performance issues and affect design convergence. In current generation *Integrated Circuits* (ICs) temperature is controlled by reducing power dissipation using *Dynamic Thermal Management* (DTM) techniques like frequency and/or voltage scaling. These techniques are reactive in nature and have detrimental effects on performance. Here, a look-ahead based task migration technique is proposed, in order to utilize the multitude of cores available in an MPSoC to eliminate thermal emergencies. Our technique is based on temperature prediction, leveraging upon a novel wavelet based thermal modeling approach.

Hence, this work addresses several optimization problems that can be reduced to constrained max-satisfiability, involving integer as well as Boolean constraints in hardware and software domains. Moreover, it provides domain specific heuristic solutions for each of them.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Moore's law is a driver for semiconductor industry. It predicts a doubling of the number of transistors on a chip every two years. As a result, it has become possible to obtain an substantial increase in density and reduction in cost of modern ICs. However, device scaling has also lead to various non-idel effects, which have escalated test and validation complexity. Consequently, it is becoming increasingly difficult to scale performance proportionately.

Due to decreasing process geometries and increasing operating frequencies, capacitive crosstalk has become one of the leading causes of circuit marginality failures in current generation designs. Owing to a higher coupling capacitance to overall capacitance ratio, long signal nets are highly susceptible to crosstalk faults. Moreover, a typical long signal net can also couple with many other nets leading to multiple aggressors crosstalk scenario. It may be impossible to activate all aggressors logically and simultaneously to constructively induce maximum crosstalk delay at the victim net during pattern generation. Therefore, activating a maximal subset of aggressors, weighted by actual coupling capacitance value, in close temporal proximity of the victim net transition, is one of the main goals of pattern generation. In addition, the above pattern generation problem also involves determining an input signal assignment so as to propagate the fault effect at the victim to the primary output. Hence, this max-satisfiability problem is constrained by fault effect propagation condition. In this work, novel ATPG solutions for multiple aggressor crosstalk faults for zero and unit delay models are presented. Moreover, we also compare the magni-

tude of crosstalk induced delay at the victim net for the above approaches. In our solution, maximal aggressor excitation is achieved using a novel 0-1 *Integer Linear Programming* (ILP) formulation, while multiple solutions for fault effect propagation are presented. These solutions involve either a divide and conquer approach which uses traditional stuck-at fault ATPG, or an approach involving generation of additional ILP constraints thus forming an integrated ILP formulation achieving both maximal aggressor excitation and fault effect propagation. For unit delay model, the effect of gate delays is taken into account by using a circuit transformation step.

As a result of voltage scaling, current generation designs in the deep sub-micron era have become more sensitive to power supply noise. Excessive noise due to improperly designed power supply network leads to performance and reliability issues. Hence, it is imperative that supply network design be done carefully. Peak supply current computation is central to power rail design and power supply switching noise analysis. Traditionally, peak supply current is computed by adding the peak switching current from all CMOS gates in a combinational circuit. If temporal and Boolean relationships are taken into consideration, then there is a significant scope for improvement in this approach. Due to logical relationship between patterns appearing at the input to a gate in a combinational circuit, worst case switching current in a subset of gates may prevent some other subset of gates from having the worst case. Moreover, the switching events may be spaced out in time due to the effect of gate delays, thus lowering the peak current. Consequently, in this work, we also take integer gate delays into consideration. Further, it has been observed that, a faster and more accurate solution is obtained when gate delays are taken into account, as it reduces the size of individual problem instances to be solved. Finally, peak current waveform generated by the proposed solver is compare against SPICE simulation to demonstrate effectiveness of the proposed solution.

With high level integration, *Multi-Processor Systems on Chip* (MPSoC) have become commonplace. MPSoC hardware platform consists of multiple heterogeneous cores communicating via a shared communication back plane. An application is mapped to an MPSoC platform using a programming model which is based on a *Task Graph*. In the task graph representation for an application, a node represents an operation to be scheduled on a core, while the edges represent the communication between these operations. Computation and communication durations act as weights for the nodes and edges, respectively. Moreover, each of the nodes also have an associated type denoting the core architecture on which they can be scheduled. A scheduling problem involving task to processor assignment to minimize the total schedule length arises, as there may be fewer cores than tasks that may run in parallel on an MPSoC. During hardware-software co-design, such scheduling is done statically based on estimated execution times. Dynamic scheduling is better as static scheduling makes a program non-portable. An embedded game theory based dynamic feedback driven task scheduling is presented in this work. This scheduling approach is based on real execution times extracted on the fly at run-time. As dynamic scheduling requires the knowledge of task graph, one of the key challenges is the run-time discovery of the task graph. Moreover, due to the limited computation capabilities of the processor cores in the MPSoC, efficient low-overhead scheduling algorithms that execute in real time are needed to be developed. Our approach is based on program phase behavior which is used for run-time discovery of task graph. This is based on the observation that an application executing on the MPSoC goes through multiple phases during its lifetime. During a stable phase, an application executes the same task graph (*Phase Graph*) repeatedly for a large number of iterations. Hence, the proposed approach will detect when an application is going through a phase and extract the phase graph at run-time. This extracted phase graph will be subsequently used for dynamic scheduling.

Increase in transistor density has lead to a substantial increase in power density. As a result, there has been generation of regions with very high temperature called *Hotspots*. Thermal hostpots are responsible for major circuit marginality related issues and adversely effect reliability and performance. In current generation integrated circuits, *Dynamic Thermal Management* (DTM) techniques like frequency and/or voltage scaling are used to eliminate hotspots by controlling power dissipation. As these techniques are based on measurement of the current temperature followed by action, they are reactive in nature and have detrimental effect on performance. In this work, predictive thermal aware task migration is proposed, where we utilize the multitude of cores available in an MPSoC to eliminate thermal emergencies. In order to achieve this goal we propose a run-time temperature prediction technique. This technique is used by a run-time look-ahead based branch-and-bound scheduling heuristic which is used to eliminate thermal emergencies while minimizing schedule length. A delay insertion technique is used to remove task execution overlaps which cause thermal emergencies, in the case that task migration fails. Finally, we also propose an ILP based scheduling heuristic which achieves the above goals statically. The above solutions leverage upon a wavelet based thermal modeling approach, which is used to characterize the system thermal response.

Therefore, in this work we propose solutions to several optimization problems in various domains, which can be reduced to constrained max-satisfiability. The constraints and the corresponding solutions are very much domain specific. The rest of the document deals with each of the above problems in separate chapters. Chapter 2 explains the crosstalk ATPG technique. This is followed by pattern generation technique for maximizing power supply currents in Chapter 3. Chapter 4 presents system level task graph extraction and dynamic scheduling. Then thermal aware task migration is presented in Chapter 5. Finally, we conclude in Chapter 6.

CHAPTER 2

ON ATPG FOR MULTIPLE AGGRESSOR CROSSTALK FAULTS

2.1 Introduction

There has been a significant increase in signal integrity related failures due to increase in switching speed and circuit density [65]. As a result of the worsening of sidewall coupling capacitance, severe design and test related problems have been created, which are known to be aggravated by variation in the fabrication process [65]. One of the major causes of signal integrity related problems in deep sub-micron technology has been attributed to capacitive crosstalk [109]. Crosstalk faults are caused by parasitic coupling between adjacent signal nets. These kind of faults are common in nets that have weaker drivers relative to their adjacent peers [25] [24].

Crosstalk noise can be classified into crosstalk induced glitches and crosstalk induced delays and speed-ups. Crosstalk induced glitches are caused when the victim net remains in a static state and one or more aggressor nets are switching, while crosstalk induced speedups/delays are caused when both the aggressor(s) and victim nets have simultaneous or near simultaneous transitions. There will be an increase in victim net transition delay if the aggressor and the victim nets switch in the opposite direction, while aggressors and victim switching in the same direction will lead to victim net transition speed-up. The amplitude and the width of the glitch, delay or speed-up introduced depends mainly, among other factors, on relative arrival times of signal transition at the aggressors and victim nets and the amount of coupling capacitance. Timing and functional errors can be caused because of these effects.

Conventional crosstalk fault ATPG is done by generating patterns to sensitize the victim node to an observable output. The delayed transition at the victim net may or may not be observable at the primary output. This requirement is met using a set of auxiliary conditions.

Either redesign or fault detection using test pattern generation is used to fix crosstalk fault sites which are extracted during design validation phase. Some of the redesign steps includes resizing drivers, rerouting signals and shielding interconnects with power distribution nets. Redesign is expensive in terms of design effort because of stringent area and performance requirements, and its effectiveness can be easily offset by process variations. Moreover, overdesign can happen due to the presence of false positives in fault site extraction. Crosstalk Fault ATPG should generate patterns which are able to maximize the total crosstalk induced delay and propagate the fault effect to the primary output. In order to robustly propagate and capture the fault effect at a scan cell, maximal victim delay excitation during manufacturing test [26] is important.

In this work, an ATPG technique to generate a vector pair which causes maximal crosstalk induced delay increase at the victim net and propagates the fault effect to a primary output, is presented. False positives can be removed by pruning the fault list using the maximum delay obtained from this technique. Firstly, we present a divide and conquer approach where maximal aggressor excitation is modeled as an ILP formulation constrained by fault effect propagation condition, obtained from stuck-at fault ATPG. In the second approach we provide a complete solution by generating single ILP formulation for both maximal aggressor excitation and fault effect propagation. In spite of having a longer execution time, this solution gives the absolute worst case crosstalk induced delay. Zero and unit delay assumptions are shown for each of the above solutions. Unit delay circuits are converted to zero delay circuits using a circuit transformation technique. The results indicate that

percentage of total capacitance that can be switched varies from 75-100% for zero delay and 30-80% for variable delay case while propagating the fault effect to the primary output. The fact that zero delay model tends to overestimate the impact of crosstalk is evident from the comparison of results obtained for zero and unit delay models. Moreover, comparison between integrated ILP formulation and divide and conquer for both the cases show that integrated ILP formulation is consistently better while taking a longer execution time as compared to the faster divide and conquer solution.

Rest of the Chapter is organized in the following manner. Next Section 2.2 presents a review of previous work. This is followed by Section 2.3 which describes the problem statement and shows various pathological scenarios. Then Section 2.4 describes the proposed solution. Finally results for ISCAS85 benchmark circuits are shown in Section 2.5 and we conclude in Section 2.6.

2.2 Related Work

It has been firmly established that crosstalk noise induced errors cannot be ignored in deep sub-micron technology [109]. A multitude of crosstalk ATPG techniques have been studied in literature. A multiple aggressor crosstalk ATPG solution has been proposed by Bai *et al.* [15]. In their technique, firstly an implication graph is used to determine a maximal set of aggressors that could be switched under the given Boolean constraints. This is followed by using a modified version of PODEM to determine the pattern satisfying the worst case transitions on the feasible aggressor set and facilitate fault propagation. Consequently, the maximal aggressor excitation problem is divided into two independent maximization sub-problems: (i) maximal feasible aggressor set generation and (ii) generation of pattern pair for worst case feasible aggressor switching. Thus the final solution is sub-optimal and underestimates the worst case crosstalk noise excitation. Moreover, the use of zero delay model gives a

gross overestimation of maximal crosstalk noise, as will be seen from our results. An ATPG technique for crosstalk induced glitches is presented by Lee *et al.* [69]. They do not provide a solution for crosstalk induced delay. ATPG solutions for multiple-aggressor crosstalk scenario cannot employ traditional techniques for delay testing which have been used for crosstalk induce delay faults. The problem of generating two-vector test which excites crosstalk induced glitches has been addressed by Chen *et al.* [26]. Fault effect is propagated to an output flip-flop such that it has the maximum amplitude. It uses static as well as dynamic signals like transition and pulse during pattern generation. Multiple-aggressor crosstalk scenario cannot be handled by this technique. Kundu *et al.* [64] present a technique for timed test pattern generation for CMOS domino circuits. A timed ATPG based algorithm to generate patterns for testing crosstalk induced delay faults in static CMOS circuit has been presented by Paul *et al.* [83]. Even tough, both [64] and [83] consider multiple-aggressor scenario, they employ computationally expensive circuit level timing simulations. Krstic *et al.* [63] present a genetic algorithm based test generation heuristic for crosstalk induced faults. Chen *et al.* [23] present a SAT based method for crosstalk ATPG. Here functional and timing information are considered to eliminate false transition combination for an aggressor victim pair. Shimizu *et al.* [97] presents a *Built in Self Test* (BIST) method to detect crosstalk faults. Ganeshpure *et al.* [44] and [45] present an ILP based technique for multiple aggressor crosstalk ATPG.

A crosstalk analysis methodology is presented by Zachariah *et al.* [65]. They do not consider the ATPG aspect. Multiple aggressor crosstalk scenario has not been considered in other crosstalk analysis models [47] [25].

It can be seen that most of the prior techniques either do not consider the effect of multiple aggressors on the victim node or do not take circuit delays in account during pattern generation process. The solutions to these problems are specifically addressed in this work.

2.3 Problem Statement

Crosstalk maximization, like conventional stuck-at fault ATPG, involves controllability and observability. The controllability and observability problems involve aggressor excitation for maximal crosstalk induced delay at victim and victim fault effect propagation, respectively. The following paragraphs explain the above two problems in detail.

- *Switching Aggressors for Victim Delay Maximization:* For a victim net coupled with multiple aggressors, in order to produce maximum crosstalk induced delay at the victim, a maximal subset of aggressors must be switched in the direction opposite (*Desired Direction*) to that of victim net. Because of circuit Boolean constrains it may not be possible to switch all the aggressors in the desired direction. Moreover, it is imperative that the maximal aggressor set and the victim net should also switch in close temporal proximity to induce maximal coupling noise. In order to address this max-satisfiability problem [47] [43], coupling weights are assigned to each of the aggressors for both $0 \rightarrow 1$ and $1 \rightarrow 0$ switching directions. These *Coupling Weights* represent the noise induced at victim net due to aggressor transitions. Then, a search is performed to determine a valid aggressor and victim switching configuration so as to maximize the total coupling weight. The coupling capacitance between the aggressors and victim is used to determine the coupling weights assigned to the aggressors.

- *Victim Fault Effect Propagation to Primary Output:* The generated pattern must propagate the delay fault effect induced at the victim net to an observable primary output, in addition to maximal noise excitation. If the fault effect passes through a strong driver gate, it may get attenuated. Moreover, the fault effect may get lost if it reaches the primary output too early to meet the setup time requirement for the output flip-flop, due to the presence of a fast path. In this

work we assume the following: (i) the fault effect does not get attenuated appreciably while passing through various gates and (ii) the delay for various paths from the victim net to the primary output of the circuit does not have a large variation, thus making the observability of delay fault at the output independent of the propagation path. Current design considerations like combinational path balancing and transistor sizing support the above assumptions.

Next, we explain the multiple-aggressor crosstalk ATPG problem with in the Example 2.3.1 shown below.

Example 2.3.1. Consider a multiple aggressor crosstalk scenario in which the victim net v is capacitively coupled with aggressors $a1$, $a2$, $a3$ and $a4$ as shown in Fig. 2.1. The aggressor lines $a1$, $a2$, $a3$ and $a4$ are driven by gates $n1$, $n2$, $n3$ and $n4$, while gate $n0$ drives victim net. Coupling weights are indicated by the numbers in the boxes associated with the aggressors. The aggressors and victim switch simultaneously because of the zero delay assumption for all the gates in the circuit. Magnitude of coupling capacitance between various aggressors and the victim determines the coupling weight for the aggressor net. With increase in coupling weight, the delay impact on the victim net also increases. The sum of the coupling weights of all the aggressors switching in the desired direction determines the total delay introduced at the victim net. Various scenarios for crosstalk fault pattern generation are explained in the subsequent sections.

- *Scenario 1:* Here a greedy approach is followed by applying the vector pair $\{0, 0, 1, \downarrow, \downarrow, 1, 1\}$ at the inputs $\{i1, i2, i3, i4, i5, i6, i7\}$. As a result the aggressor $a2$ (with highest coupling weight of 100) and $a3$ (coupling weight = 20) are switched in the desired direction ($\{a2, a3, v\} = \{\uparrow, \uparrow, \downarrow\}$) due to the nodes $i4$ and $i5$ transitioning from high to low. A total coupling weight of $120 = (100 + 20)$ is induced at the victim. A slow-to-rise fault is induced at the input of gates $n6$

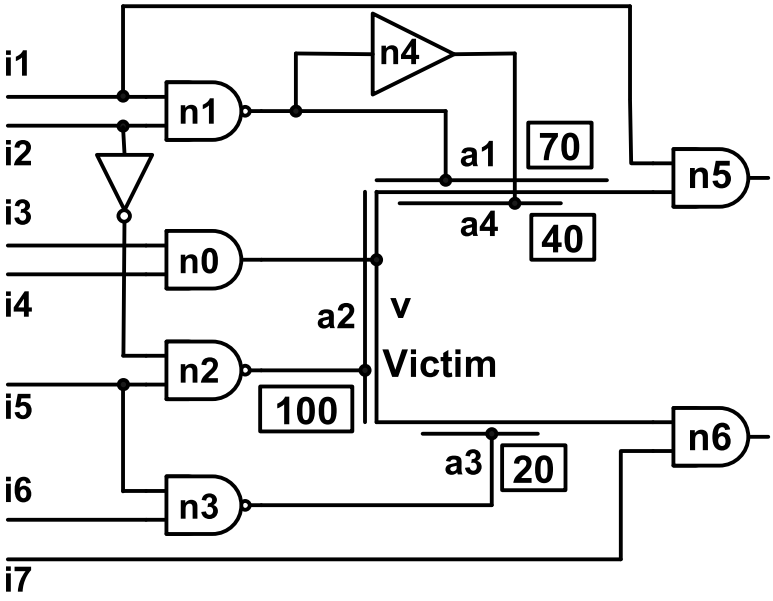


Figure 2.1. Multiple aggressor crosstalk scenario under zero delay model [42]

and $n5$, as both aggressors $a2$ and $a3$ couple to the nets connected to these gate inputs. By setting the input $i7$ to 1, we can propagate the fault effect to the primary outputs via gate $n6$.

- *Scenario 2:* Now consider the scenario where aggressors $a1$, $a3$ and $a4$ are switched in the desired direction ($\{a1, a3, a4, v\} = \{\uparrow, \uparrow, \uparrow, \downarrow\}$), due to another input vector pair $\{\downarrow, 1, 1, \downarrow, \downarrow, 1, 1\}$ which produces a total coupling weight of 130. This illustrates that maximal aggressor switching cannot be achieved by using a greedy approach. The input of gates $n5$ and $n6$ is delayed due to application of the aforementioned pattern. It can be seen that the fault effect is squashed at gate $n5$ by a controlling side input. Moreover, this pattern will not propagate the fault effect through $n6$, if it has a large slack. This example illustrates that the propagation problem is an integral part of the max-satisfiability problem involving switching maximal aggressor weight.

- *Scenario 3:* A total coupling weight of 120 is induced when the aggressors $a2$ and $a3$ switch in desired direction ($\{a2, a3, v\} = \{\uparrow, \uparrow, \downarrow\}$) due to the vector pair $\{1, 0, 1, \downarrow, \downarrow, 1, 1\}$. This fault effect can be propagated through both the AND gates $n5$ and $n6$ as their side inputs are 1. As this pattern accomplishes the propagation objective while creating a relatively large crosstalk effect, it is better than the previous two. This shows that multiple aggressor crosstalk fault test generation problem is qualitative in nature. ■

The importance of considering a combination of gate delays and logical constraints for maximal aggressor switching and fault propagation is illustrated by the following Example 2.3.2. The presence of gate delays makes the problem more constrained as now the aggressors need to be switched in a close temporal proximity to switching time of the victim net, while following the circuit Boolean constraints.

Example 2.3.2. A circuit with aggressors $a1$, $a2$ and $a3$ coupled to the victim net v is shown in the Fig. 2.2. The coupled aggressor lines $a1$, $a2$ and $a3$ are driven by gates $n1$, $n4$ and $n6$ while gate $n5$ drives victim net (v). The numbers present below the gate names represent the associated delays. Because of integer delay values, a net can only switch at integer times. If a pair of aggressor and victim nets switch instantaneously in the desired direction, then the corresponding delay introduced at the victim is indicated by the coupling weights represented by numbers in the box associated with the aggressors. In this example we assume that, if aggressor and victim transitions are not temporally aligned, there is no coupling and hence crosstalk induced delay. Next, pattern generating problem for the above scenario is discussed for various cases.

- *Scenario 1*: Consider the scenario when the vector pair $\{\uparrow, \uparrow, 1, \downarrow, \uparrow\}$ is applied to the inputs $\{i1, i2, i3, i4, i5\}$. As a result aggressors $a1$ (coupling weight = 100) and $a2$ (coupling weight = 130) are switched in the desired direction ($\{a1, a2, v\} = \{\uparrow, \uparrow, \downarrow\}$). The total coupling weight switched under zero delay

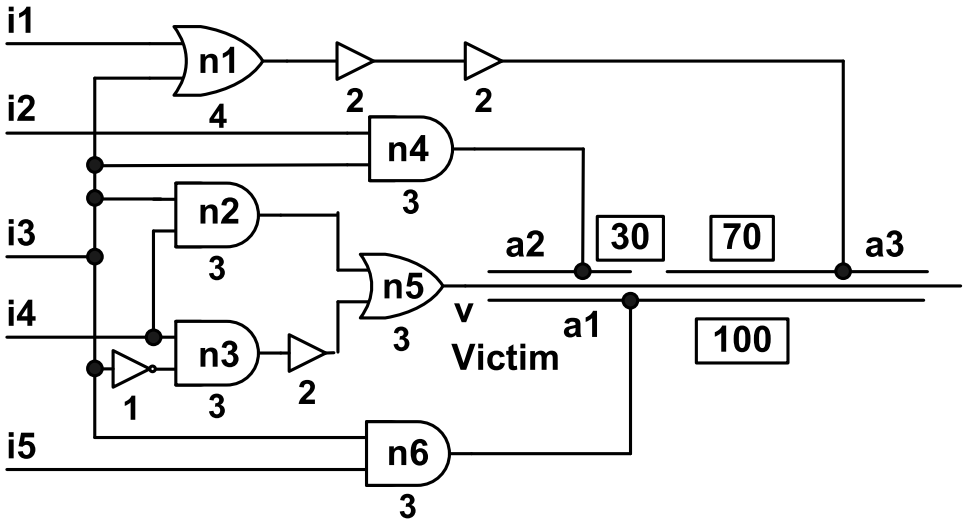


Figure 2.2. Multiple aggressor crosstalk scenario under integer delay model [42]

assumption is given by $100 + 30 = 130$. Due to delay of the gates $n2$ and $n5$, the aggressors $a1$ and $a2$ switch at time 3 and the victim v switches at time 6, hence nullifying impact of the aggressors on the victim. Therefore, it can be seen that, a pattern that excites high aggressor switching in zero delay model may not do as well under integer delay model.

- *Scenario 2*: Now for the vector pair $\{\uparrow, \uparrow, 0, \downarrow, \uparrow\} = \{i1, i2, i3, i4, i5\}$, only one of the aggressors $a3$ (coupling weight = 70) is switched in the desired direction ($\{a3, v\} = \{\uparrow, \downarrow\}$). This will be considered a sub-optimal pattern under zero delay assumption, as it only switches a coupling weight of 70. On the contrary, the victim has a greater impact with full coupling weight of 70, as the switching event at aggressor $a3$ and victim v occur at the same time instance (at time instance 8). ■

The above example shows that, an over estimation of crosstalk noise will happen when patterns generated using zero delay model are used. Consequently, it is important to take gate delays into consideration while generating patterns in order to maximize crosstalk induced victim delay.

2.4 Proposed Approach

It can be seen from the previous section that maximal aggressor excitation is intractable as it involves the solution max-satisfiability problem. Heuristic techniques are mostly used for the solution of all intractable problems. We present various heuristics to address this problem in the subsequent sections.

This work presents two different approaches for the solution of this problem. A divide and conquer approach is presented in Section 2.4.3. In this approach maximal aggressor excitation and fault effect propagation problems are solved independently. In contrast, a single solution is developed for both maximal aggressor excitation and fault effect propagation using an integrated ILP formulation presented in Section 2.4.4. The integrated ILP formulation gives the absolute worst case crosstalk induced delay at the expense of a longer execution time as compared to divide and conquer. In order to compare the effect of gate delays on maximal noise generation, each of the above approaches are presented for zero and unit gate delay assumptions. In all the above approaches, ILP formulation is used for maximal aggressor excitation. This ILP formulation and the associated modeling approach which are used to obtain maximal aggressor switching condition, are presented in the next two sections.

2.4.1 Crosstalk Modeling

Fig. 2.4 shows a multiple aggressor crosstalk fault site consisting aggressor victim pair (a, v) coupled by a capacitance C_c. The aggressor does not get affected by a transition at the victim net as it is driven by a strong driver. On the contrary, the victim net is driven by a weak driver. The wire resistance and capacitance to ground for the victim net are given by R and C_g respectively. It is assumed that the transitions on the aggressor and victim nets happen between voltage levels 0 and V_m, instantaneously and simultaneously.

Consider an RC circuit with resistance R and capacitance C_{eq} connected in series. If a unit step voltage waveform varying from 0 to V_m is applied at the input, the voltage $V(t)$ across the capacitor is given by the following equation.

$$V(t) = V_m \cdot \left(1 - e^{(-t/R \cdot C_{eq})}\right) \tag{2.1}$$

Now the Equation 2.1 can be modified in the following way to obtain the time τ at which $V(t) = 0.5 \cdot V_m$.

$$\tau = 0.693 \cdot R \cdot C_{eq} \tag{2.2}$$

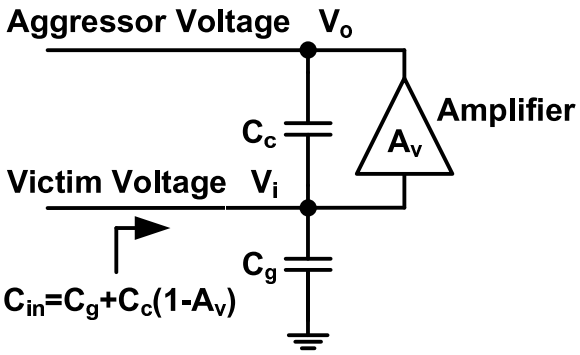


Figure 2.3. Circuit to obtain equivalent input capacitance using Miller effect [42]

It can be seen from Equation 2.2 that τ is dependent on the capacitance C_{eq}. Now, we will use Miller effect [100] in order to determine the equivalent capacitance C_{eq} corresponding to various aggressor victim transition scenarios. The effect of coupling between input and output nodes of an amplifier on its equivalent input impedance is determined using Miller effect. A crosstalk fault site can be mapped onto an amplifier with victim as the input node and aggressor as the output node as shown in Fig. 2.3. The amplifier is controlled by a unit gain voltage transfer function A_v, which can take values $\{-1, 0, 1\}$ depending on the direction of victim (input) and aggressor (output) transitions. Therefore, the following equation shows the equivalent input capacitance variation due to Miller effect.

$$C_{eq} = C_g + (C_c \cdot (1 - A_v)) \tag{2.3}$$

The arrival time τ is obtained by substituting C_{eq} from Equation 2.3 into the Equation 2.2 as shown below.

$$\tau = 0.693 \cdot R \cdot (C_g + (C_c \cdot (1 - A_v))) \tag{2.4}$$

Now, we obtain the crosstalk induced delay $\delta\tau$ for the following cases of aggressor and victim transitions.

- *Aggressor not switching:* For the reference case, where only the victim net switches we have $A_v = 0$ as the aggressor does not switch. The nominal arrival time τ_{nom} is therefore given by the following equation.

$$\tau_{nom} = 0.693 \cdot R \cdot (C_g + C_c) \tag{2.5}$$

- *Aggressor and victim switch in opposite direction:* As the aggressor and victim nets are switching in opposite direction ($A_v = -1$). This causes a crosstalk induced delay of $\delta\tau > 0$. The arrival time τ_{high} and the associated delay $\delta\tau$ are given by the following equation.

$$\tau_{high} = 0.693 \cdot R \cdot (C_g + 2 \cdot C_c) \tag{2.6}$$

$$\delta\tau = (\tau_{high} - \tau_{nom}) = 0.693 \cdot R \cdot C_c \tag{2.7}$$

- *Aggressor and victim switch in same direction:* As the aggressor and victim nets are switching in same direction, $A_v = 1$. Moreover, $\delta\tau < 0$ as the victim transition is sped-up. Consequently, the arrival time τ_{low} and the delay $\delta\tau$ are given by the following equations.

$$\tau_{low} = 0.693 \cdot R \cdot C_g \tag{2.8}$$

$$\delta\tau = (\tau_{low} - \tau_{nom}) = -0.693 \cdot R \cdot C_c \tag{2.9}$$

For the aggressor victim pair (a, v), a *Sign Variable* k and a *Coupling Weight* $W = 0.693 \cdot R \cdot C_c$ are introduced to simplify the effect of various scenarios for $\delta\tau$ as shown in Equations 2.5, 2.7 and 2.9 and to consider the effect of capacitive coupling, respectively. Therefore, the following equation is obtained to represent crosstalk induced delay variation.

$$\delta\tau = k \cdot W \tag{2.10}$$

Table 2.1. Sign variable value k [42]

Cases	k
Aggressor not switching	0
Aggressor and victim switch in opposite direction	1
Aggressor and victim switch in same direction	-1

Definition 2.4.1. A multiple aggressor crosstalk fault site '$XTFltSite$' in a combinational circuit 'C' is represented by a set of 'M' aggressor victim pairs (a_i, v) where aggressor 'a_i' is coupled to a victim 'v' through coupling capacitance 'C_i'.

Therefore, there corresponds a coupling weight $W_i = 0.693 \cdot R \cdot C_i$ and a sign variable k_i for each of the i aggressor victim pairs (a_i, v). Consequently, the total crosstalk induced delay at the victim is given by the following equation.

$$\delta\tau_i = k_i \cdot W_i \tag{2.11}$$

An aggressor net a_i, in a crosstalk fault site $XTFltSite$ with victim v and aggressors $a_i : (i = 0, 1, .., n)$, may be considered as a victim with its own set of aggressor

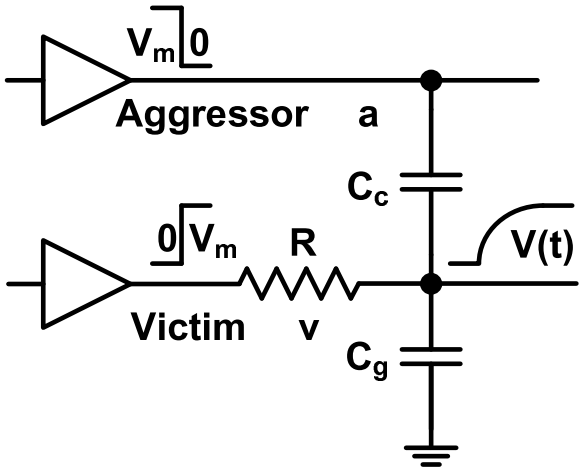


Figure 2.4. Coupled (a, v) pair and associated resistance and capacitance [42]

nets b_j : $(j = 0, 1, .., m)$, where $v \in \{b_j\}$. The crosstalk problem becomes a large coupled systems problem when these transitive relations are considered. The impact of crosstalk on the aggressors has been ignored in previous crosstalk ATPG solutions ([65],[109],[25],[26],[15] ,[69] ,[64] ,[83] ,[63] ,[23] ,[43],[45],[44],[47],[97] and [102]), in order to simplify the coupled-system problem. This coupled system scenario is not only complex for ATPG but also for simulation due to potential cyclical dependencies. In line with the above, this work also makes the same simplifying assumption.

Now, we determine the cumulative crosstalk induced delay $\Delta\tau$ at victim due to all the aggressor nets. This is given as the sum of the individual delays.

$$\Delta\tau = \sum_i (k_i \cdot W_i) \tag{2.12}$$

2.4.2 ILP Constraints for Combinational Circuits

In this section, we present a technique to generate Boolean functions for combinational circuits [41] using ILP constraints. Clausal description of the function of the gates [66] are used to form ILP equations of logic gates. For example, following set of implications can be used to describe an AND gate with inputs a, b and output c:

$$(a = 0) \Rightarrow (c = 0) \tag{2.13}$$

$$(b = 0) \Rightarrow (c = 0) \tag{2.14}$$

$$(c = 1) \Rightarrow (a = 1) \wedge (b = 1) \tag{2.15}$$

The following equations show how we can directly generate ILP constraints for the Boolean variables a, b and c using the implications in Equations 2.13 to 2.15 as shown in the following equations.

$$(a + (1 - c)) \geq 1 \tag{2.16}$$

$$(b + (1 - c)) \geq 1 \tag{2.17}$$

$$((1 - a) + (1 - b) + c) \geq 1 \tag{2.18}$$

For a combinational circuit, a union of ILP constraints generated for the individual gates form the complete set of Boolean constraints. For example, the complete set of ILP constrains for the circuit shown in Fig. 2.5 are generated as follows.

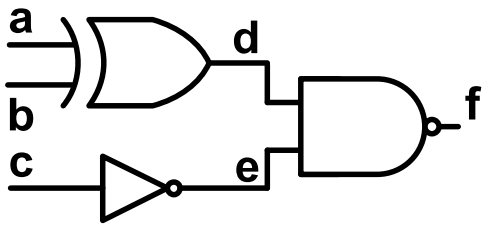


Figure 2.5. Combinational circuit with node names [42]

$$\text{NAND} \quad \begin{cases} (d+f) \geq 1 \\ (e+f) \geq 1 \\ ((1-d)+(1-e)+(1-f)) \geq 1 \end{cases} \tag{2.19}$$

$$\text{INVERTER} \quad \begin{cases} (c+e) = 1 \end{cases} \tag{2.20}$$

$$\text{XOR} \quad \begin{cases} (a+b+(1-d)) \geq 1 \\ ((1-a)+b+d) \geq 1 \\ (a+(1-b)+d) \geq 1 \\ ((1-a)+(1-b)+(1-d)) \geq 1 \end{cases} \tag{2.21}$$

2.4.3 Divide and Conquer

A fast divide and conquer approach is described in this section where maximal aggressor excitation and fault effect propagation problems are solved separately. In this approach maximal noise generation problem is solved by using an ILP formulation with the constrains obtained from stuck-at fault ATPG based approach used for fault effect propagation. A circuit bi-partitioning solution is used for structurally dividing the maximal aggressor excitation and fault effect propagation problems. In the first case this solution is applied for zero delay model while in the second case unit delay model is used. A circuit transformation based approach is used to account for gate delays. In rest of this chapter, we use *Zero Delay Stuck-at Framework* (ZDSF) and *Unit Delay Stuck-at Framework* (UDSF) to refer to zero and unit delay approaches using stuck-at fault ATPG, respectively.

2.4.3.1 ZDSF Approach

In this approach, separate solution for max-satisfiability and fault effect propagation problems are used. Fault effect propagation is achieved using stuck-at fault

ATPG while ILP is used for maximal aggressor excitation. A circuit bi-partitioned technique is used here to divide the above two problems and the two solutions are applied on the separate partitions of the circuit. The approach can be divided into the following steps, as shown below.

- *Circuit Partitioning:* Here the original combinational circuit is bi-partitioned such that the output logic cone of the victim belongs to the right partition and the input logic cones of aggressors and the victim nets belong to the left partition. There are multiple ways to bi-partition the circuit with these constraints. It can be observed that the cut line from bi-partitioning should pass through the victim net. Moreover, the cut line should pass through least number of circuit nodes for efficiency reasons which will become apparent in the subsequent discussion. For the left partition, the cut points represent circuit outputs, while they represent inputs for the right partition. It can be seen that the victim net itself is an input for the right partition while it is an output to the left partition.

- *Fault Effect Propagation:* Fault effect generated at the victim net propagates to the primary outputs after passing through the right partition. In this step, stuck-at fault ATPG is invoked to generate a pattern at the input of the right partition so as to enable fault effect propagation for a stuck-at 0 or 1 fault placed on the victim net. The Boolean constrains corresponding to this pattern represents the requirements for the cut points.

- *Maximal Noise Generation:* In this step the left partition which includes all the aggressors is targeted for pattern generation. The set of constraints derived from previous steps are transferred onto the outputs of the left partition. An ILP formulation with an objective function maximizing total crosstalk induced delay at the victim is generated for the left partition. Now, the above constraints in conjunction with those generated at the outputs of the left partition are solved

using an ILP solver to obtain the final input vector pair that leads to maximal aggressor switching.

The following sections provide a more detailed explanation of the above steps.

1. **Circuit Partitioning**

In this phase *Kernighan-Lin-Fiduccia-Mattheyses* (KLFM) algorithm [40] is used to bi-partition the original combinational circuit represented by a directed acyclic graph $G = (V, E)$.

Definition 2.4.2. KLFM partitions the set of vertices V, for a given directed acyclic graph $G = (V, E)$, into disjoint subsets U and Q so as to minimize the number of edges $\{(u, q) \in E \mid u \in U, q \in Q\}$ between U and Q.

In order to obtain an initial partition, the circuit is cut along the input to output level of the victim net such that all the nodes with level lower than or equal to that of the victim belong to the left partition while the nodes with level higher than the victim belong to the right partition. This is followed by moving the aggressors and their input cones that are part of the right partition into the left partition.

Next, the number cut points between the left and the right partitions are reduced by invoking KLFM. It has been observed in [50] that, with a random initial cut, KLFM algorithm gives fairly good result as compared to most other heuristics used for obtaining the initial cut. A better solution is obtained as the number of constraints in ILP are reduced for a smaller cut size. A comparison of results obtained with and without the application of KLFM algorithm is presented in Table 2.2.

Thus the original circuit is partitioned into left (U) and right (Q) partitions such that U contains the input logic cones of the aggressors and victim nets while Q contains the victim output logic cone.

2. **Stuck-at Fault ATPG**

A publicly available ATPG tool, ATALANTA [68] was used to perform stuck-at fault ATPG on the circuit Q, to determine the input node values of the right partition that propagate the fault effect to a primary output. There are multiple other ways of achieving this fault propagation. [70] suggested various slack based heuristics to guide fault propagation through longest paths. This input pattern which is generated by ATPG for Q acts as a constraint at the output of U. The partition U may not be able to satisfy these constraints or may satisfy them at the expense of compromising on maximal aggressor weight. Consequently, a single test vector to constrain U may not be optimal. However, the results for our heuristic solution shows that almost always, a single propagation pattern is satisfied by U with very high percentage of total aggressor weight switched.

3. **Maximal Noise Generation**

As the aggressors and victim net, and their input logic cones are present in the partition U, we use it for maximal noise generation. The partition U is duplicated into two copies U^0 and U^1, each representing logic values at time frames corresponding to the first and the second input vectors respectively, in order to generate an ILP formulation to represent aggressor victim switching. Thus the input vector pair $\langle I^0, I^1 \rangle$ is generated for the copies U^0 and U^1 combining the vectors I^0 and I^1 respectively. This is followed by renaming the node n in U to n^0 and n^1 in the copies U^0 and U^1 of the original circuit, respectively.

The outputs of partition U^1 are assigned the constraints derived from stuck-at fault ATPG. Only the inputs necessary for fault effect propagation are specified by ATALANTA, while is sets the rest to don't-care values. Consequently, all the outputs of U^0 and some of the node values at the output of U^1 are set to don't-cares.

ILP equations are formulated for U^0 and U^1 by using the procedure explained in Section 2.4.2. This is done for the nodes in the input logic cones of the aggressors, victim and the cut nodes that are not assigned don't-care values by stuck-at fault ATPG. In the ILP formulation, the Equation 2.12 acts as the objective function with a goal of maximizing the total number of aggressors switching in the opposite direction to the victim net. The final test vector pair $\langle I^0, I^1 \rangle$ is obtained from the solution to this objective function. The generation of this objective function is explained next.

After circuit transformation, the aggressor victim pairs (a_i, v) in U are renamed to (a_i^0, v^0) and (a_i^1, v^1) in the partitions U^0 and U^1 respectively. In order to indicate switching at the aggressor and victim nets, we define Boolean variables ξ_i and ξ which are $TRUE$ when a_i and v switch, respectively. We can interpret this conditions as aggressor and victim having different logic values in U^0 and U^1. This is represented by using an XOR gate in the following way.

$$\xi_i = (a_i^0 \oplus a_i^1) \tag{2.22}$$

$$\xi = (v^0 \oplus v^1) \tag{2.23}$$

In order to indicate that the aggressor a_i and victim v have different initial values in U^0, we define another Boolean variable ψ_i.

$$\psi_i = a_i^0 \oplus v^0 \tag{2.24}$$

In order to indicate the scenario where the aggressor and victim switch in the opposite direction, we set the Boolean variable χ_i to $TRUE$. Consequently, χ_i is $TRUE$ only when a_i is switching ($\xi_i = TRUE$) and v is switching ($\xi = TRUE$)

and they start from different initial values ($\psi_i = TRUE$). These implications are represented in the following set of constraints shown below.

$$\chi_i = \xi_i \wedge \xi \wedge \psi_i \tag{2.25}$$

In order to indicate the condition that the aggressor and victim are switching in the same direction, we define a Boolean variable π_i. Thus, when a_i is switching ($\xi_i = TRUE$) and v is switching ($\xi = TRUE$) and they start from the same initial value ($\psi_i = FALSE$), the variable π_i is set to $TRUE$. These conditions are given by the following set of constraints.

$$\pi_i = \xi_i \wedge \xi \wedge \overline{\psi_i} \tag{2.26}$$

Consequently, the following set constraints are obtained for the sign variable k_i.

$$k_i = (\chi_i - \pi_i) \tag{2.27}$$

The objective function maximizing crosstalk induced delay for an aggressor victim pair (a_i, v) with coupling weight W_i is obtained by substituting k_i in Equation 2.12.

$$max \left\{ \Delta\tau = \sum_i ((\chi_i - \pi_i) \cdot W_i) \right\} \tag{2.28}$$

2.4.3.2 UDSF Approach

In this approach test patterns are generated considering the effect of gate delays. The approach is divided into the following steps:

- *Circuit Transformation:* A time domain transformation of the circuit is used to incorporate unit delay model. As a result, the original circuit is transformed

such that there is a one to one correspondence between the gate outputs in the transformed circuit Z and the arrival times of the original circuit C represented by the graph $G = (V, E)$.

- *Fault Effect Propagation:* Similar to the ZDSF approach, in this step, stuck-at fault ATPG is invoked by placing a stuck-at 0 or stuck-at 1 fault on the victim line to generate an input pattern that propagates the fault effect at the victim to the primary outputs.

- *Maximal Noise Excitation:* Here, Boolean functions of logic gates are used to formulate ILP constraints as explained in Section 2.4.2. The ILP formulation is further constrained by using the input pattern derived from stuck-at fault ATPG. This is followed by the generation of an objective function that maximizes the total crosstalk induced delay at the victim. Thus, ILP constraints are generated for both the logic circuit and the maximal aggressor weight objective. Finally, an ILP solver is used to solve simultaneous logic constraints in order to maximize aggressor switching and consequently generate the final test vector pair.

Our approach is based on the assumption that the delay introduced by the vector pair will be large enough to not get subsumed in timing slacks and always causes an error at the output.

1. **Circuit Transformation:**

In order to obtain the transformed circuit Z, a time frame expansion is done to represent the circuit switching activity in the presence of unit gate delays [75]. Fig. 2.6 shows the example of C17 ISCAS85 benchmark circuit. The possible signal arrival times are represented at the node outputs. The delay of all the possible paths from input to a node is used to generate the list of arrival times for the node.

For example, the signal arrival times for the node $n23$ are given by $\{t_0, t_2, t_3\}$. Here the initial logic value is set at time t_0, signal arrival from the paths $i2 \rightarrow n16 \rightarrow n23$ and $i7 \rightarrow n19 \rightarrow n23$ is represented by the time t_2 and that from the paths $i3 \rightarrow n11 \rightarrow n16 \rightarrow n23$ and $i6 \rightarrow n11 \rightarrow n16 \rightarrow n23$ leads to the arrival time t_3.

Fig. 2.6 also shows the resultant expanded circuit. It can be seen that each of the gates is replicated as many times as the number of all the possible signal arrival times in the original circuit. *Time-slots* are assigned to the gates corresponding to each of the above signal arrival times. A time-slot represents the start of a time duration at which a gate output transitions to a new logic value. Consequently, the final logic value after the signal transition is represented by the logic value of a gate appearing at a particular time-slot. The set of all the integer times, corresponding to time-slots at which a particular gate n transitions, is given by $TimeSlot_n$. The gate inputs for each of the replicas appearing at a particular time-slot are connected to the gate outputs appearing in the previous time-slot. The maximum time-slot in Z at which a particular gate n appears is denoted by the variable $MaxTimeSlot_n$. The primary inputs I corresponding to the first and the second input vector can only appear in the time-slot $0t$ (I^0)and $1t$ (I^1). Consequently, final input vector pair is given by the combination $\langle I^0, I^1 \rangle$.

For example, the gate $n23$ is replicated thrice, hence appearing in the time-slots $0t$, $2t$ and $3t$ corresponding to the arrival times $\{t_0, t_2, t_3\}$. These replicated gates are renamed to $n23^0$, $n23^2$ and $n23^3$ in Z. The gates $n16^2$, $n19^2$ and $n16^1$, $n19^1$ supply inputs to the gate $n23^3$ and $n23^2$, respectively. Moreover, for the gate $n23$, by definition, $TimeSlot_{n23} = \{2, 3\}$ and $MaxTimeSlot_{n23} = 3$. Primary inputs in Z corresponding to the first and second vectors are $I^0 = \{i1^0, i2^0, i3^0, i6^0, i7^0\}$ in time-slot $0t$ and $I^1 = \{i1^1, i2^1, i3^1, i6^1, i7^1\}$ in time-slot $1t$, respectively.

Therefore the gate $n23$ attains an initial value at time-slot $0t$ represented by $n23^0$ on application of the initial vector to the inputs I^0. Now, due to the application

of the second vector to the inputs I^1, the gate $n23$ either transitions at time t_2 and attain a new logic value represented by $n23^2$, and/or transitions again at time t_3 to attain a new logic represented by $n23^3$.

Now, in order to maximize crosstalk, the aggressor and victim transitions at various time-slots have to be determined. In order to indicate these transitions, the aggressor and victim outputs at various time-slots are $XORed$. In our case ξ^1 and ξ^2 are high when the victim transitions at time-slot $1t$ and $2t$, respectively. The Boolean variables ξ^1 (ξ^2) are used to indicate a transition from initial value v_0 (v_1) to the final value v_1 (v_2). Consequently, ξ_i^1 and ξ_i^2 are used to represent the same for the aggressor a_i^1 and a_i^2. We can easily extend the zero delay model to general integer delays by adding unit delay buffers.

2. **Stuck-at Fault ATPG:**

To determine the input vector that propagates the fault effect at the victim to the primary outputs, stuck-at fault ATPG was performed on the transformed circuit using the stuck-at fault ATPG tool, ATALANTA [68]. For the victim net appearing at the $VictimTimeSlot$, T in Z, a stuck-at 0 fault is placed. Pseudocode 2.1 explains how the value of T is determined. A fault effect generated at the victim can be propagated to a primary output in multiple ways. As mentioned earlier, [70] propose various slack based heuristics to guide fault propagation through longest paths. In the transformed circuit Z, inputs I^1 are assigned the input pattern generated by stuck-at fault ATPG while the input I^0 are set to don't-cares. Only the inputs necessary for fault effect propagation are specified by ATALANTA, while it sets the rest to don't-cares. The complete input vector pair $\langle I^0, I^1 \rangle$ which maximizes the total aggressor weight, corresponding to the victim switching at T is generated by specifying the above input values as constraints to the ILP formulation.

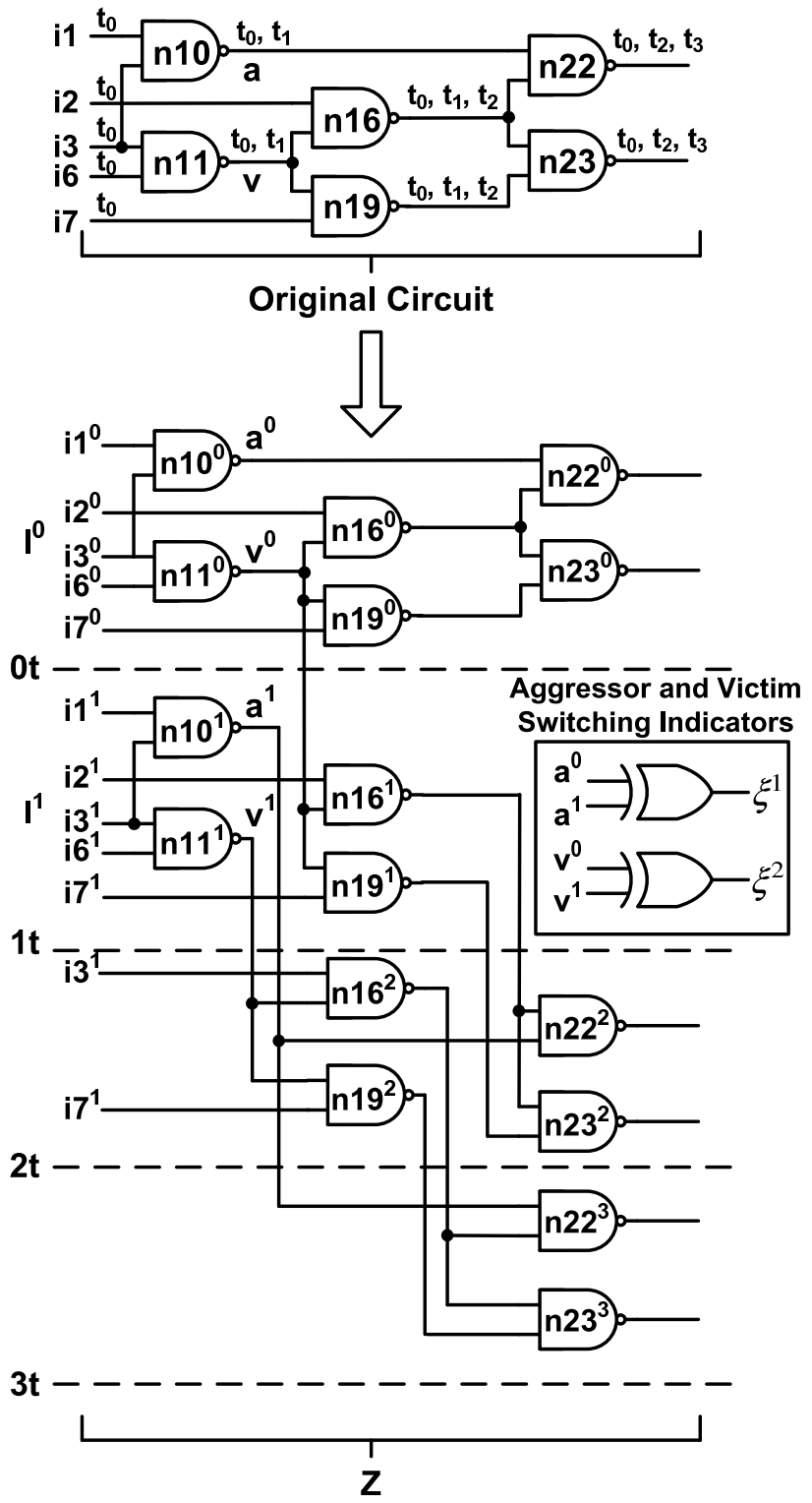


Figure 2.6. Circuit transformation for unit gate delays [42]

3. **Maximal Noise Generation:**

The difference between the arrival time of transitions at aggressor and victim nets is known as *Relative Arrival Time*. This time determines the magnitude of coupling between a pair of aggressor and victim nets. Based on the relative arrival time, a *Scaling Factor* r_i parameter is used to proportionally scale the coupling weight W_i. The range of relative arrival times of the aggressor and victim transitions for which crosstalk noise is induced at the Vitim net is represented by the $WindowSize$ parameter. It has been previously shown that, the effect of difference in arrival times of aggressor and victim transitions on crosstalk induced delay can be effectively modeled by using a window based approach. For a crosstalk fault site, the total crosstalk induced delay at the victim net has been characterized by Sasaki *et al.* [95] [94] using circuit simulation. SPICE simulations are performed for a crosstalk fault site consisting of coupled aggressor and victim nets in order to calculate $WindowSize$ and scaling factor r. Fig. 2.7 shows our experimental setup for evaluating the above parameters. In the above figure, it can be seen that the aggressor is driven by a large inverter (4 times the minimum size) while the victim is driven by a minimum sized inverter. A standard load consisting of 4 minimum sized inverters is connected at the outputs driving both the victim and aggressor nets. The nominal arrival delay for the victim transition is measured by switching only the victim net. Now, the victim and aggressors are made to transition in the opposite direction by applying the required inputs. The transition arrival time of the aggressor is varied in a range starting from a time before the victim starts transitioning to that after the victim transition ends. The increase in victim net delay with respect to the nominal arrival time is measured for each of the aggressor arrival times. The variation of crosstalk induced delay increase is shown in the Fig. 2.8. This delay increase shown in the figure has been normalized with respect to the maximum value. Delay is discretized to a value that is equal to the propagation delay of a minimum sized inverter loaded with

equivalent input capacitance of four minimum sized inverters. This discretization interval is called *Unit Delay Interval*. Now the set of scaling factors r is obtained by sampling the normalized delay increase curve at discrete time instances separated by unit delay intervals. The Fig. 2.8 also shows that $WindowSize$ is determined using the total number of points for which $r \neq 0$. In this case a $WindowSize$ of 5 is obtained corresponding to non-zero values of scaling factor r.

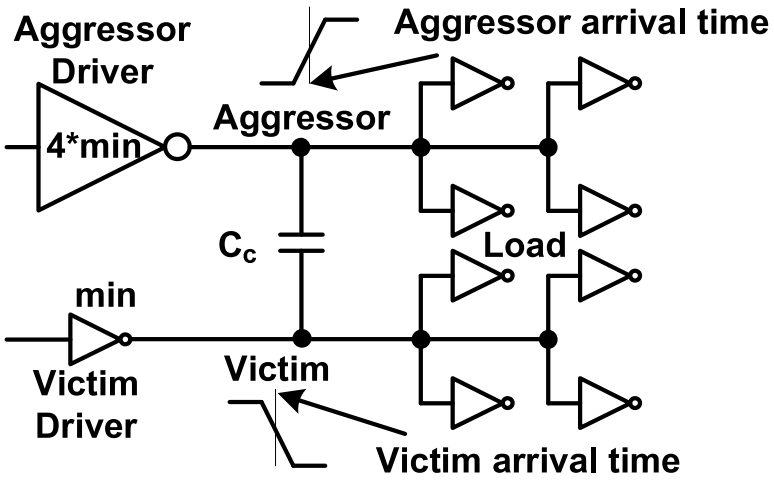


Figure 2.7. Experimental setup for calculating $WindowSize$ and scaling factor r [42]

We use a triangular approximation for the delay impact on victim relative to the time when signal transition occurs at the aggressor, for the solution presented in this work. Please note that such delay is a function of the waveform of the current injection through coupled capacitance. A triangular approximation of current provides a first order approximation in Taylor series expansion of a more complex waveform [59]. We use a triangular model for this work because it has been used by many authors previously. Consequently, it should be noted that our solution approach itself is agnostic to the shape of the delay waveform used here. The scaling factor values for various aggressor victim arrival time differences corresponding to a $WindowSize$ of 6 is shown in Fig. 2.9.

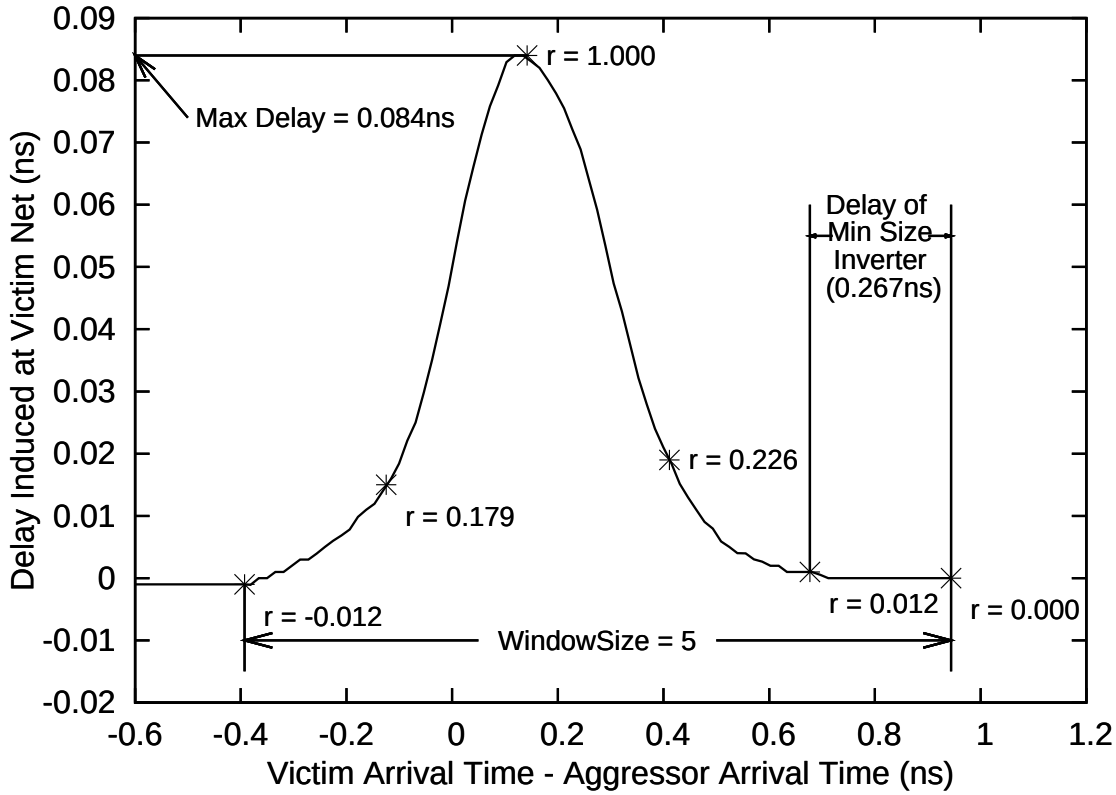


Figure 2.8. Variation of victim delay increase with the difference between victim and aggressor arrival times (Coupling capacitance = 20fF) [42]

For input logic cone of the victim net in the $VictimTimeSlot = T$ and aggressor nets in the time-slots within the $WindowSize$ duration of the victim net, we generate a set of ILP constraints. Moreover, additional constraints for maximal aggressor excitation are constructed in the following manner.

Consider an aggressor victim pair (a_i, v) within a crosstalk fault site $XTFltSite$. The aggressor and victim names a_i and v are renamed to a_i^t and v^t, after circuit transformation.

For the Victim net at the $VictimTimeSlot = T$ obtained from Line 3 of Pseudocode 2.1, the ILP constraints presented here in the Line 5 of Pseudocode 2.1.

Let victim transitioning at time-slot T and i^{th} aggressor transitioning at time-slot $x \in \{TimeSlot_{a_i} :\mid T - x \mid < (WindowSize/2)\}$ within $WindowSize$ of the victim

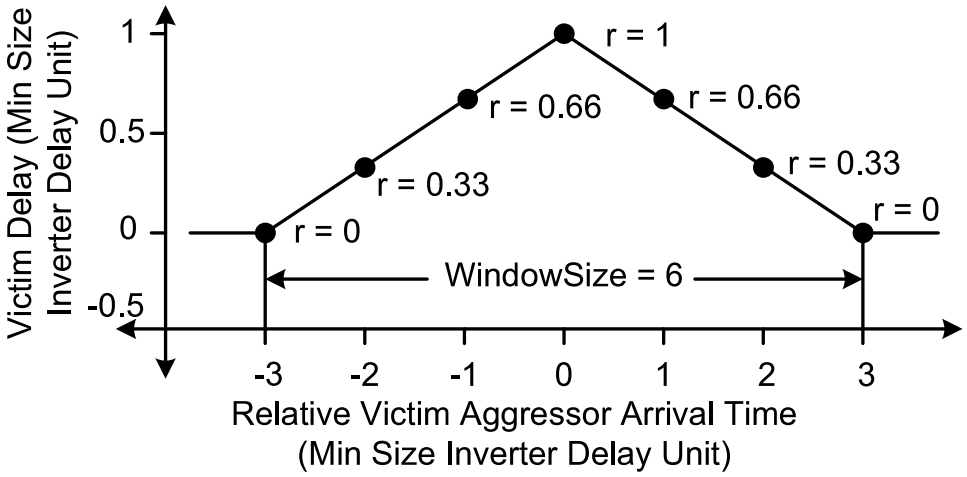


Figure 2.9. Triangular approximation to determine the $WindowSize$ [42]

transition, be represented by the Boolean variables ξ^T and ξ_i^x, respectively. Therefore we obtain the following set of constraints.

$$\xi_i^x = \left(a_i^x \oplus a_i^{x-1}\right) \tag{2.29}$$

$$\xi^T = \left(v^T \oplus v^{T-1}\right) \tag{2.30}$$

In order to indicate different final values of the aggressor a_i^x in the x^{th} time-slot and the victim v^T at the T^{th} time-slot, a set of Boolean variables ψ_i^x are set to $TRUE$. This constraint is represented by the following equation.

$$\psi_i^x = \left(a_i^x \oplus v^T\right) \tag{2.31}$$

In order to indicate the aggressor a_i^x in the x^{th} time-slot and the victim v^T at the T^{th} time-slot, switching in the opposite direction, we define a Boolean variable χ_i^x. Hence, if a_i^x is switching ($\xi_i^x = TRUE$) and v^T is switching ($\xi^T = TRUE$) and they

have different final values ($\psi_i^x = TRUE$), we set χ_i^x to be $TRUE$, This condition is represented in the following equation.

$$\chi_i^x = \xi_i^x \wedge \xi^T \wedge \psi_i^x \tag{2.32}$$

Now, in order to indicate the condition where the aggressor and victim nets switch in the same direction, we define a Boolean variable π_i^x. If a_i^x is switching ($\xi_i^x = TRUE$) and v^T is switching ($\xi^T = TRUE$) and they have same final values ($\psi_i^x = FALSE$), then we set π_i^x to $TRUE$. These conditions are represented by the following equation.

$$\pi_i^x = \xi_i^x \wedge \xi^T \wedge \overline{\psi_i^x} \tag{2.33}$$

Sign factor k_i^x for the aggressor victim pair (a_i, v) with aggressor a_i switching in time-slot x, is obtained using the following equations.

$$k_i^x = (\chi_i^x - \pi_i^x) \tag{2.34}$$

We define a set of real constants r_i^x which represent the scaling factor for various relative arrival times as a function of $\mid T - x \mid$ described in Fig. 2.9. The coupling weight W_i is scaled according to the scaling factor r_i^x, depending on various relative arrival times ($\mid T - x \mid$) of the aggressor and victim nets. For the victim in T^{th} time-slot, the objective (Equation 2.12) function for crosstalk noise maximization is given by the following equation.

$$max \left\{ \Delta\tau = \sum_i \left(\sum_x (\chi_i^x - \pi_i^x) \cdot r_i^x \cdot W_i \right) \right\} \tag{2.35}$$

Here: $x \in \{TimeSlot_{a_i} :\mid T - x \mid < (WindowSize/2)\}$

In order to prevent the victim in all the subsequent time-slots after the T^{th} $x \in \{TimeSlot_v : (T, MaxTimeSlot_v]\}$ from switching, additional set of constraints as shown below are generated.

$$v^x = v^{x-1} \tag{2.36}$$

In order to represent the Boolean constrains of the combinational circuit, ILP formulation is done for the input logic cone of the victim in subsequent time-slots $x \in \{TimeSlot_v : (T, MaxTimeSlot_v]\}$.

4. **Pseudocode:**

The algorithm used to determine worst case pattern pair is explained in Pseudocode 2.1. The circuit transformation described earlier is performed in Line 1. A for loop that iterates through all the time-slots of the victim net generating patterns using stuck-at fault ATPG and ILP is shown in Line 3. For the victim net the current $VictimTimeSlot$ value is determined here. The input pattern pair $\langle I^0, I^1 \rangle$ as well as the total coupling weight switched $weightSwitched$ are generated by the solution to the ILP formulation shown in the Line 6. A track of the maximum value of $weightSwitched$ and the corresponding pattern $\langle I^0, I^1 \rangle_{max}$ is kept by the if statement in Lines 7-10.

2.4.4 Integrated ILP Formulation With Error Propagation

In this step, the constraints for fault effect propagation are added to those for maximal noise generation in order to obtain a unified ILP formulation. Fault effect propagation constraints are obtained using circuit transformation. As this solution will produce an aggressor switching that will generate absolute maximum crosstalk induced delay at the victim net, given enough time, this is a complete solution. In order to compare the effect of delay on the absolute maximum crosstalk noise induced

Algorithm 2.1 $max\_xtalk\_delay(C, XtFltSite)$ [42]

1: $Z \leftarrow circuit\_xform(C)$
2: $maxWeightSwitched = 0$
3: **for all** $T \in TimeSlot_v$ **do**
4: $I^1 \leftarrow stuck\_at\_ATPG(Z, T)$
5: $ILPEqns \leftarrow generate\_ILP\_eqns(Z, I^1)$
6: $\left\{\left\langle I^0, I^1\right\rangle, weightSwitched\right\} \leftarrow solve\_ILP(ILPEqns)$
7: **if** $maxWeightSwitched \leq weightSwitched$ **then**
8: $maxWeightSwitched = weightSwitched$
9: $\left\langle I^0, I^1\right\rangle_{max} \leftarrow \left\langle I^0, I^1\right\rangle$
10: **end if**
11: **end for**
12: **return** $\left\langle I^0, I^1\right\rangle_{max}$

at the victim, this solution is presented for zero and unit delay assumptions. A circuit transformation is used to take gate delays into account. In rest of this chapter, we use *Zero Delay ILP Framework* (ZDIF) and *Unit Delay ILP Framework* (UDIF), to refer to the zero and unit delay approaches using an integrated ILP formulation, respectively.

2.4.4.1 ZDIF Approach

In this approach, a unified ILP formulation to solve both fault effect propagation and maximal aggressor excitation is used in order to address the sub-optimality of the approach presented in Section 2.4.3.1. Using this approach, we would eventually be able to obtain input pattern that will lead to absolute worst case crosstalk induced delay on the victim net. The following steps are used in this approach.

- *Circuit Transformation for Maximal Noise Generation:* In order to represent initial and final logic values, in this step, the original circuit is duplicated. This is followed by formulation of ILP equations to represent Boolean functions of logic gates for the input logic cones of the aggressors and victim nets.
- *Circuit Transformation for Fault Effect Propagation:* In order to generate ILP equations for fault effect propagation, circuit transformation is done for the out-

put logic cone of the victim net appearing in the circuit copy that represents the second time-slot. This circuit transformation involves the duplication of output logic cone of the victim net. As a result, the duplicated logic cone represents the faulty machine while the original logic cone represents the good machine. In order to represent the difference in the logic values between good and faulty machines for fault propagation, a D value is generated. Hence, for each of the gates in the duplicated fault propagation logic cones, these D values are generated by $XORing$ the original and duplicated gate outputs. These D value are propagated from victim net to the primary outputs by using an ILP formulation.

1. **Circuit Transformation for Maximal Noise Generation:**

   As shown in the Fig. 2.10, the original circuit $G = (V, E)$ is duplicated into two copies U^0 and U^1, each representing one time frame corresponding to the first and the second input vectors respectively. The input vector pair $\langle I^0, I^1 \rangle$ is formed by combining the input to the copies U^0 and U^1 correspond to pattern I^0 and I^1, respectively. Moreover, a node n in $G = (V, E)$ is renamed to $n^0(n^1)$ in $U^0(U^1)$.

   In order to maximizes the total number of aggressors switching in the opposite direction to the victim net we use the Equation 2.12 as the objective function. The final test vector pair $\langle I^0, I^1 \rangle$ is obtained from the solution to this objective function.

2. **Circuit Transformation for Fault Effect Propagation:**

   The output logic cone of the victim net in the copy U^1 is used for fault effect propagation. Consequently, the output logic cone of the victim given by Γ_v^1 corresponding the copy U^1 is duplicated to Γ_v^{1c}. The true and faulty logic values are represented using original logic cone Γ_v^1 and the duplicated logic cone Γ_v^{1c} respectively. Therefore, every node n^1 in Γ_v^1 is duplicated to form node n^{1c} in Γ_v^{1c} to represent true and faulty logic values. This leads to formation of the node pair given by (n^1, n^{1c}). For the

output logic cone of the victim net, each of the above pairs (n^1, n^{1c}) are $XORed$ to generate a set of D values denoted by n^{1d}.

The victim node $n11$ in original circuit is duplicated into $n11^0$ in the copy U^0 and to $n11^1$ in U^1, as shown in Fig. 2.10. The corresponding set of original and duplicated output logic cones for victim in U^1 consists of the nodes given by $\Gamma^1_{n11} = \{n16^1, n19^1, n22^1, n23^1\}$ and $\Gamma^{1c}_{n11} = \{n16^{1c}, n19^{1c}, n22^{1c}, n23^{1c}\}$, respectively. The D values generated using the above pair of nets is given by $\{n16^{1d}, n19^{1d}, n22^{1d}, n23^{1d}\}$.

ILP formulation for the victim net is done for the input, output and side input logic cone in U^1 and the input logic cone in U^0. Moreover, ILP formulation for the victim is also done for input logic cone in U^0 and U^1. This formulation done for maximal aggressor excitation and fault effect propagation is explained next.

The aggressor victim pair (a_i, v) in the original circuit are renamed to (a_i^0, v^0) and (a_i^1, v^1) in the circuit copy U^0 and U^1, respectively, after performing the first circuit transformation. For both the copies of the circuit, ILP formulation for maximal aggressor excitation is performed in the same way explained in Section 2.4.3.1.

The above set of constraints are augmented by the fault effect propagation constrains. The output logic cone of the victim net v is duplicated in the copy U^1. For the output logic cones in Γ^1, let Λ^1 be the set of all the nodes and Λ^1_o be the set of all the primary output nodes. Similarly, let Λ^{1c} and Λ^{1c}_o be the set of all the nodes and the set of all the primary outputs in the output logic cone Γ^{1c}, respectively. In Fig. 2.10 for Γ^1 these set of nodes are given by $\Lambda^1 = \{n16^1, n19^1, n22^1, n23^1\}$, $\Lambda^{1c} = \{n16^{1c}, n19^{1c}, n22^{1c}, n23^{1c}\}$ and $\Lambda^{1d} = \{n16_{1d}, n19_{1d}, n22_{1d}, n23_{1d}\}$, respectively. Similarly, we have the set of primary outputs $\Lambda^1_o = \{n22^1, n23^1\}$, $\Lambda^{1c}_o = \{n22^{1c}, n23^{1c}\}$ and $\Lambda^{1d}_o = \{n22^{1d}, n23^{1d}\}$.

Therefore D values Λ^{1d} are given by the following equations.

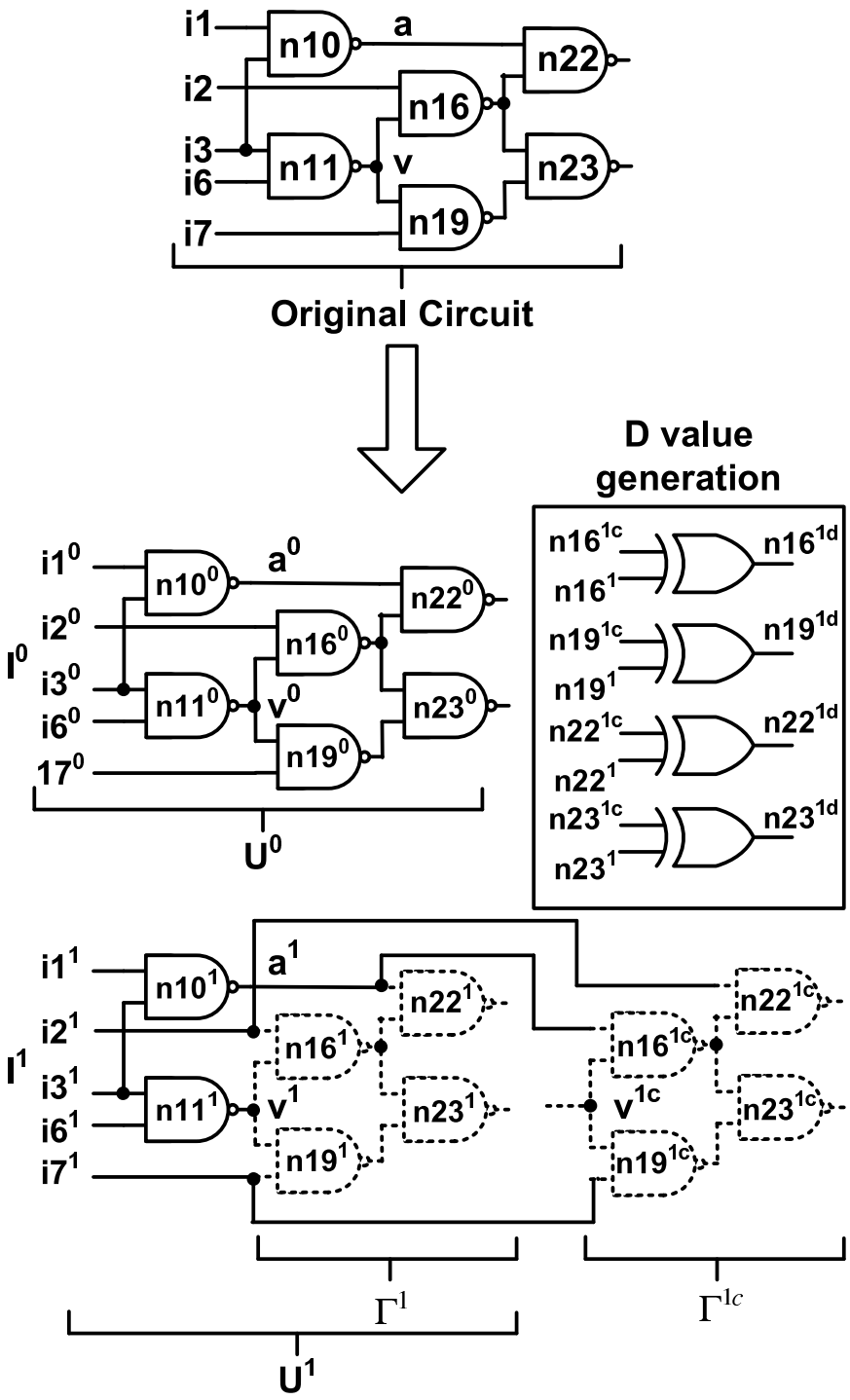


Figure 2.10. Circuit transformation for fault effect propagation in the absence of gate delays [42]

$$\Lambda^{1d} = (\Lambda^1 \oplus \Lambda^{1c}) \tag{2.37}$$

$$\Lambda_o^{1d} = (\Lambda_o^1 \oplus \Lambda_o^{1c}) \tag{2.38}$$

When, at least one of the gate inputs, which are a part of the output logic cone of the victim net, has a D value and all the other inputs are set to non-controlling values, a D value is generated at a gate output. As a result, the following implication is observed for a gate with output D value Λ^{1d} and inputs Λ_i^{1d}: $\Lambda^{1d} \Rightarrow \left(\Lambda_1^{1d} \vee \Lambda_2^{1d} \vee ...\right)$. Therefore fault propagation from the D values at the output Λ^{1d} to the inputs Λ_i^{1d} can be enabled by using the following equation.

$$\left(1 - \Lambda^{1d}\right) + \sum_i \left(\Lambda_i^{1d}\right) \geq 1 \tag{2.39}$$

Now we need to propagate this D value generated at the victim node to at-least one of the primary outputs Λ_o^{1d}. Consequently, the following constraint is generated, for the set of primary outputs Λ_o^{1d} in the output logic cone of the victim net.

$$\sum \Lambda_o^{1d} \geq 1 \tag{2.40}$$

2.4.4.2 UDIF approach

In this approach the zero delay assumption is remove and test patterns are generated for unit gate delay model. Moreover, a complete solution consisting of a unified ILP formulation for both maximal aggressor excitation and fault effect propagation is presented here. Consequently, we will be able to obtain input pattern that leads to absolute worst case crosstalk induced delay on the victim net, provided the ILP formulation is run for enough time. The following steps are involved in this approach.

- *Circuit Transformation for Gate Delays:* Circuit transformation is done on the original circuit to take the effect of unit gate delays into account.

- *Circuit Transformation for Fault Effect Propagation:* In order to generate conditions for fault effect propagation, circuit transformation is performed for the output logic cone of the victim appearing at a particular time-slot.

- *ILP Formulation:* An ILP formulation presented in Section 2.4.2 is used to represent the Boolean constrains generated by the above circuit transformations. Finally, an objective function maximizing the delay induced at the victim net is formed. A solution to the above ILP formulation gives the final vector pair.

1. **Circuit Transformation for Gate Delays:**

   The time domain expansion presented in Section 2.4.3.2 is used here. Thus, gate delays are taken into account by transforming the original circuit C into a new expanded circuit Z.

2. **Circuit Transformation for Fault Effect Propagation:**

   In this phase, fault effect propagation is achieved by using true and faulty logic values obtained by duplicating the output logic cone of victim net switching at time-slot $VictimTimeSlot = T$ in the transformed circuit Z.

   The set of nodes in the output logic cone Γ^T of the victim appearing at the time-slot T (represented by Λ^T and Λ_o^T) are duplicated to form Λ^{Tb} and Λ_o^{Tb} in Γ^{Tb}, respectively. As explained in Section 2.4.4.1, the corresponding D values generated are Λ^{Td} and Λ_o^{Td} in Γ^{Td}.

3. **ILP Formulation:**

   An ILP formulation for the Boolean constrains generated by the above circuit transformations is done in the following way.

   ILP formulation for maximal aggressor excitation is performed in the same way explained in the Section 2.4.3.2, for a crosstalk fault site with the aggressor victim pair (a_i, v).

   As presented for ZDIF approach in Section 2.4.4.1, the set of fault effect propagation constraints are generated for output logic cone Γ^T of victim in time-slot T.

An algorithm similar to the one presented in Pseudocode 2.1, is used here to obtain vector pair $\langle I^0, I^1 \rangle$. The only difference in this case is in the Line 4, where circuit transformation for fault effect propagation is performed instead of performing stuck-at fault ATPG. Moreover, only the transformed circuit is taken as input to generate $ILPEquns$ in the function $generate\_ILP\_eqns()$ appearing in the Line 5.

2.5 Results

The results below are obtained by running crosstalk ATPG on all ISCAS 85 benchmark circuits. A post-processing step after RC extraction from a physical layout is ideally required to generate the crosstalk fault list. Crosstalk fault extraction has been addressed in the following previous works [65] [102]. For the purpose of experimentation we use a randomly generated fault list as our main focus in this work is on crosstalk ATPG. We note that, a list of extracted faults can easily just be substituted here. However, a randomly generated fault list suffices for ATPG analysis. Every node in the circuit is selected with equal probability in order to obtain the aggressor and victim nets for a crosstalk fault site. The maximum number of aggressors per victim is limited to ten in keeping with previous observations on Intel circuits [65]. A value selected randomly between 1 and 0 is used for the coupling weight W_i.

In the following experiments *GNU Linear Programming Kit* (GLPK) [5] was used for solving the ILP equations and ATALANTA [68] was used for stuck-at fault ATPG. The GLPK ILP solver was run with a maximum time limit of 1000 seconds. GLPK may not generate a solution or may potentially produce a sub-optimal solution, in case of a timeout. In ZDSF approach, if the pattern generated by stuck-at fault ATPG at the inputs of the right partition causes a conflict in the left partition, a solution is not found. We use '$-$' to mark the cases where no solution is found in the tables reported here. Results are generated for a $WindowSize$ of 6 for the UDSF and UDIF cases.

A Dell PowerEdge 2800 server with 2.8GHz Dual Core Intel Xeon Processor, 2MB L2 cache and 2GB RAM is used as a platform for these experiments.

M is the total number of aggressors in a crosstalk fault site in the following tables. The sum of coupling weights of all the aggressors is given by $\Delta\tau_m$ which represent the worst case crosstalk induced delay that could be generated at the victim net, if all the aggressors simultaneously switch in the desired direction. The number of aggressors which switch in the desired direction and the corresponding crosstalk induced delay are given by a_s and $\Delta\tau$, respectively. Moreover, the percentage of $\Delta\tau_m$ induced at the victim is given by $\%\Delta\tau = (\Delta\tau/\Delta\tau_m) \cdot 100$. Finally the column titled *time* shows the total execution time in seconds for execution of the tool.

Section 2.5.1 presents results for divide and conquer approach. This is followed by the results for integrated ILP formulation in Section 2.5.2. Finally, both results are compared in Section 2.5.3.

2.5.1 Divide and Conquer

The results for divide and conquer approaches explained earlier are presented in this section. The results for ZDSF approach are presented first. the effectiveness of KLFM min-cut algorithm is studied here. Finally, the results for UDSF are presented.

2.5.1.1 ZDSF Approach

The result for ZDSF approach with and without applying KLFM min-cut algorithm are compared in Table 2.2. For both the cases a large $\%\Delta\tau$ value is obtained as ZSDF is able switch almost all the aggressors in the desired direction. Moreover, The effectiveness of min-cut algorithm is apparent, as results are obtained for more number of circuits with consistently smaller execution times.

For example, we are able to switch all the aggressors in the direction opposite to the victim net, for big circuits like c7552 and c5315. Moreover, for a large circuit like c7552, the min-cut approach generates result while the one without min-cut just fails.

Table 2.2. Results for ZDSF approach on ISCAS85 benchmark circuits [42]

Ckt	M	$\Delta\tau_m$	With Min-cut				Without Min-cut			
			a_s	$\Delta\tau$	$\%\Delta\tau$	$time$	a_s	$\Delta\tau$	$\%\Delta\tau$	$time$
c17	4	2.78	3	2.20	79.14	0.175	3	2.20	79.14	1.664
c432	4	2.27	3	1.94	85.46	0.771	3	1.94	85.46	7.21
c499	6	2.90	6	2.90	100.00	0.839	6	2.90	100.00	6.021
c880	3	1.39	3	1.39	100.00	0.911	3	1.39	100.00	7.656
c1355	2	1.19	1	0.78	65.55	1.291	1	0.78	65.55	11.106
c1908	9	4.42	8	4.33	97.96	162.132	8	4.33	97.96	99.267
c2670	5	1.38	-	-	-	2.183	-	-	-	0.841
c3540	5	2.75	-	-	-	0.744	-	-	-	2.13
c5315	9	5.80	9	5.80	100.00	14.382	9	5.80	100.00	95.914
c6288	6	1.90	-	-	-	1011.589	-	-	-	1041.286
c7552	5	3.16	5	3.16	100.00	26.395	-	-	-	3.891

For the circuit c5315, min-cut approach generates the input vector pair in one fifth of time as compared to the approach without min-cut and switches all the aggressors in the opposite direction to the victim net.

2.5.1.2 UDSF Approach

Table 2.3 shows the effect of the $\%\Delta\tau$ obtained for unit delay model for the UDSF approach. The overestimation obtained in the UDSF case is evident from the fact that $\%\Delta\tau$ is much reduced as compared to ZDSF case.

2.5.2 Integrated ILP Formulation With Error Propagation

Results for the integrated ILP formulation explained earlier are presented in this section. The results for ZDIF approach is presented firstly. This is followed by those for UDIF approach.

2.5.2.1 ZDIF Approach

The results for the ZDIF approach, where single ILP formulation was done for maximal aggressor excitation and fault effect propagation considering zero delay

Table 2.3. Results for UDSF approach on ISCAS85 benchmark circuits [42]

Ckt	M	$\Delta\tau_m$	a_s	$\Delta\tau$	$\%\Delta\tau$	$time$
c17	4	2.78	3	1.59	57.19	0.167
c432	4	2.27	3	1.30	57.27	14.317
c499	6	2.90	4	2.21	76.21	3001.403
c880	3	1.39	3	1.29	92.81	8043.739
c1355	2	1.19	0	0.00	0.00	0.265
c1908	9	4.42	2	0.80	18.1	1398.481
c2670	5	1.38	1	0.27	19.57	2078.042
c3540	5	2.75	2	0.97	35.27	16159.618
c5315	9	5.80	5	1.95	33.62	1001.569
c6288	6	1.90	-	-	-	2.667
c7552	5	3.16	-	-	-	4251.22

model are presented in Table 2.4. It can be seen that a maximal amount of total coupling weight is switched by the patterns obtained here.

Table 2.4. Results for ZDIF approach on ISCAS85 benchmark circuits [42]

Ckt	M	$\Delta\tau_m$	a_s	$\Delta\tau$	$\%\Delta\tau$	$time$
c17	4	2.78	2	1.64	58.99	0.328
c432	4	2.27	4	2.27	100.00	1.772
c499	6	2.90	6	2.90	100.00	1.075
c880	3	1.39	3	1.39	100.00	0.988
c1355	2	1.19	2	1.19	100.00	20.399
c1908	9	4.42	8	4.33	97.96	122.645
c2670	5	1.38	5	1.38	100.00	33.715
c3540	5	2.75	4	2.34	85.09	1006.875
c5315	9	5.80	9	5.80	100.00	2.165
c6288	6	1.90	-	-	-	1074.57
c7552	5	3.16	5	3.16	100.00	95.216

2.5.2.2 UDIF Approach

Results for UDIF case, where single ILP formulation is used for maximal aggressor excitation and fault effect propagation, are presented in Table 2.5. It can be seen

that, a large fraction of maximum delay $\Delta\tau_m$ is switched by the ILP formulation. Comparing these results with ZDIF approach shows that this switched weight is still less than that obtained from ZDIF.

Table 2.5. Results for UDIF approach on ISCAS85 benchmark circuits [42]

Ckt	M	$\Delta\tau_m$	a_s	$\Delta\tau$	$\%\Delta\tau$	*time*
c17	4	2.78	3	1.67	60.07	0.131
c432	4	2.27	3	1.30	57.27	6012.964
c499	6	2.90	5	2.79	96.21	5013.876
c880	3	1.39	3	1.29	92.81	16008.455
c1355	2	1.19	-	-	-	1671.375
c1908	9	4.42	7	2.08	47.06	27191.043
c2670	5	1.38	1	0.17	12.32	5217.54
c3540	5	2.75	-	-	-	34389.516
c5315	9	5.80	6	2.72	46.9	1000.995
c6288	6	1.90	-	-	-	12140.493
c7552	5	3.16	-	-	-	3006.121

2.5.3 Comparison

The $\%\Delta\tau$ value for all the above approaches is compared in Fig. 2.11. It can be seen that, as compared to the zero delay approach, the unit delay approach generates smaller delay. This shows the importance of delay model for crosstalk ATPG and the over estimation obtained from the zero delay model. The crosstalk fault list can be pruned by using the above observation by removing the fault sites that do not violate a critical path in the circuit. Moreover, as compared to divide and conquer approach, the integrated ILP formulation consistently does better. The sub-optimality of the divide and conquer approach is evident from the above observation.

It can be seen that no solution is obtained for the circuit C6288, in the above tables. This benchmark circuit is a 32-bit multiplier, which is a known nemesis for many SAT solvers [66].

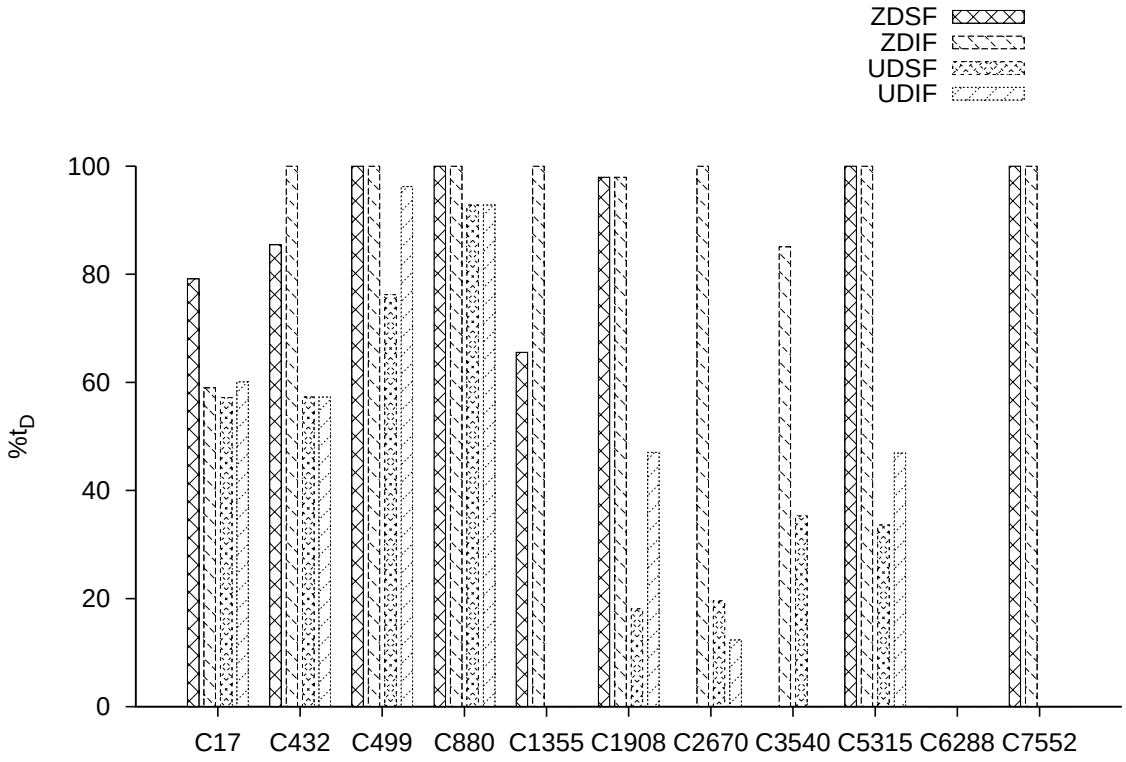


Figure 2.11. Comparison between various approaches for multiple aggressor crosstalk ATPG [42]

2.5.4 Scalability

The proposed solution is highly scalable, as will be seen in this section. Crosstalk ATPG performance, resulting size of the test set and ability to handle non-unit gate delays are some of the factors used to evaluate scalability of the proposed approach.

2.5.4.1 Performance

The number of ILP equations determine the scalability of the solution. The likelihood of finding an exact solution increases with reduction in the number of ILP constraints. The cone of logic needed to formulate justification and propagation conditions for the crosstalk faults as shown in Fig. 2.12, which in turn relates to logic depth, determines the number of equations. Modern designs tend to have a shallow logic depth which is typically 6-8 levels of logic gates [51]. Due to non-linear increase

in gate delay with transistor stack height in CMOS circuits, the number of fanins is limited. It has been seen that [88] the number of fanins in CMOS gates is limited to 4. Hence for a circuit with logic depth l and fanin of f, the number of gates in a logic cone is of the order of $O\left(f^l\right)$. The worst case size of such logic cone is of the order of $O\left(l \cdot f^l\right)$ when unit delay model is considered. The logic depth tends to be much greater in ISCAS circuits. The logic depth for C3540, for which we had the worst case run-time, is 47. Modern circuits tend to have a greater run-time as the logic cone of interest is correspondingly much larger than expected logic cone size. An interesting observation that can be made here is that the run-time actually decreases for circuits such as C7552 where the total gate count was larger but the logic depth was smaller. This is because of the reasons described here.

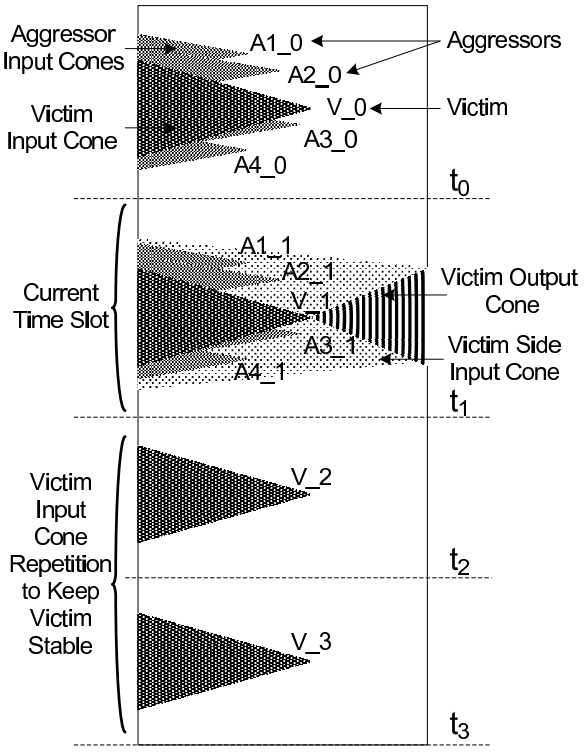


Figure 2.12. Input and output logic cones used for ILP formulation [42]

2.5.4.2 Test Compression

Test compression is another benefit of the proposed approach. The scenario where the bits in a test vector are unspecified works best for test compression. The logic cone of interest, in a multi-million gate circuit, includes only a small fraction of the inputs. The inputs outside the cone of interest remain unspecified as they are not included in the ILP formulation. Thus, the test cubes have all the necessary characteristics for good compression. Moreover, Xs can be unmasked through simulation process, even for the inputs included in the ILP formulation which get fully specified during ILP solution. A procedure to extract Xs for a set of specified inputs is described in [93].

2.5.4.3 Beyond Unit Delay

Unit delay buffers can be used to easily convert an integer delay circuit into an equivalent unit delay circuit. For example, we can insert two buffers between gate output and its fanouts for a NAND gate with an integer delay of 3 units. Please note that the number of equations in our formulation will not increase as a result of this transformation. Similarly we can extend this solution to circuits with real delays that can be normalized to have equivalent integer delays.

2.6 Conclusion

Various ATPG techniques to generate a two pattern test for multiple aggressor crosstalk faults is presented here. This problem involving generation of input pattern pair for maximizing delay at the victim net in the presence of multiple aggressors is a max-satisfiability problem which is known to be intractable. Essentially, in all our solutions presented here, we approach max-satisfiability problem using ILP formulation. The solution to fault propagation problem is either obtained independently through circuit partitioning or by using an integrated ILP formulation. Results show that integrated approach produces better quality solution while partitioning based approach is faster as the number of equations is fewer.

Moreover, the effect of gate delays on the maximal weight switched is also studied here. Circuit transformation is done in order to account for gate delays. This can be extended to arbitrary integer gate delays by adding unit delay buffers. Moreover, floating delays may be scaled and approximated as integer delays without any loss of generality of the solution. A comparison with zero delay case shows that zero delay case results in a gross overestimation of the maximum crosstalk in the circuit. The importance of gate delays in crosstalk ATPG is evident from the above observation. Finally, as we only consider the input and output logic cones for ILP formulation, our approach is shown to be highly scalable for modern circuits which are known to have a small logic depth.

CHAPTER 3

A PATTERN GENERATION TECHNIQUE FOR MAXIMIZING SWITCHING SUPPLY CURRENTS CONSIDERING GATE DELAYS

3.1 Introduction

The design of power and ground rails so as to ensure reliability and performance of VLSI circuits is one of the major challenges in current generation integrated circuit design. A degradation in the switching speed of CMOS circuits, as a result of voltage droop in the power rails, can be attributed to a poorly designed power delivery network [18]. Long term reliability problems are caused by excessive current flow through metal conductors as a result of an increased electromigration rate [18]. The power delivery network may be insufficiently matched to demand without proper analysis. Moreover, excessive current through die-package interface can cause thermal meltdown of solder bumps [33].

Therefore, accurate current estimation through power supply lines in very important. There are two major components to the total power supply current: (i) current due to switching of CMOS gates; and (ii) the CMOS leakage current. CMOS leakage current has gained attention in the literature [58] as it has been steadily rising due to increased sub-threshold and gate oxide leakages. This work presents an *Integer Linear Program* (ILP) based pattern generation technique to create worst case (best case) current draw from power supply rails. By appropriately setting the optimization cost functions, this generic technique can be applied to static, dynamic or composite currents. In this work we limit our discussion to dynamic current only. One of the most important part of the problem specification process is defining the cost function.

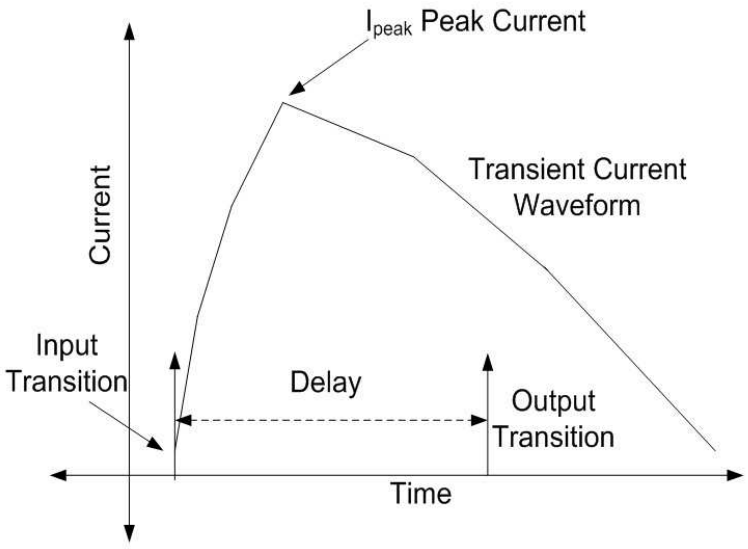


Figure 3.1. Piece wise linear approximation of supply current waveform [57]

In this work, our main focus is solely on the current drawn, during transitions in gate outputs, from the power supply rails. Multiple current conduction paths between supply and ground rails are formed when this current flows through various parasitic capacitances of interconnect lines and output load(s).

For example, for an inverter output switching from $0 \rightarrow 1$, current flows through the PMOS transistor. The intrinsic gate/drain capacitance of the CMOS gate and the input capacitance of fanout gates are charged by this current flowing through this PMOS. Moreover, this current may flow partially through the coupling capacitors to other interconnect lines and eventually to other gates, as these fanout lines are capacitively coupled with other switching and non-switching lines. However, we can use Kirchoffs law to focus on the currents drawn by transistors within a gate, without having to consider coupling currents separately.

The power supply current which flows through the $p-tree$ ($n-tree$) transistors, during $0 \rightarrow 1$ ($1 \rightarrow 0$) transition at gate output, charges the output capacitance. For a combinational circuit with n primary inputs, the possible set of input patterns is defined by using a vector of 6-valued logic given by $l \rightarrow l$ ($low\ to\ low$), $l \rightarrow h$

($low\ to\ high$), $h \rightarrow l$ ($high\ to\ low$), $h \rightarrow l \rightarrow h$ ($high\ to\ low\ to\ high$) and $l \rightarrow h \rightarrow l$ ($low\ to\ high\ to\ low$). The glitches that could possibly occur within a single clock period under the assumption of unit delay model are represented by $h \rightarrow l \rightarrow h$ and $l \rightarrow h \rightarrow l$. For various input patterns, *Transient Current Waveforms* are drawn at the contact points. The transient current waveforms at each contact point can be described by using a *Piece-Wise Linear* (PWL) function with a peak current value of I_{peak} (Fig. 3.1). In order to perform accurate estimation of maximum current waveform at every contact point, we need to determine the set of current waveforms corresponding to all possible input patterns.

For a circuit with n primary inputs, since each input can assume one of the six logic states: $l \rightarrow l$, $h \rightarrow h$, $l \rightarrow h$, $h \rightarrow l$, $l \rightarrow h \rightarrow l$ and $h \rightarrow l \rightarrow h$, we need to simulate for a total number of 6^n input vectors. As the number of simulations which must be performed in order to find the maximum current is *exponential* in the number of inputs to the network, estimating maximum current for a large CMOS logic network is computationally intractable. Thus, peak current maximization can be mapped to constrained max-satisfiabilty which is $NP - Hard$.

In this work, a two pattern test is generated assuming unit delay model for the gates as explained in detail in Section 3.3.3. As a result, there is a set of time-slots associated with every gate representing the time at which it could possibly switch. At a given time-slot, when the biggest subset of gates appearing at that time-slot switch, so as to cause a maximal supply rail current while obeying Boolean relationships, peak current is achieved. The maximum supply rail current observed over all time-slots represents the peak current for the entire circuit. It is important to note that, we improve the quality of the solution further compared to zero delay model by taking glitches into consideration with unit delay assumption.

In this work we build upon the fact that only a fraction of all the gates fall under a specific time window when they could possibly switch, if individual gate delays are

considered. This observation results in a significant pruning the size of individual problem instances improving the accuracy of the solution as well as increasing the chances that an exact solution may become feasible.

The proposed approach is presented in the rest of the chapter as follows: we review previous work in Section 3.2. In Section 3.3 a mathematical formulation of the peak current pattern generation problem is provided. The ILP-based proposed approach is explained in detail in Section 3.4. This is followed by Section 3.5 where we discuss the scalability of the proposed solution. Section 3.6 presents experimental data and analysis on ISCAS-85 and ISCAS-89 benchmark circuits. Finally we present conclusions in Section 3.7.

3.2 Related Work

The problems of estimating maximum current and the peak power dissipation for a CMOS circuit are mathematically identical in nature. Moreover, they have been addressed in literature with significant importance over the last decade. Kriplani *et al.* [49] presents a pattern independent approach for supply rail current estimation. This offers improvement in execution times compared to SPICE-based methods presented in [78] and [54], but is overly pessimistic as Boolean filtering is not used.

An approach for maximum current estimation is presented by Chowdhury *et al.* [28]. In this approach, the above problem is addressed by partitioning the circuit into macro modules and then applying the exact search technique or a suitable heuristic separately on each of them to come up with the solution. However, because of the assumption that all the macros draw their maximum currents simultaneously, their methodology suffers from an over-estimation trend. Ganeshpure *et al.* presents a pattern-dependent peak current estimation approach which employs a branch-and-bound heuristic to incrementally modify the Boolean clause for satisfiability towards

attaining a bounded solution [56]. In this approach Boolean filtering is considered, but it does not take internal gate delays into account.

Jiang *et al.* [111] evaluate a set of different algorithms and their relative performance. They reported that the ILP-with-partitioning approach provides tightest upper bound for small circuits, while for large circuits the lower bound obtained by the GA-based approach seems to be most effective. This upper bound obtained from GA-based approach outperforms the other timed-ATPG and the probability-based approaches. They also observed that for combinational circuits, the timed-ATPG, probability-based and ILP-based approaches are only applicable, while the GA approach applies to sequential circuits as well. Chai *et al.* [19] present an ILP-based algorithm using the signal correlations within a circuit, in a similar context of leakage current minimization. Through relaxing the constraints of the integer program, they claimed a faster solution compared to [90]. The solution resulted in a search space explosion as ILP formulation was done by considering all the possible input combinations for every gate. Moreover, internal gate delays were not taken into account for their approach. In this work, we significantly reduce the complexity of a single instance of the search space by considering unit gate delay model under which only a subset of gates could possibly switch at a given time window. The unit gate delay model is not only more realistic compared to zero gate delay model but also reduces pessimism of the solution.

Devadas *et al.* [90] present a technique for the estimation of worst case power dissipation in CMOS combinational circuits. In their approach the above problem was reduced to a weighted max-satisfiability problem on a set of multi-output Boolean functions. This is followed by using either a *disjoint cover enumeration* algorithm or the *branch-and-bound* algorithm to solve the $NP - Hard$ problem. However, even under the unit gate delay assumption, for a multilevel logic circuit, the functions generated by their algorithm are fairly complex, and suffer from long execution times.

Moreover, this approach does not provide a sub-optimal solution that can be used as upper bounds of switching activity as it requires that the problem be solved optimally. A solution which gives an upper bound of maximum transition or switching density of individual gates for CMOS combinational circuits computed via propagation of uncertainty waveforms is presented by Najm *et al.* [79]. Wang *et al.* [107] present a test generation based approach which tries to find test patterns that would produce the maximum gate switching corresponding to the maximum power dissipation. However, both Najm *et al.* [79] and Wang *et al.* [107] did not take gate delays into account. Gate delays were considered by Manich *et al.* [75] to show improvement in the quality of the solution when non-zero delay model is considered. A circuit transformation-based approach which performs time-domain expansion of a given combinational circuit to incorporate gate delays was developed in this work. The equivalence between switching activity maximization problem and stuck-at fault-testing problem on a transformed circuit was shown by the authors. Here maximum weighted activity is achieved by test vectors covering a selected set of faults on the transformed circuit. We have adhered to this circuit transformation technique to incorporate unit gate delays in this work.

Wu *et al.* [87] present a statistical approach based on the asymptotic theory of extreme order statistics. They applied probabilistic distributions of the cycle-by-cycle power consumption and the maximum likelihood estimation in the context of peak power maximization problem. A peak power estimation tool K2 was proposed by Hsiao *et al.* [74]. It generates a specific vector sequence that produces maximum power dissipation in both combinational and sequential circuits. Gupta *et al.* [48], present a hamming distance-based approach. In this work they estimate energy and peak current for every input vector pair. *Boolean Satisfiability* (SAT) and ILP based solvers were used to conduct experiments on identifying an input pattern pair that maximizes weighted switching activity for a CMOS combinational circuit by Sagahy-

roon *et al.* [92]. They used solvers like PBS 4.0 [38], Galena [20], and MiniSAT+ [37] in addition to the commercial ILP solver CPLEX 7.0 [7] and reported exact solution for most of the small to medium size MCNC benchmark circuits within a reasonable time. However, when gate delays are taken into account, for very large scale commercial designs the SAT-based formulation still may suffer from extremely long execution time. One of our major objectives in this work, is to show that SAT and ILP-based exact solutions may become feasible when internal gate delays of the combinational circuit are considered.

[9] [55] show several ATPG-based approaches. They generate input vectors to either estimate the power supply noise or in the context of delay testing. In these approaches, delay is seen as an effect of variable IR-drop across the power supply rails. A fault model to address the problem of vector generation for delay faults arising out of power delivery problems has been presented in Tirumurti *et al.* [18].

From the above survey, it can be seen that, the previous work for calculation of maximum currents in the power supply rails suffer from limitations such as long execution time or weak upper/lower bounds for maximum current value. We explain the switching current model and the gate delay models assumed in this work, in the following sub-sections. Moreover, we also formally define the problem statement.

3.3 Problem Formulation

In this section we explain the switching current and the gate delay models assumed in this work and formally define the problem statement.

3.3.1 Switching Current Model

In this work we use a real valued weight, known as switching weight, to represent peak current. In order to represent the peak current through the supply lines during

$0 \rightarrow 1$ and $1 \rightarrow 0$ transition, each gate is associated with $0 \rightarrow 1$ and $1 \rightarrow 0$ switching weights $w_P(g)$ and $w_N(g)$, respectively.

For a CMOS logic gate, the output line may charge/discharge through the load capacitance as shown in Fig. 3.2 c. The various effective resistance and capacitances that are a part of the equivalent circuit [110] are shown in part (b). The drain diffusion capacitances given by C_{NMOS} and C_{PMOS} are contributed by the driver gate. The capacitance to ground for the wire connecting the source and the sink constitutes the interconnect capacitance. For the sink inverter, we have the gate capacitances of the $PMOS$ and $NMOS$ transistors represented as Cg_{PMOS} and Cg_{NMOS} respectively. The output load capacitance C_{GND} used for transient analysis, is equal to the sum of all the above capacitive components. Moreover, a transistor may be represented as a switch, in a first-order simplified model. According to this model the transistor has an ON effective resistance of R_{PMOS} and R_{NMOS} for the $PMOS$ and $NMOS$ transistors respectively.

The output capacitance, as shown in part (c) of the Fig. 3.2 is discharged (charged) through R_{NMOS} (R_{PMOS}) during a $1 \rightarrow 0$ ($0 \rightarrow 1$) transition at the output net of the driver through the loop current $I(t)_{1\rightarrow 0}$ ($I(t)_{0\rightarrow 1}$).

Let V_{DD} be the supply voltage (assuming $V_{SS} = 0$) and $V(t)$ be the voltage at the output node of the first inverter. Consequently, the output current $I(t)_{0\rightarrow 1}$ for $0 \rightarrow 1$ switch is obtained from the following equation:

$$V(t) = V_{DD} \cdot \left(1 - e^{\left(-\frac{t}{R_{PMOS} \cdot C_{GND}}\right)}\right) \tag{3.1}$$

$$I(t)_{0\rightarrow 1} = \frac{(V_{DD} - V(t))}{R_{PMOS}} = \frac{V_{DD}}{R_{PMOS}} \cdot e^{\frac{-t}{R_{PMOS} \cdot C_{GND}}} \tag{3.2}$$

Therefore, for a gate g corresponding to the maximum values of the switching current I_{peak}, the switching weights $w_P(g)$ and $w_N(g)$ are obtained using the following set of equations.

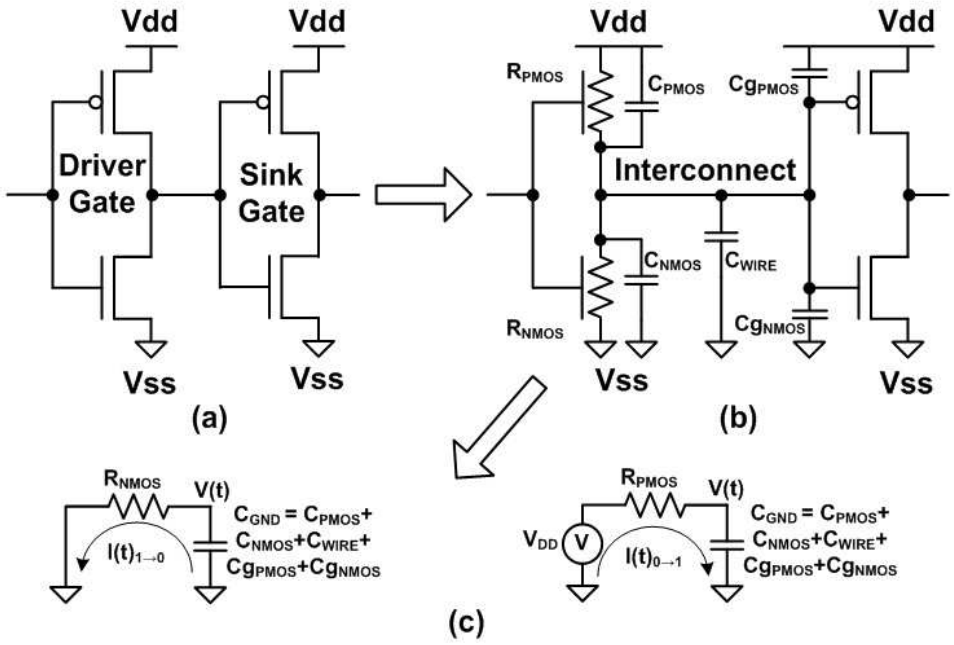


Figure 3.2. Switching current model for an inverter driving another inverter. (a) a driver inverter driving a sink inverter (b) equivalent RC model for the driver and the sink (c) equivalent RC circuit for $0 \rightarrow 1$ and $1 \rightarrow 0$ transition at the driver output. [57]

$$w_P(g) = I_{peak}(0 \rightarrow 1) = \frac{V_{DD}}{R_{PMOS}} \tag{3.3}$$

$$w_N(g) = I_{peak}(1 \rightarrow 0) = \frac{V_{DD}}{R_{NMOS}} \tag{3.4}$$

Please note that we use a simplified first order model for the above equations. These weights, in proposed calculation, may be obtained from simulation.

3.3.2 Peak Current Weight Extraction for Logic Gates

The peak current for gates present in the cell library is computed by SPICE simulation [6]. The goal of peak current estimation is to switch a given gate by applying pattern pairs in such a way that it causes worst case power supply rail current during $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions. A maximum number of parallel conduction between V_{DD} and V_{SS} for either rising or falling output transition are switched due to the application of these pattern pairs. In order to draw maximal current from

the power supply lines, these pattern pairs should ideally switch all transistors in the $n-tree$ for peak $1 \rightarrow 0$ transition current (input pattern pair transitioning from all ones to all zeros) and all transistors in the $p-tree$ for peak $0 \rightarrow 1$ transition current (input pattern pair transitioning from all zeroes to all ones). In order to measure the peak value of the current waveform, a current meter is placed on the power supply rails. The above process is repeated for a range of output loads. These loads are set to an integral multiple of minimum sized inverter load in the same technology. Finally, the measured peak currents are normalized with respect to the minimum value among the cells in the library. These weights are stored in a look-up table and a combination of gate type and the number of fan-outs are used to access them. Multiple simulations are necessary to determine the worst case weight if it turns out that it would be impossible to switch on all transistors of a given type due to input signal dependencies.

3.3.3 Unit Gate Delay Model

Unit delay assumptions are used for the above pattern generation technique for logic-level circuits. At the transistor level, pattern generation remains an elusive goal. In Section 3.5.2 we have shown that with introduction of unit delay buffers, the unit delay model can closely approximate the behavior of a circuit due to exact delays. The need to consider circuit delays is explained in the following example.

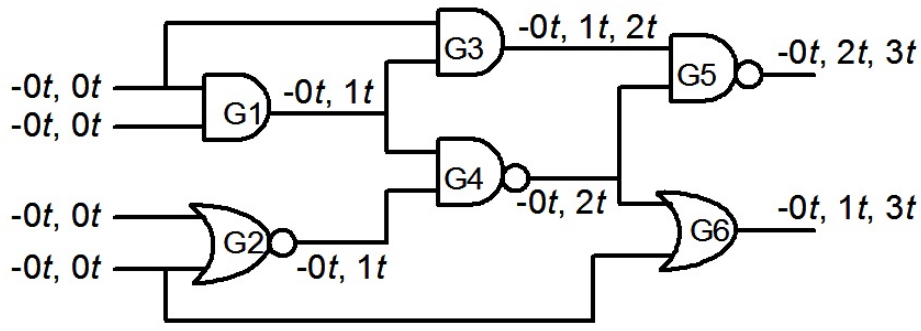


Figure 3.3. An example circuit C with transition times for unit delay model [57]

Example 3.3.1. Fig. 3.3 shows a combinational circuit C where all the gates are assumed to have unit switching delay. The anterior vector I^0 is the first input vector of the pattern pair which was applied to the inputs prior to time $0t$. At time $0t$, the input changes to the present vector I^1. The lengths of all the paths arriving at the node output obtained using the signal propagation times are shown above the signal lines in Fig. 3.3. The propagation of input transition may produce two transitions on gates $G3$, $G5$ and $G6$; and one transitions in gates $G1$, $G2$ and $G4$. The time at which the gate output possibly transitions and attains a stable logic value is represented by the time values at the gate outputs. These time values are referred to as *Time-Slots*. Hence, only the gates $G1$, $G2$, $G3$ and $G6$ may switch at the time-slot $1t$. Similarly, at time-slot $2t$ the gates $G3$, $G4$ and $G5$ may switch and so on. Hence, we observe that at the same instant of time not all gates may switch. Consequently, for each time-slot, we may create a list of gates that may become active. From the above example we can see that, in order to obtain a transition at a gate output we require a pattern pair. The first pattern in this pair stabilizes a specific logic value (0 or 1) at the gate output while the second pattern defines a transition from one stable logic value to another (possibly, even same). Our objective, in this work, is to find a pattern pair that causes worst case peak current. Once the pattern pair is found, HSPICE simulation can be done in order to compute the actual peak current.

From the previous work shown in [56], it can be seen that, if the two patterns of the pattern pair are not generated concurrently, then the final solution can be sub-optimal. This pattern pair problem is best solved with two copies of the same circuit merged together (Fig. 3.4) to account for the correlation between two patterns to manifest their combined effect in the final switched weight. ■

A gate output may experience one or more transitions for a given pattern pair as described in the previous example, when we take internal gate delays into account. Consequently, we may consider duplicating the circuit for n times corresponding to n

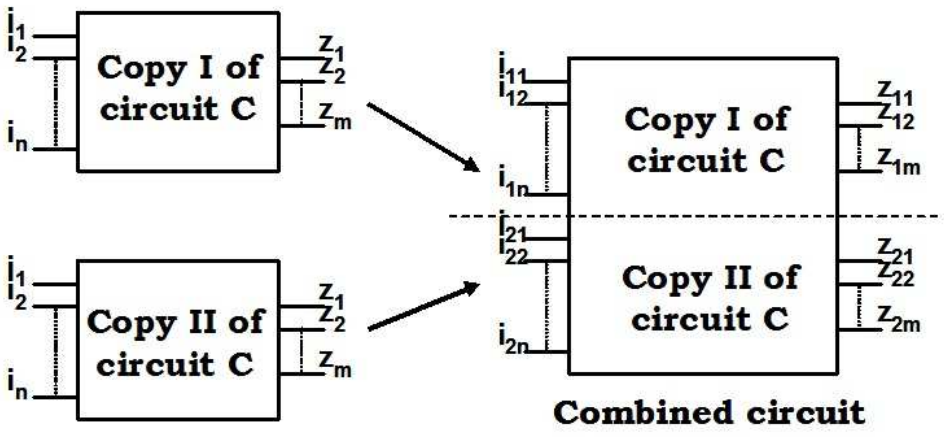


Figure 3.4. Block diagram showing two identical copies of a circuit C placed side by side to form a combined circuit [57]

time-slots. Hence, the circuit size can potentially increase n fold, where n is delay of the longest path under unit gate delay model. From the above example, it is observed that only a subset of all the gates may switch at a given time-slot. Hence, we don't need to consider the set of all gates switching at an individual time-slot. Instead, we may create a copy of the circuit with only the gates that may possibly switch for each individual time-slot.

Based on this observation, a circuit transformation is done that converts the original circuit to an expanded circuit where individual gate are copied for every single time-slot whenever that particular gate may become active. This circuit transformation has been used in [75] to take delay model into consideration for maximizing the weighted switching activity in combinational circuits. If n represents the total number of possible propagation times in the original circuit, the transformed circuit will have n levels. Section 3.4.1, explains this circuit transformation step in further detail. This circuit transformation is similar to the one presented in Sections 2.4.3.2 to consider the effect of delays in crosstalk ATPG. It can be seen here that, this transformation step allows time to be modeled as space such that Boolean analysis alone on the transformed circuit will suffice.

3.3.4 Problem Statement

We now focus on formally defining the problem statement with the above discussion on switching current and gate delay models assumed in this work.

Consider a combinational circuit C. The circuit operates over N time-slots, under the assumption of unit delay model. Here we assume that for each gate g, there is an associated pull-up/pull-down weight pair $\langle w_P(g), w_N(g)\rangle$. Here the pull-up weight $w_P(g)$ and pull-down weight $w_N(g)$ represents the cost associated with the current drawn from the supply rails when the output of the gate g transitions from $0 \rightarrow 1$ and $1 \rightarrow 0$ respectively.

We need to maximize the total pull-up/pull-down weight of all the gates at the same time-slot, in order to have maximum switching activity. Therefore, at the given time-slot, the total switching weight is given by the following equation.

$$W\left(t_j\right) = \sum_{g \in G(t_j)} \left(w_P(g) \cdot U\left(g, t_j\right) + w_N(g) \cdot D\left(g, t_j\right)\right) \tag{3.5}$$

In the above equation, the set of gates which can switch in the given time-slot t_j is represented by the set $G\left(t_j\right)$. Now, when a gate g transitions from $0 \rightarrow 1$ $(1 \rightarrow 0)$ and attains a value of 1 (0) in the time-slot t_j, then the Boolean variable $U\left(g, t_j\right)$ $\left(D\left(g, t_j\right)\right)$ is set to $TRUE$.

In this work, we need to find the pattern pair $\langle I^0, I^1\rangle$ that causes maximal switching activity for the current time-slot.

$$\langle I^0, I^1\rangle = pattern\ pair\left\{\max_{t_j \in N}\left(W\left(t_j\right)\right)\right\} \tag{3.6}$$

For example, the total weight $W(2t)$ corresponding to the time-slot $2t$, as shown in Fig. 3.3 is found as the sum of the weights switched for the gates $G3$, $G4$ and $G5$, which are targeted to switch between consecutive time-slots $\{1t, 2t\}$, $\{0t, 2t\}$ and $\{0t, 2t\}$, respectively. While keeping a track of the weight switched and the

corresponding pattern pair obtained, this process is repeated for all the above time-slots. The time-slot for which we obtain the maximum weight, gives the worst case vector pair.

3.4 The Proposed Approach

As peak current estimation is an $NP - Hard$ problem, our objective is to search for a nearly optimal solution and record the corresponding vector pair $\langle I^0, I^1 \rangle$. The proposed approach consists of the following two basic steps.

- *Circuit Transformation:* Here a time-space transformation of the original circuit C is done. As a result of this, time sequence of two vectors, $\langle I^0, I^1 \rangle$ is translated into the expanded circuit Z representing the switching of various gates. This is followed by adding a set of 2-input XOR gates to each of the internal nodes of the circuit in order to represent switching. These outputs of the XOR gates represent the outputs of the expanded circuit.

- *ILP Formulation:* In this step and ILP based formulation is done to generate a set of objective functions corresponding to each time-slot of the expanded circuit so as to search for an exact solution to the maximization problem expressed in the above Equation 3.6.

The above two steps are elaborated in the following sub-sections. This is followed by a detailed description of the ILP-based peak current estimation algorithm ($I - PEAK$) in the Section 3.4.2.

3.4.1 Circuit Transformation for Gate Delays

Manich *et al.* [75] have shown previously in the context of peak current estimation that, choice of gate delay model plays an important role in peak current determination. Under the assumption of non-zero gate delay model, the problem becomes more

realistic, because only then the temporal proximity of different gates can be considered. A set of gates will not effect the power/ground rail at a given time instant if they do not switch within a finite time window with respect to each other. Consequently, they should not be considered together to compute the peak current. This timing filtering cannot be applied under zero gate delay model, resulting in an overly-pessimistic measure for peak current. We assume unit gate delay model in this work in order to address the issue of timing filtering. According to the unit delay model, we assume that it takes 1 unit of time between 50% transition of the input to the 50% transition of the output for any given gate. Hence, we do not consider the transport delay to propagate an output transition of a gate to the input of its fan-out.

In order to take gate delays into account, time domain expansion is used to translate a circuit structure under unit delay model to an equivalent expanded circuit without any delays. As a result of circuit transformation, there is a one-to-one correspondence between the gate replications in the expanded circuit and the corresponding transitions in the original circuit. The signal transitions occurring at various time-slots are represented by using XOR outputs in the expanded circuit. These XOR gates are connected to the gate outputs appearing in a pair of consecutive time-slots in the expanded circuit [75]. This circuit transformation step has been represented by the sub-routine given by $circuit\_xform()$ in the Pseudocode description of the I-PEAK algorithm, described in Section 3.4.2. This circuit transformation is also used in context of generating a pattern pair maximizing total crosstalk induced delay at the victim net in multiple aggressor crosstalk scenario [42]. A detailed explanation of the above procedure is presented in Section 2.4.3.2 appearing in Chapter 2. In the following example we explaining the above time domain expansion as it is used in our problem.

Example 3.4.1. The original circuit given in Fig. 3.3 is transformed after time domain expansion to the circuit in Fig. 3.5, under unit delay assumption, where each

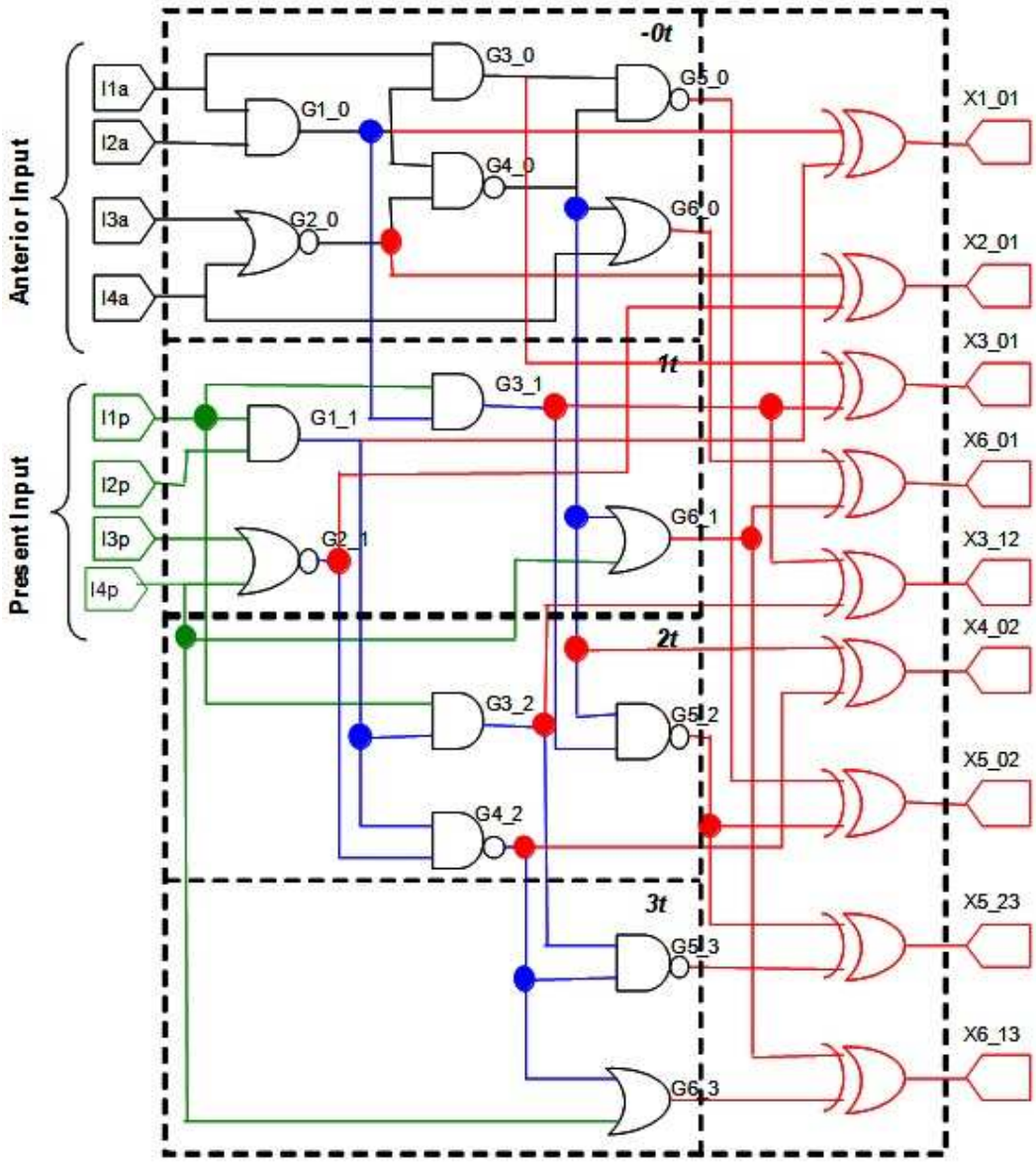


Figure 3.5. The expanded circuit (from Fig. 3.3) after performing circuit transformation under unit delay model [57]

gate is assumed to have a unit propagation delay. Here we avoid the interconnect delay between the output of a gate to the next stage input. The anterior vector I^0 is applied to the inputs prior to the time $0t$. These anterior inputs which are applied at the time-slot $-0t$ for the circuit Z are shown by $\{I1a, I2a, I3a, I4a\}$, as shown in the Fig. 3.5, while $\{I1p, I2p, I3p, I4p\}$ represent the present inputs to which the

present vector is applied after time $0t$. The number of replicas of a gate in this expanded circuit is equal to the possible propagation times listed against individual gates in the original circuit (Fig. 3.3). The circuit Z in our case covers time-slots $\{-0t, 1t, 2t, 3t\}$. At each instant of the expansion every gate has a replica in which it shows activity. For example, the gate $G3$ has three replicas given by $G3\_0$, $G3\_1$ and $G3\_2$ at various expansion instances $-0t$, $1t$ and $2t$ corresponding to the set of propagation times $\{-0t, 1t, 2t\}$ respectively. For the gate $G3$, its two inputs are connected to the primary input $I1$ and the output of gate $G1$ in the original circuit C. Consequently, in the expanded circuit Z, the input of $G3$, originally connected to the primary input $I1$, is connected to the primary input $I1a$ for the replica $G3\_0$, and to $I1p$ for the replicas $G3\_1$ and $G3\_2$ appearing at the expansion instants $1t$ and $2t$ respectively. Similarly, the replica of the fan-in gate $G1$ at the previous time-slots supply the other input for the gate $G3$. In the figure the color blue is used to show the connections from fan-in gates of previous time-slots while green is used to show that originating from present inputs.

The auxiliary XOR gates are used to compare the output signals of each gate for any two consecutive replicas in order to detect transitions made by various internal nodes. For the gate $G3$ a pair of XOR gates are assigned between $G3\_0$ and $G3\_1$, and between $G3\_1$ and $G3\_2$, in order to represent transiting at time-slots $1t$ and $2t$ respectively. In the figure the gates and their input connections are shown in red. A transition at the gate output between time-slots $0t$ and $2t$ is represented by the XOR output $X5\_02$. For the time-slot $1t$, the transitions are represented by the gate outputs given by $X1\_01$, $X2\_01$, $X3\_01$ and $X6\_01$. Similarly, for time-slots $2t$ and $3t$, the above transitions are represented by $X3\_12$, $X4\_02$ and $X5\_02$, and $X5\_23$ and $X6\_23$, respectively. ■

It should be noted that, by adding unit delay buffers to the original circuit we can generalize time domain expansion for arbitrary integer delays. Consequently, any

floating point delay can be scaled and approximated as integer delays without any loss of generality of the solution.

3.4.2 ILP Formulation

ILP formulation is done for the circuit by writing ILP equations for the logic gates [41], in order to obtain the maximal switching activity for a given circuit. This is done by using the clausal description of the function of the gates as developed by Larrabee [66]. The Section 2.4.2 provides the description for how these ILP constraints can be generated for a combinational circuit. This has been used in the context of maximal aggressor crosstalk ATPG [42].

For switching current maximization problem, the ILP formulation consists of the following parts:

- ILP formulation for circuit Boolean constraints.
- Constraints for switching occurring at the output of a given gate at a given time-slot t_i.
  - Switching representing a $0 \rightarrow 1$ transition.
  - Switching representing a $1 \rightarrow 0$ transition.

Finally, the objective function is expressed in terms of the variables used as part of the ILP constraints.

Fig. 3.5 shows the transformed combinational Z circuit used here to explain the ILP formulation. The circuit operates on a set of N time-slots under the assumption of unit delay model. The list of time-slots in which a gate g appears is given by the set $N_g = \{t_0, t_1, t_2, ..., t_n\}$. Here the initial time-slot corresponding to the time $0t$ is represented by t_0. As explained earlier, each gate has an associated pull-up/pull-down weight pair $\langle w_P(g), w_N(g) \rangle$. The pull-up and pull-down weights $w_P(g)$ and

$w_N(g)$ represent the switching supply current drawn when the output of the gate g transitions from $0 \rightarrow 1$ and $1 \rightarrow 0$ respectively.

The Boolean logic value at the output of gate g in the time-slot t_i is represented by $\alpha(g, t_i)$. In order to indicate the condition that the gate g has different logic values in the time-slots t_i and t_j, we introduce the Boolean variable $Xg\_t_it_j$. The above variable correspond to the XOR output gates in the transformed circuit Z. Hence, $Xg\_t_{i-1}t_i$ can be calculated for a gate g switching at the time-slot t_i using the following equation.

$$Xg\_t_{i-1}t_i = \alpha(g, t_i) \oplus \alpha(g, t_{i-1}) \tag{3.7}$$

Now, a weight of $w_P(g)$ is switched when the gate output g switches from $0 \rightarrow 1$. In order for the above condition to be $TRUE$, we need to switch the gate g at time-slot t_i ($Xg\_t_{i-1}t_i = TRUE$) and it should have a $TRUE$ value in the time-slot t_i ($\alpha(g, t_i) = TRUE$) after switching. We define a Boolean variable $U(g, t_i)$ to indicate the above condition. Hence, we obtain the following equation.

$$U(g, t_i) = (\alpha(g, t_i) \wedge Xg\_t_{i-1}t_i) \tag{3.8}$$

Similarly, we define a variable $D(g, t_i)$ in order to indicate the $1 \rightarrow 0$ switching at the gate g.

$$D(g, t_i) = \left(\overline{\alpha(g, t_i)} \wedge Xg\_t_{i-1}t_i\right) \tag{3.9}$$

Hence, we define the objective function $O(t_i)$ which maximizes the switching activity at t_i using the following equation.

$$O(t_i) = \sum_{g \in G(t_i)} \left(w_P(g, t_i) \cdot U(g, t_i) + w_N(g, t_i) \cdot D(g, t_i)\right) \tag{3.10}$$

where: $G(t_i)$ represents the set of gates that could possibly switch in time-slot t_i.

This formulation is solved for all the time-slots $t_i : \forall t_i \in [1, N]$ one at a time. The final vector pair $\langle I^0, I^1 \rangle$ is the one which produces the maximum switching current among all the time-slots.

The Pseudocode 3.1 description of the I_PEAK algorithm is presented in the following section. The subroutine $generate\_ILP\_eqns()$ is used for the formulation of ILP constraints.

Algorithm 3.1 $I\_PEAK(C)$ [57]

```
1: {Z, N} ← circuit_xform(C)
2: maxWeightSwitched = 0
3: for all t_i ∈ [1, N] do
4:    ILPEqns ← generate_ILP_eqns(Z, t_i)
5:    {⟨I^0, I^1⟩, weightSwitched, T_s} ← solve_ILP(ILPEqns, T_L)
6:    if T_s < T_L then
7:       if maxWeightSwitched < weightSwitched then
8:          maxWeightSwitched = weightSwitched
9:          ⟨I^0, I^1⟩_max ← ⟨I^0, I^1⟩
10:      end if
11:   end if
12: end for
13: return ⟨I^0, I^1⟩_max
```

The I_PEAK algorithm shown in the above Pseudocode, takes the original circuit C as an input and returns the pattern pair $\langle I^0, I^1 \rangle$ which maximizes the total switching activity at a particular time-slot. Circuit transformation is done in Line 1 to incorporate the effect of gate delays using the function $circuit\_xform()$ using the original circuit C. This generates the transformed circuit Z which goes over a set of N time-slots. This is followed by setting the variable $maxWeightSwitched$ to 0 in Line 2. This variable is used to keep track of the maximum weight switched among all the time-slots.

Now we iterate through all time-slots in the Lines 3-10. In Line 4 the ILP equations for the transformed circuit Z at the current time-slot t_i is generated. This is followed by the solution of the above ILP formulation in Line 5 to generate an input pattern pair $\langle I^0, I^1 \rangle$ and the corresponding $weightSwitched$. It also give the duration T_s for

which the ILP formulation was run. *Gnu Linear Programming Kit* (GLPK) [5] is used for the solution of this ILP formulation for the time limit T_L. Based on this time limit there are two outcomes of the $solve\_ILP()$ function:

- *Success:* when an optimal solution is found by the solver within the specified time limit T_L. In this case the ILP solver returns with a pattern pair $\langle I^0, I^1 \rangle$ and the corresponding weight $weightSwitched$.

- *Timeout:* when the solver fails to find an optimal solution within the specified time limit T_L. As a result, the ILP solver may or may not return with a valid pattern pair. In case it does return with a valid solution, the returned pattern pair corresponds to a sub-optimal solution. We neglect this sub-optimal solution by checking for the timeout condition.

The above mentioned timeout condition is checked in Line 6 where we compare the returned time T_s taken by the ILP solver with the time out limit T_L. In case of a time out $T_s \geq T_L$. When a success is obtained, we keep a track of the pattern pair causing worst case switching in the Lines 7 to 10. This is done by comparing the returned $weightSwitched$ with the $maxWeightSwitched$ in Line 7 followed by updating the $maxWeightSwitched$ to $weightSwitched$ and finally storing the corresponding vector pair into $\langle I^0, I^1 \rangle_{max}$. The above process stops at the Line 12 where we have gone through for all the time-slots $t_i \in [1, N]$. Finally the pattern pair $\langle I^0, I^1 \rangle_{max}$ is returned in Line 13.

The following example explains the above I_PEAK algorithm in more detail.

Example 3.4.2. The transformed circuit Z, corresponding to the original circuit C, shown in Fig. 3.5, is expanded over a set of 3 time-slots. First the ILP solution for the gate $G5$ is presented here, which is later generalized for any gate in the circuit. Now, the set of time-slot for the gate $G5$ is given by $N_{G5} = \{0t, 2t, 3t\}$. Then we do

ILP formulation for each of these time-slots available for the circuit. Next, for the time-slot $2t$, we will explain the generation of ILP formulation.

In order for gate $G5$ to switch between the current time-slot $2t$ (for $i = 2$) and the previous time-slot $0t$(for $i = 0$), we get the following set of equations.

$$X5\_02 = \alpha\left(G5, 2t\right) \oplus \alpha\left(G5, 0t\right) \tag{3.11}$$

Now a $0 \rightarrow 1$ transition at the gate $G5$ is given by the following equation.

$$U\left(G5, 2t\right) = \left(\alpha\left(G5, 2t\right) \wedge X5\_02\right) \tag{3.12}$$

Similarly, for a $1 \rightarrow 0$ transition at the gate $G5$ we get the following equation.

$$D\left(G5, 2t\right) = \left(\overline{\alpha\left(G5, 2t\right)} \wedge X5\_02\right) \tag{3.13}$$

Now, as shown in the Fig. 3.5 for the current time-slot $2t$, the set of gates that appear is given by $G(2t) = \{G3, G4, G5\}$. The objective function $O(2t)$ is obtained using the following equation.

$$\begin{aligned} O(2t) \;=\; & w_P(G5) \cdot U(G5, 2t) + w_N(G5) \cdot D(G5, 2t) + \\ & w_P(G4) \cdot U(G4, 2t) + w_N(G4) \cdot D(G4, 2t) + \\ & w_P(G3) \cdot U(G3, 2t) + w_N(G3) \cdot D(G3, 2t) \end{aligned} \tag{3.14}$$

Similarly, the objective functions $O(1t)$ and $O(3t)$ are generated by doing the above ILP formulation for all the other time-slots at which the gate $G5$ appears in Z. Finally, the maximum switching weight value and the corresponding pattern pair obtained among all the time-slots is returned. ■

3.5 Scalability

The proposed solution is highly scalable. The scalability of the approach is presented in terms of performance of the ILP-based algorithm and its ability to handle non-unit gate delays.

3.5.1 Performance

The number of ILP equations primarily determine the scalability of the solution. The likelihood of finding an exact solution within a specified time limit is higher for smaller number of equations. The number of gates in a particular time-slot and their input logic cones determine the number of equations in the ILP formulation (Fig. 3.6). As shown in Table 3.1, this usually constitutes a small fraction of the total size of the combinational circuit.

Now, for a gate in time-slot t in a circuit with average fan-in of f, the number of gates in the logic cone of this gate is of the order of $O(f^t)$. This worst case gate count can be of $O(n \cdot f^t)$, if n is the total number of gates appearing in time-slot t.

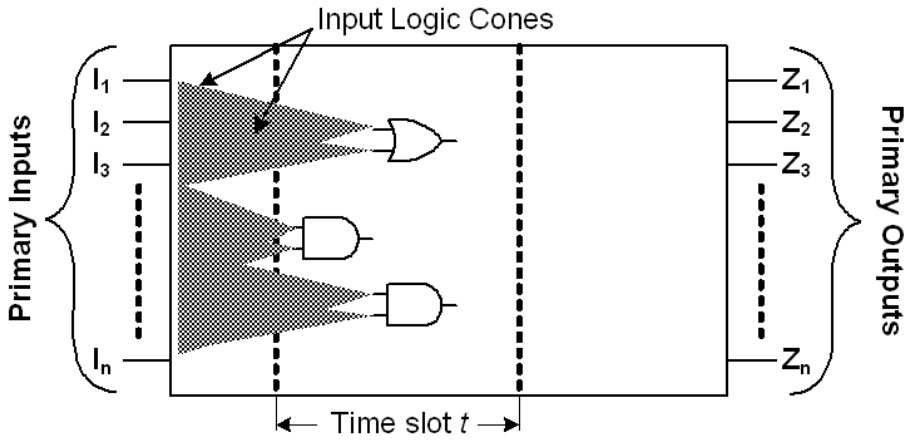


Figure 3.6. Boolean justification clauses generated only for the input logic cones of the active gates in the time-slot t [57]

For most of the benchmark circuits, due to switching of gates in the first few levels, peak current occurs at the initial time-slots. Moreover, the logic depth in modern

designs tend to be typically 6-8 levels of logic gates as seen in [51]. This logic depth for most of the circuits is also very small.

Moreover, as the number of gates in the initial time-slots is a small fraction of the total number of gates in the circuit, n is small. Moreover, the number of fan-ins in CMOS circuits is limited, due to non-linear increase in gate delay with transistor stack height. Typically, this fan-in count in CMOS gates is limited to 4 [51]. This tends to limit the size of the circuit that resides in the input cone of gates in a time-slot. As a result, the total number of equations generated is a small fraction of the total circuit size.

Table 3.1. Maximum number of gates used in an instance of ILP reported as a fraction of total number of gates for ISCAS-85 benchmarks [57]

Circuit	Total Gate Count	Maximum Gate Count Per Instance	Maximum fraction of total number of gates used per instance (%)
c432	160	81	50.6
c499	202	72	35.6
c880	383	148	38.6
c1355	546	168	30.8
c1908	880	154	17.5
c2670	1193	280	23.5
c3540	1669	289	17.3
c5315	2307	496	21.4
c6288	2416	256	10.6
c7552	3512	291	8.29

It has been observed here that, incorporating unit delay model into the problem specification significantly reduces the search space for individual instances of the problem. This is because the entire set of gates for a given circuit gets partitioned onto smaller sub-sets, which are listed as the active gates for individual time-slots. A comparison between the total number of gates for different ISCAS-85 benchmark circuits and the maximal number of gates considered for a single instance of the ILP formulation over all such time-slots, is shown in Table 3.1. It is observed that,

the fraction of gates that are considered for a single instance of the problem becomes lower (Table 3.1) with the growth of circuit size. This makes the peak current pattern generation problem particularly suitable for ILP-based exact solution approach.

Consequently, the unit-delay model assumption staggers the set of active gates over individual time-slots in such a way that individual instances of the peak current estimation problem tends to converge within a reasonable time.

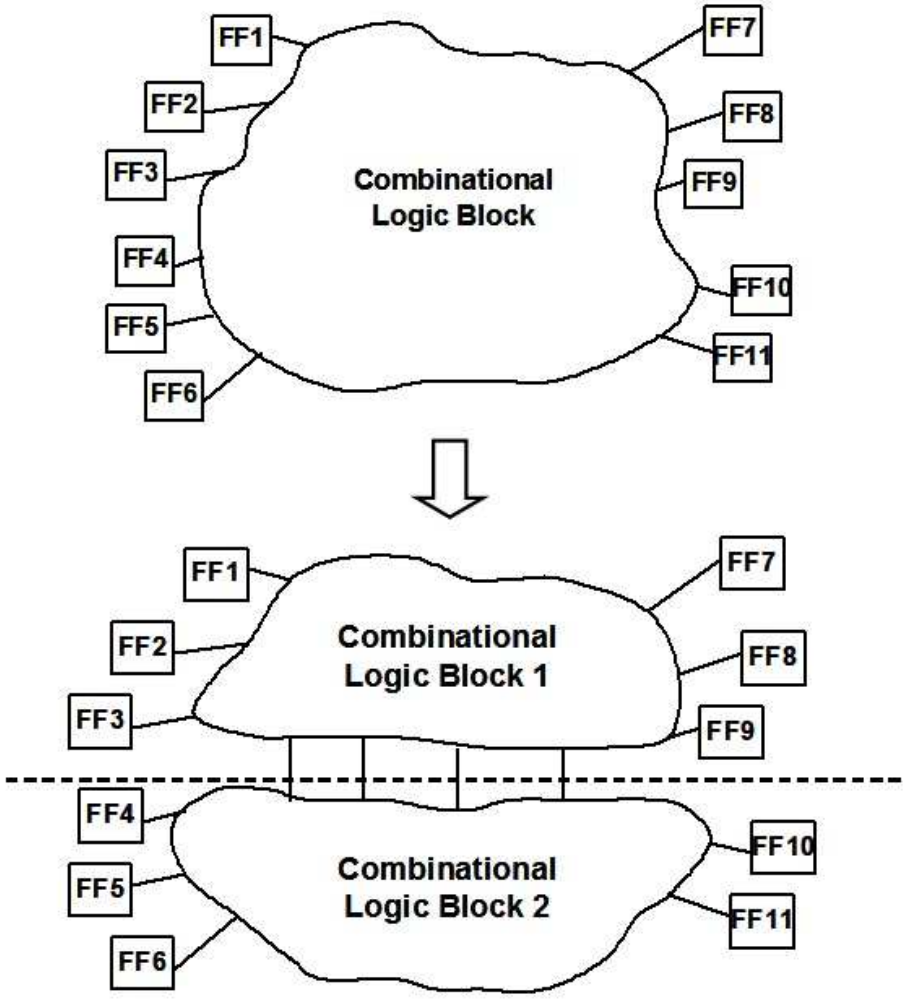


Figure 3.7. Bi-partitioning of a large combinational logic block [57]

The ILP formulation for multi-million gate designs may still run into scalability issues. However as shown in the Fig. 3.7, such cases may be solved by partitioning logic between successive flip-flops stages into horizontal bands. Here, each of the

band is solved separately using the proposed approach. Moreover, it is imperative to minimize the number of signals crossing from one band to another. A min-cut partitioning based heuristic can be used here. In both the partitions, the cut points are assumed fully controllable. An overall solution is generated by aggregating the current waveforms from individual bands. Even though the solution in this case will not be exact, but still significantly better than the state-of-the-art solutions. It can be seen from Table 3.2 that the absolute maximum weight before and after partitioning differs. This is because of time-frame expansion, as the number of time-slots in which a gate may appear in the individual sub-circuits varies based on the cut-points. This exercise shows that (i) the solution is feasible, (ii) partitioning produces an over-estimation of the maximum switched weight and hence is typically pessimistic, and that (iii) after circuit partitioning pattern generation takes significantly less time than running on the full circuit. Consequently, this partitioning based approach can be used as a quick and approximate solution to solve an otherwise difficult problem.

Table 3.2. Comparison of switched weight for c432 benchmark with and without bi-partitioning [57]

Parameter	Without bi-partitioning	After bi-partitioning c432		
		Partition I	Partition II	Combined
Abs. Max. Weight	58.41	42.55	23.48	65.07
Max. Swt. Weight	40.73	31.27	18.13	49.41
Time	65094	15875		

3.5.2 Beyond Unit Delay

Unit delay buffer insertion can be done in order to convert integer delay circuit into an equivalent unit delay circuit. For example, if a NOR gate has a delay of 3, we can insert two buffers between NOR gate output and its fan-outs. Consequently, we can extend this solution for real delays by normalizing real delays using integer delays and applying the above solution.

3.6 Results

In this section, we present the results for the proposed ILP based approach followed by those for the circuit partitioning based approach. Finally, we validate the proposed solution with HSPICE and also provide comparison of zero and unit delay models.

3.6.1 Experimental Setup

Experiments are performed on ISCAS-85 and ISCAS-89 benchmark circuits. As ISCAS-89 benchmarks are sequential, they are used after removing all the sequential elements and converting them to combinational benchmark circuits. For our purpose, the peak current weights for individual gates were randomly generated with the values in the range from 0 to 1. The open source ILP solver GLPK [5] was used to solve the ILP formulation. The solver was run with a time out limit of 1000 seconds set for each of its invocations. A Dell PowerEdge 2800 server [3] with 2.8GHz dual core IntelTM Xeon processors, 2MB L2 cache and 2GB RAM was used as a platform for these experiments.

The experimental results for ISCAS-85 and ISCAS-89 benchmark circuits have been presented and analyzed in the following two sub-sections. This is followed by the validation of the proposed approach with respect to an exact solution obtained through an accurate circuit-level field solver, such as HSPICE [6].

3.6.2 Results on ISCAS-85 and ISCAS-89 Benchmarks

The peak current data generated from the proposed ILP-based approach for ISCAS-85 and ISCAS-89 combinational benchmark suite is presented in Table 3.3 and Table 3.4. The number of gates in the circuit are shown in Column 3. The total number of time-slots for which a given circuit exhibits switching activity when unit gate delay model is considered is shown in the Column 4 of the table. It can be clearly seen that, not each and every gate in the circuit is active for every individual time instant. Every single gate will switch at the same instant giving rise to a significantly

pessimistic measure for peak current for a given circuit, if we do not take unit gate delay model into account. The Boolean relationship between individual gates is not considered in case of static measure of the peak current. Column 6 reports this measure for the circuit. Under the assumption of unit gate delay model, in Column 7 we report the maximum peak current that could be induced for a given benchmark circuit. Boolean relationship among individual gates are also taken into account in order to generated these values, in our proposed approach. The corresponding time-slot for which this maximum peak current is obtained is shown in Column 5. We observe that, a solution for all ISCAS-85 and ISCAS-89 benchmark circuits is obtained by our peak current pattern generation tool. In generating these solutions we use a time out limit of 1000 seconds for each invocation of the ILP solver. This exact measure of peak current is reported as a fraction of the static maximum measure in Column 8. It also show the increasing deviation of the static measure for peak current compared to the exact solution with the growth of circuit size.

We make a second interesting observation by reporting, for individual circuits, the time-slot which experiences the peak current, as shown in Column 5. It is observed that, for all the circuits, this time instant happens to be within the first few time-slots. As a matter of fact, this is within the first two time-slots. The reason behind this is that, the gates in the early time-slots have very high controllabiliy [17]. This is because most of the fan-ins of these gates are either directly connected to primary inputs, or have at most one gate in between itself and a primary input. Consequently, it is significantly easier to create transitions in these gates by setting appropriate logic values at the primary inputs, as compared to the gates away from the primary inputs by several levels of logic.

We can see that a very small percentage of maximum weight is switched for larger ISCAS89 circuits shown in bold at the bottom of Table 3.4. For these circuits, ILP is able to generate a pattern only in the later time-slots appearing towards the end.

The ILP formulation is able to provide a solution, as the total number of gates in these time-slots is very small. A pattern generation approach utilizing circuit bi-partitioning can be preferably done for these circuits.

Table 3.3. Peak current data obtained from ILP for ISCAS-85 benchmarks [57]

Circuit	Circuit Size	Total time-slots	Max. time-slot	Static Max. Weight	Max. Weight ILP	Relative Max. Weight	Execution Time in Seconds	Total Memory KB
c17	6	3	1	2.19	2.19	100	0.06	0
c432	160	18	1	58.41	40.73	69.73	65094	13704
c499	202	12	1	82.40	37.73	45.80	35707	14940
c880	383	25	1	138.43	61.06	46.06	89065	25452
c1355	546	25	1	211.16	64.24	31.86	91256	56343
c1908	880	41	2	268.60	75.85	32.41	81749	42988
c2670	1193	33	2	399.02	134.45	37.18	100199	54860
c3540	1669	48	1	568.50	64.58	16.18	123373	74630
c5315	2307	50	1	856.16	180.84	24.56	125325	778649
c5315	2307	50	1	856.16	180.84	24.56	125325	778649
c6288	2416	125	1	1362.01	140.03	10.28	1728234	333110
c7552	3512	44	1	1567.17	123.34	9.50	413397	132820

Next, we present a validation for the peak current weight switched using the proposed approach.

3.6.3 Validation Using HSPICE

HSPICE was used to validate the result from the pattern generation approach [6]. Firstly, logic simulation was done on the circuit using a pattern pair. During this simulation the number of transitions for each individual time instance were counted and recorded. The current waveform obtained through unit delay simulation model are shown by the green curve in Fig. 3.8. Next, the same pattern pair is applied to HSPICE [6]. Not only the current waveform but also the gate delays are generated by HSPICE. This delay is scaled to be equal to the longest path delay in the unit delay model. Finally the overlaid current waveform is shown in red in Fig. 3.8.

Table 3.4. Peak current data obtained from ILP for ISCAS-89 benchmarks [57]

Circuit	Circuit Size	Total time-slots	Max. time-slot	Static Max. Weight	Max. Weight ILP	Relative Max. Weight	Execution Time in Seconds	Total Memory KB
s208	96	15	1	35.391	17.296	48.87	114	16316
s382	158	21	6	64.566	36.092	55.90	2017	49300
s386	159	10	1	47.178	30.883	65.46	3937	73640
s344	160	12	2	66.586	26.017	39.07	4588	52512
s400	162	10	1	49.159	32.482	66.08	2841	50640
s526	193	13	2	85.281	26.702	31.31	2968	44632
s510	211	10	1	84.866	46.539	54.84	3803	44160
s832	287	75	2	119.993	38.62	32.19	4630	47804
s820	289	11	1	137.15	65.427	47.70	3997	47408
s838	390	11	1	143.034	52.267	36.54	33067	39960
s713	393	57	1	139.836	55.122	39.42	65866	50968
s1238	508	25	2	198.039	52.087	26.30	17426	39964
s1196	529	23	2	217.291	52.964	24.37	19120	37632
s1494	647	60	2	226.688	112.879	49.79	13214	47128
s1488	653	18	1	287.779	54.412	18.91	13149	45608
s1423	657	18	2	291.867	73.962	25.34	53121	46448
s5378	2779	26	3	501.27	216.006	43.09	21164	85776
s9234	**5597**	**59**	**58**	**1483.228**	**1.932**	**0.13**	**92324**	**281784**
s13207	**7951**	**60**	**54**	**1380.799**	**1.117**	**0.08**	**75967**	**272996**
s38584	19253	23	2	217.291	52.964	24.37	283402	2223000
s38417	**22179**	**57**	**49**	**4814.18**	**4.134**	**0.09**	**866263**	**1744556**

As seen from the figure, the overlay shows close proximity between the two waveforms. This experiment shows that the actual switching behavior can be reasonably approximated using unit delay model. Current pattern generation techniques do not work well at the transistor level. Consequently, the result shows that patterns generated at the logic-level can be valuable in assessing actual switching currents.

3.6.4 Comparison of Zero and Unit Delay Models

In this section, we compare the effect of delay model on the worst case switched weight. The maximum weight switched by ATPG considering unit and zero delay models for circuits c432 and c499 is shown in the Fig. 3.10. In order to generate results for the zero delay model, ATPG was done by duplicating the circuit into two copies and then generating ILP formulation which would maximize the total weight

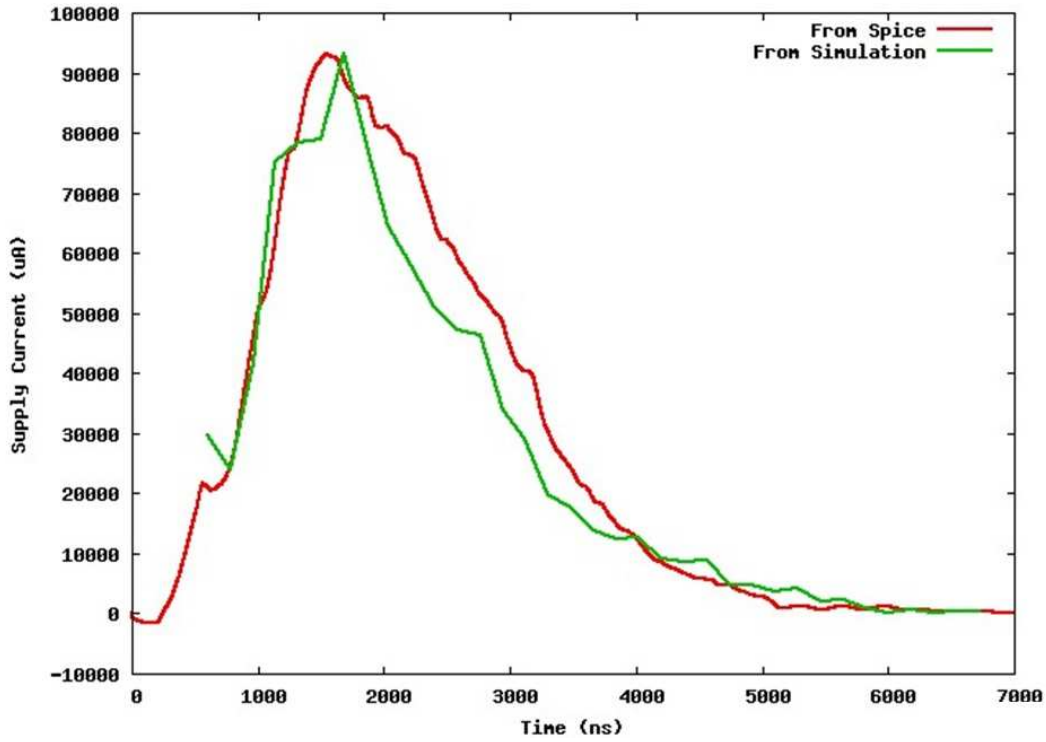


Figure 3.8. Peak current waveform for ISCAS-85 benchmark c7552 obtained through (a) HSPICE simulation (the red line); and (b) logic simulation based on pattern pair obtained from proposed approach [57]

switching between the two copies. The input to the first and seconds copies represent the initial and final patterns, respectively. Consequently, for comparison purpose, the zero delay model can be said to be consisting of just two time-slots corresponding to the first and the second patterns.

Hence, for the zero delay case, the absolute maximum weight for a circuit with G gates is the weight that is switched without considering any Boolean constrains of the circuit. In this case, each of the nodes g, is switched in the direction exciting the higher of $w_P(g)$ or $w_N(g)$. Hence the absolute maximum weight is given by.

$$W_{max}^{ZERO} = \sum_{g \in G} \max \{w_P(g), w_N(g)\} \tag{3.15}$$

This absolute maximum weight obtained for zero delay case is compared to that obtained from the unit delay case. For unit delay case, the absolute maximum weight is calculated for every time-slot after circuit transformation. The maximum weight for time-slot t_i is given by the following equation.

$$W_{max}^{UNIT}(t) = \sum_{g \in G(t_i)} \max\{w_P(g), w_N(g)\} \tag{3.16}$$

This summation is done for all gates $g \in G(t_i)$ appearing in a particular time-slot t_i. The highest of the up and down switching weights for the gate g is returned by the max operation.

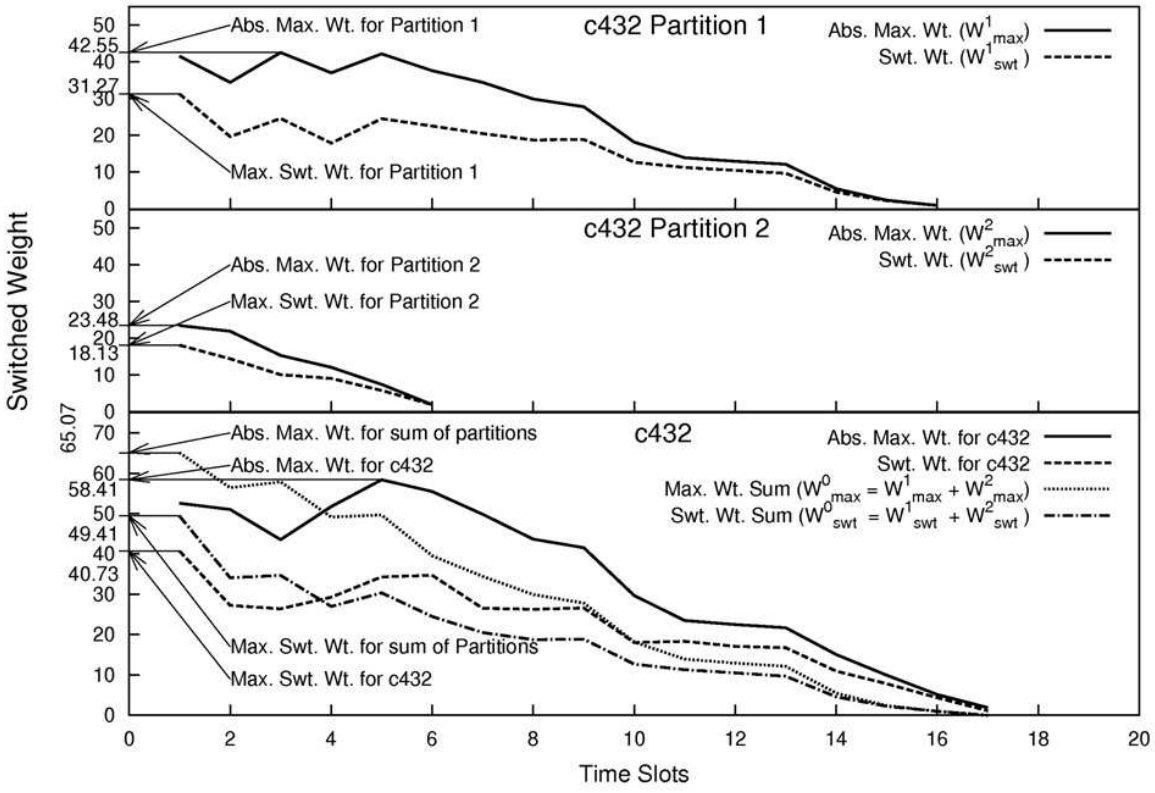


Figure 3.9. ATPG done on c432 after circuit partitioning [57]

The absolute maximum weight switched and the weight switched at first time-slot for the zero delay case by ATPG is represented by the horizontal lines shown in

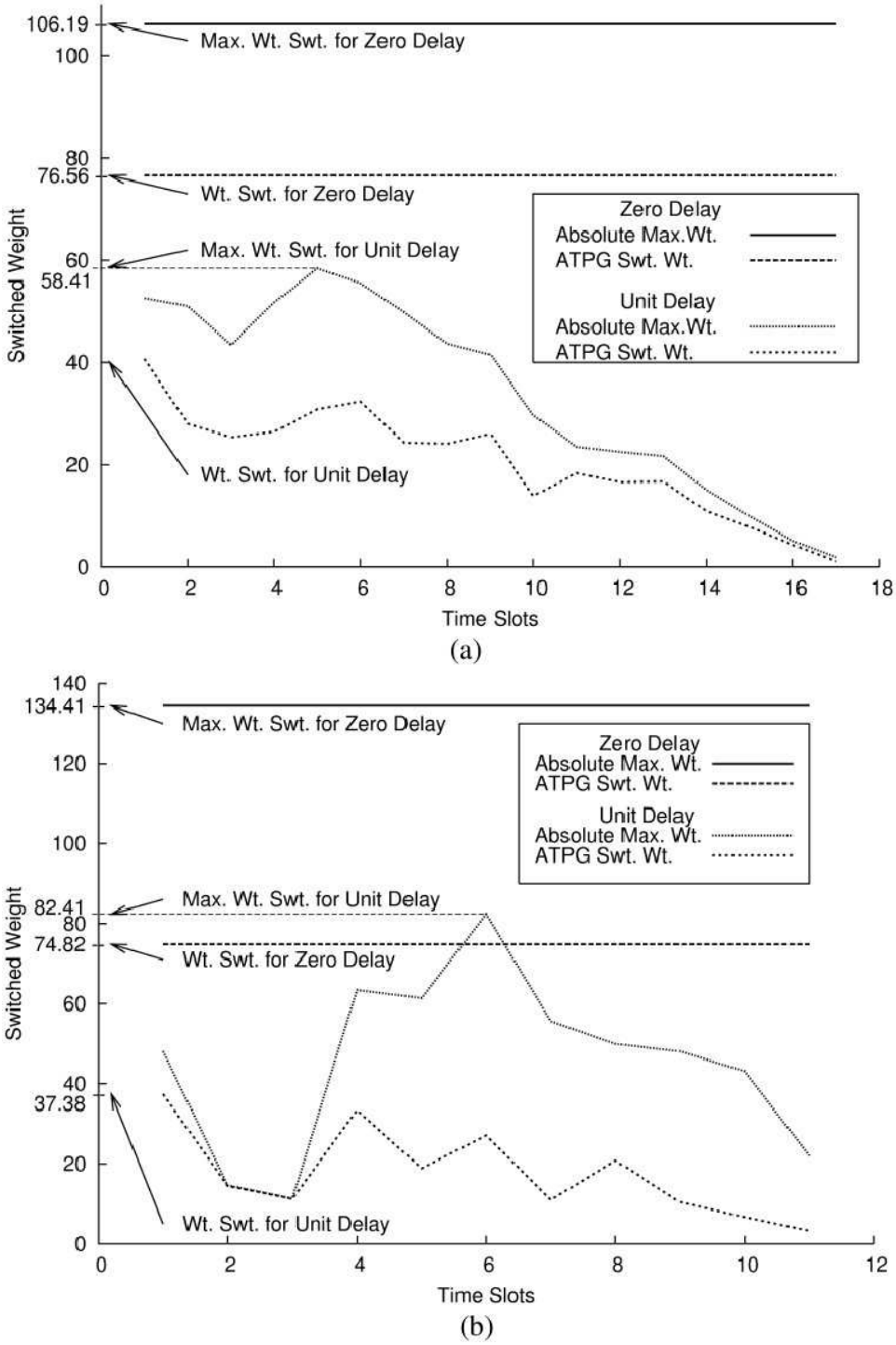


Figure 3.10. Comparison of maximum weight obtained for zero and unit delay cases for (a) c432 and (b) c499 [57]

the Fig. 3.10(a) and (b). We can see that zero delay model overestimates the total weight that can be switched for the circuit. For example, the absolute maximum weight without considering circuit Boolean relationships for zero delay case is 106.19 (134.41) while ATPG is able to switch a weight of 76.56 (74.82) for the circuit C432 (C499). The absolute maximum weight variation is represented by the continuous curve for unit delay case. At the time-slot 5, the maximum of this curve appears (Time-slot 6) with a value of 58.41 (82.41). The ATPG based approach considering unit delays is only able to switch a weight of 40 (37.38) in the first time-slot.

It can be seen that the absolute maximum weight that can be switched, for the unit delay model, without considering Boolean relationships of the circuit varies drastically with time-slot index. Moreover, absolute weight obtained for the zero delay case is much higher than that obtained by ATPG. The importance of the proposed solution is seen from the results which indicate that overall improvement is achieved both from Boolean and temporal dependencies.

3.7 Conclusions

Accurate and efficient analysis of peak current has become a necessary element for power delivery network design and analysis of power supply noise. Peak current estimation is a computationally intractable problem. Moreover, as the state-of-the-art simplified methodologies do not consider Boolean relationship among different gates or their temporal separation from each other, they suffer from significant pessimism. As we consider non-zero gate delay model, in this work, we obtain an improved accuracy of the solution. Non-zero delay model helps us with accounting for the temporal separation between individual gate transitions. Moreover, the non-zero gate delay assumption also helps us in reducing a single instance of the problem size by restricting the focus on only the set of active gates for a given time instant. As a result of this reduction in problem size we are able to obtain an exact solution through

ILP based formulation of the peak current pattern generation problem. Moreover, the proposed delay model is shown to approximate real delay model. The proposed model has a knack for deriving a pattern-pair that produces worst case peak current. The final peak current may be obtained from SPICE simulation of derived patterns, once such a pattern-pair is generated. We can further improve the quality of the solution by considering real gate delays. This is possible, by normalizing actual gate delays to integer values, within the framework of the proposed solution.

CHAPTER 4

RUN-TIME TASK GRAPH EXTRACTION FOR DYNAMIC SCHEDULING IN MPSOCS

4.1 Introduction

Multi-Processor System on Chip (MPSoC) have become prevalent due to increase in transistor density. MPSoCs are composed of multiple processor cores that communicate via a communication back-plane. Various architectures like bus, bridged bus or *Network on Chip* (NoC) may be used for the communication between cores. In an MPSoC, each processor core may be optimized for certain functionality, such as audio, graphics, memory or network. An MPSoC can have a combination of several general purpose cores and various specialized cores performing functions like graphics, multimedia, security and wireless processing [8]. The general purpose cores may be specialized for certain type of applications or vary in performance. For example, a combination of multiple "*Synergistic Processor Elements* (SPEs)" for data intensive processing and a "*Power Processing Elements*" which acts as a controller for SPEs, have been used in the IBM Cell processor [73]. The processor cores in an MPSoC may be further differentiated by power performance characteristics rather than functionality. Some of the examples include, a multiple issue speculative out-of-order core alongside a single issue in-order instruction execution core, both capable of executing the same instructions. Therefore, for power/performance optimization, writing applications on MPSoCs is a challenging task. An application, in SoC programming model, is viewed as a task graph which is a directed acyclic graph. In a task graph, a node represents a task that execute on processor and the edges represent the data

dependencies between such tasks. Each task has an associated functionality type. Moreover, there could be multiple tasks with the matching functionality type. A task can only be scheduled on a processor of the same functionality type. For example, a graphics task may only run on a graphics core. Consequently, task graph scheduling consists of a task to processors mapping so as to minimize the total execution time of a task graph on an MPSoC.

Static task graph scheduling is done during hardware software co-design based on the task execution time estimates and functionality types [106]. Hence, static task scheduling does not take dynamic behavior into account. Consequently, there is a need for adaptability of task scheduling due to following reasons.

- *Disabling some of the resources for a different market segment:* Identical MPSoCs may be targeted to different market segments, in order to save mask development cost and increase production efficiency. This is done, for example, by disabling some features for the low power or low-end markets. In this case, an MPSoC program needs to be re-optimized for each variant.

- *Process Technology Migration:* As a result of the process migration to a smaller technology node, relative core performance may change. Due to the change in feature size, there could be a change in device characteristics and interconnect delays. As a result, the task execution time estimated at design time of an *Intellectual Property* (IP) core can change considerably. Consequently, there is a need for rescheduling tasks to optimize overall execution time.

- *Process Variation:* Due to the presence of process variation, there is an inherent performance discrepancy among chips, during manufacturing. This may lead to variation in processor performance from one die to another.

- *Long term Reliability/Performance Degradation:* Aging related performance degradations may lead to hardware failures in MPSoCs. This underscores the importance of dynamic adaptation of task scheduling.

- *Disabling cores to improve Chip Yield:* Defective chips are reused by disabling cores which are found to be faulty during test, in order to improve chip yield during manufacturing. As these chips have a reduced performance, they can be sold at lower price to different market segments [1].

- *Hardware Design Revision:* Changes in core performance can be caused due to hardware design revisions. Such design revision may involve resource sizing such as changes in instruction issue width, thus affecting sizes of re-order buffer, cache, TLB, branch history table, etc., hence, resulting in changes in performance.

- *Hardware Software Abstraction:* For product development, conventional separation between hardware and software through an abstraction layer is highly desired. This is because, the application designers should not have to worry about optimizing the application for all the variations of hardware platform on which it is running. Thus, there is a need to abstract out the variability in the hardware performance/architecture from the application developers. At the same time, we need to avoid idle times in cores due to poor scheduling.

- *Dynamic Voltage/Frequency Scaling (DVFS):* Dynamic voltage and frequency scaling techniques are used to prevent chips from overheating so as to prevent the creation of temperature hotspots. This reduction in voltage and frequency leads to a decrease in core performance. There can be an increase in performance variation among cores when local DVFS action is taken due to uneven temperature distribution.

This motivates the need for dynamic task scheduling as the task to processor mapping can be adaptively changed based on the processor performance. Either the *Operating System* (OS) or the hardware can be used to perform dynamic scheduling on the fly. The following drawbacks are associated with OS based dynamic scheduling.

As the task graph is known during software development, task scheduling is easier. However, during software development phase, exposing hardware performance to software suffers from problems mentioned earlier. Moreover, committing the hardware to a fixed communication infrastructure or resource sizes reduces the degrees of freedom for hardware designers. Therefore, hardware infrastructure details need to be abstracted out from software development without losing the flexibility of subsequent changes to the hardware. Consequently, our goal is to allow software development to continue without the knowledge about hardware performance. Moreover, we need to allow hardware designers to optimize power/performance without having to worry about its impact on every single software application.

Thus we envision an intermediate embedded layer which separates hardware design and application development processes by optimizing the performance of a given set of applications on a particular hardware. This layer consists of both hardware and software components so as to enable dynamic task graph scheduling to minimize the total schedule length. Ordinarily, the task graph must be available to this intermediate layer during run-time, in order to achieve the above goal. In this work, we present a novel solution that enables run time task graph extraction from the executing software, using the intermediate layer.

The proposed approach aims at improving the performance of an MPSoC application on all variants of hardware while maintaining the traditional separation between hardware and software development process, through a layer of abstraction, without requiring either process to be informed of the internal details of the other. In order to achieve the above goal, we proposed a solution which consisting of following steps.

- Recognize phases in a MPSoC application at run-time through minimal architectural support
- Dynamically extract the task graph associated with each program phase
- Reschedule the task graph with minimal compute overhead to deliver overall performance benefit

Our target platform consists of an MPSoC with asymmetric cores. We show that the proposed approach can improve system performance with *minimal changes to the hardware and software layers*, in todays systems. Some of the contributions and features of our proposed scheme are:

- *Low Cost/Overhead Phase Detection and Classification:* Hardware assisted phase detection is useful, as our scheme relies on fine-grain phase classification within an MPSoC application. We present a new phase detection scheme which tracks the execution frequencies of tasks in an interval. Additionally, the computational demands of the phase are revealed by our *Phase Graph Extraction* (PGE) technique. This can be used for rescheduling tasks that involves task to core mapping.
- *Managing Tasks to Cores Mapping:* Insulating management from the MPSoC OS enables a system level solution which is scalable while allowing the hardware to evolve freely. The proposed process which manages the scheduling of tasks to cores, dynamically adapts from phase to phase as well as within a given phase. An algorithm which performs fast computation of task scheduling decisions is the underlying mechanism used to enable such mapping.

The proposed solution is applicable to many real applications. For example, application program development cannot be done in an optimal way, in Android based devices [2], for all the variants of the hardware employing different kinds of MPSoC

hardware platforms. As our dynamic scheduling solution decouples software development process from the hardware platform used, it is independent of the application and the underlying hardware. In order to perform dynamic scheduling, our solution uses a concealed software layer that is akin to hypervisor which is commercially available [81].

The idea of dynamic adaptation presented here has been used in Transmeta Crusoe^{TM} [61] processor. This processor is able to run X86 instructions on a conventional VLIW based CPU. A software based Code Morphing^{TM} engine, is used here, in order to translate the CISC based X86 instructions optimally at run-time so as to run them on a low power VLIW based CPU. Hence, depending on the customers power/area requirements, the underlying hardware can be changed, without any changes in the original software application.

Some of the static task graph scheduling techniques involve using *Integer Linear Programming* (ILP). These techniques have been used for optimal static scheduling with various goals [105]. However, it can be seen from our results that ILP requires large computation resources. Thus, the gain from rescheduling will be significantly offset by run-time of ILP. Consequently, we need a low cost solution which need not be provably optimal as long as the net benefit is positive. Hence, non-cooperative game theory based dynamic scheduling approach to minimize the total execution time is presented here. This approach is network topology independent and works for buses, bridged buses as well as for NOCs.

Our experiments show that, the proposed phase graph extraction approach can unmask a phase graph within 200 phase graph iterations during a program phase. Moreover, we demonstrate that our proposed scheduling algorithm can perform task migration at run-time.

The rest of the work is organized as follows: Section 4.2 presents the previous work in this area. The MPSoC architecture which is used in this work has been described

in Section 4.3. Then we provide an overview of the proposed approach in Section 4.4. Section 4.5 explains the proposed phase detection approach. This is followed by phase graph extraction in Section 4.6 and dynamic scheduling in Section 4.7. Section 4.8 presents the experimental results and the work finally concludes in Section 4.9.

4.2 Related Work

During hardware software co-design, task graph extraction is known to be used to perform static scheduling. Here, the application task graph is extracted from the application code mapped to a particular hardware. In this context, various task graph extraction techniques have been developed. Vallerio *et al.* [106] present a task graph extraction tool which extracts task graph from a C program of the application. An Abstract Syntax Tree is generated in order to extract task dependencies from the C program. The extracted task names are used to annotate sections of the C code. Then the code is run on the target hardware/simulator to obtain the execution times for each tasks and amount of data transferred is also extracted. Hence, the task execution and transfer times are determined by the target hardware platform. A methodology to extract communication graphs from application at run-time is presented by Liu *et al.* [71]. The data-flow information between multiple threads, is extracted by the tool, by tracking their memory reads and writes. Ganeshpure *et al.* [46] present an on chip task graph extraction that is done for a bus based system, where the arbiter extracts the task graph on the fly. Their approach is only applicable to bus based MPSoC system. Moreover, the extracted task graph is not used for dynamic scheduling.

Phase behavior of an application has been observed in the previous literature [34][81][96]. Program phases are rooted in static structure of programs [46]. Earlier researchers have taken advantage of this time varying behavior by performing reconfiguration of thread to core mapping [53][76][21]. The implementation has to be low overhead and scalable in order to make such phase detection feasible at run-time.

A hardware phase detection scheme based on working set signatures of instructions executed in a fixed interval of time has been presented by Dhodapakar *et al.* [99][35]. Moreover, a hardware based scheme using *Basic Block Vectors* (BBV) to track the execution frequencies of basic blocks touched in a particular interval has been presented by Sherwood *et al.* [96]. They found that, a program executes every basic block only a certain number of times in a given time interval, during a stable phase. A hardware counter based approach for stable phase classification using an *Instruction Type Vector* (ITV) scheme, has been proposed by Khan *et al.* [81]. As, these phase classification schemes were developed for microprocessors they do not directly address the problem of task graph extraction for an MPSoC application. We apply the principle of stable program phases to extract the phase graph from the application. During a phase, a program repeatedly iterates through the same phase graph for a large number of times. Our approach involves a combination of phase detection and phase graph extraction.

Dynamic scheduling is performed on the extracted phase graph. Prior literature presents various scheduling techniques [67][103][11][89][27][108][30] for MPSoCs. A bidding based approach is used by Theocharides *et al.* [103] to perform run-time task allocation. In this approach, a "bid" is send by every processor to the *Task Allocation Engine* (TAE). A bid represents the amount of additional processing capability the processor can handle. The TAE schedules ready tasks to the processors based on the bid value and task deadlines. This technique is greedy and does not take task dependencies into consideration and hence it is sub-optimal. Rao *et al.* [89] use the task graph periodicity seen in multimedia applications to perform battery aware dynamic scheduling. In order to maximize battery life, a greedy approach is used to determine the operating voltages of the cores dynamically. Chen *et al.* [27] present a dynamic scheduling algorithm to reduce the total execution time by interleaving the execution of multiple task graphs. Wang *et al.* [108] present a static scheduling algorithm which

removes the communication latencies between various tasks by interleaving consecutive executions of the task graph. This algorithm is only applicable to streaming applications due to the absence of dependencies between consecutive task graph iterations. Cong *et al.* [30] present various techniques for task graph scheduling with soft deadlines. A cooperative game theory based approach is used for energy aware task scheduling for heterogeneous multiprocessors is proposed by Puschini *et al.* [11]. They provide a solution for a set of independent tasks in the presence of deadlines. In our solution, we take the effects of execution time as well as communication times on the total schedule by modeling the application using a directed acyclic graph.

In our approach, we use a game theory based dynamic scheduling algorithm which generates a new improved schedule on each phase graph iteration. The algorithm converges eventually to a schedule which gives lower phase graph execution time. This approach does not take task graph deadlines into account. Our goal here is to minimize the total schedule length/response time as seen by the user. We explain the preliminaries of the system where phase graph extraction is done, in the next section.

4.3 System Description

As our phase graph extraction approach is generic, it can be used for any MPSoC architecture. An MPSoC consisting of a set of asymmetric processor cores that communicate through an communication network, is used as the target platform. This is shown in the Fig. 4.1. Each of the *Processing Elements* (PE) is optimized so as to provide high throughput for a particular functionality type. It should be noted that, we will interchangeably use the word "processor" and the acronym "PE" to represent a processor core in an MPSoC.

The applications that are run on the MPSoC consists of a combination of multiple phase graphs, each of them corresponding to a different stable phase.

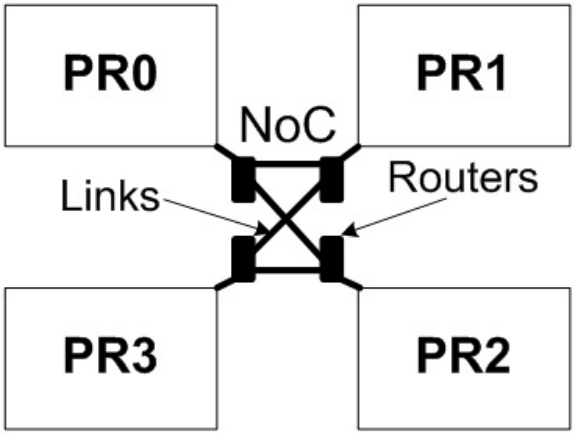


Figure 4.1. An example of an MPSoC with 4 PEs communicating with an NoC

Definition 4.3.1. When an application executes a sub-graph of the original task graph repeatedly for a large number of iterations, it is said to be going through a stable *Phase*.

Definition 4.3.2. When an application is going through a phase, a sub-graph within an application task graph, which repeatedly executes for a large number of iterations, is called a *Phase Graph*.

Program structures that tend to execute in loops, are responsible for phase behavior of an application. The application task graph, as shown in Fig. 4.2, consists of an *Initialization Task Graph* which only executes once. This is followed by one or more phases, each of which consist of a phase graph repeating for millions of iterations. The application is said to be showing phase behavior, during this time, and is said to be going through a stable phase. Finally, after running through a *Termination Task Graph*, which executes only once, the application may end its execution. In this work, first we detect the presence of a stable phase, followed by extracting the associated phase graph.

As mentioned earlier, every task in a phase graph has an associated functionality type. This represents the kind of execution performed by the task. A given task can

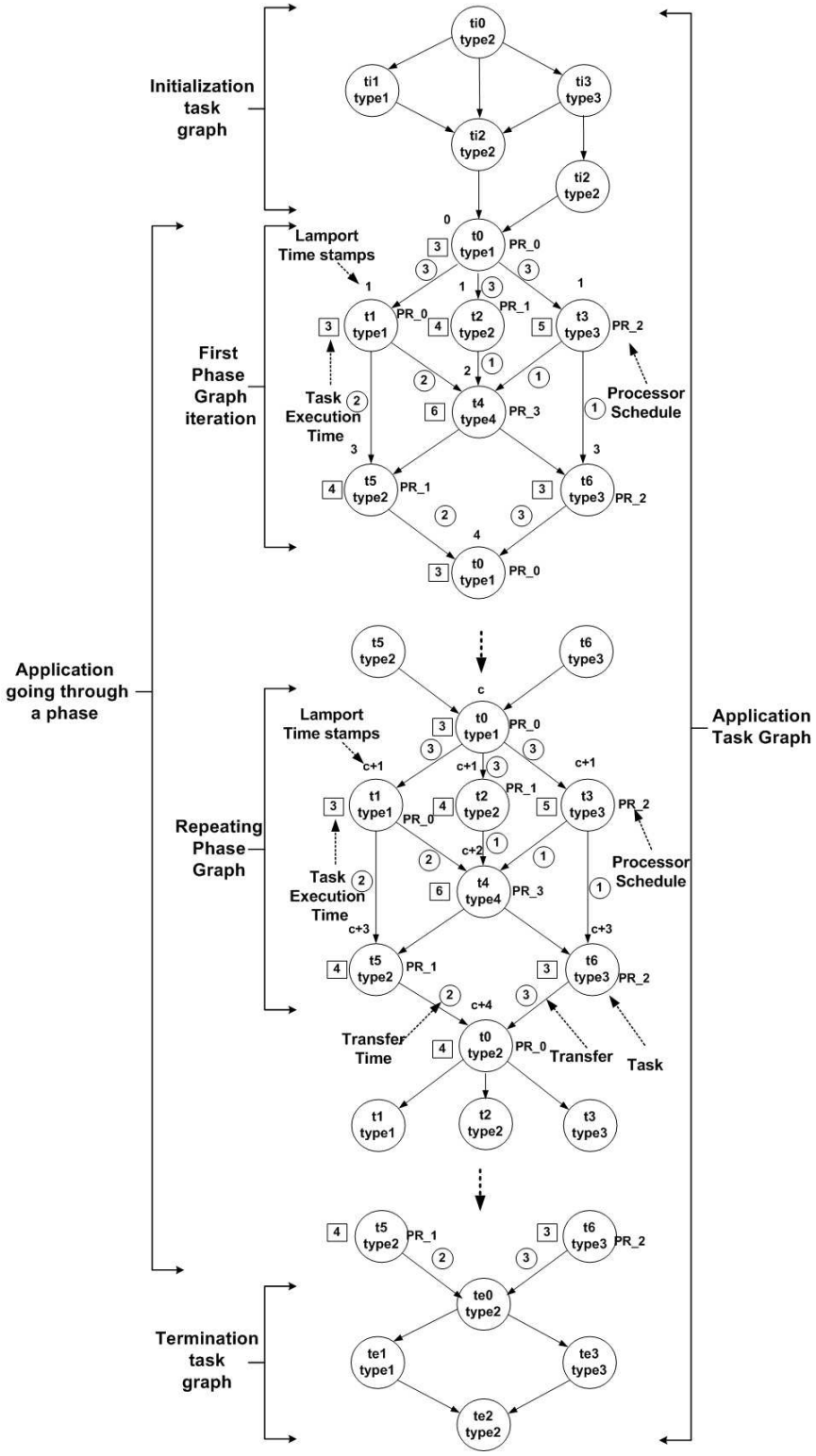


Figure 4.2. Example of an application going through a phase

be scheduled only onto a processor with a matching functionality type. In order to perform phase detection, we assume here that initially the processor assignment to task is unchanging, as the application is statically scheduled. Consequently, every task of a particular type is scheduled onto a processor of the matching type.

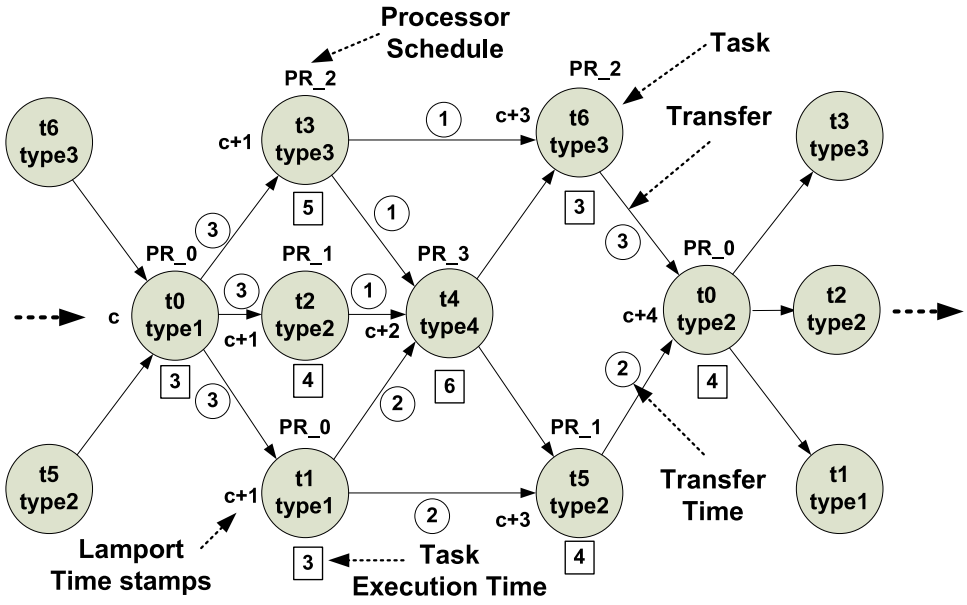


Figure 4.3. An example of recurring phase graph

Phase graph execution during a program phase, where the same phase graph is executed repeatedly, is shown in Fig. 4.3. Task execution and transfer times are represented by the numbers in boxes below the nodes and those in circle above the edges, respectively. The processor names above the nodes, represent the phase graph schedule and the task type is shown below the node name. The *Start Node* of the phase graph is given by node $t0$. Nodes $t5$ and $t6$ send data back to node $t0$, at the end of phase graph iteration. This starts the next iteration.

The processing elements communicate by sending data to destination PEs through the NoC. Each processor has a local scheduler which keeps the list of all the tasks that are scheduled onto the PE, as the task graph is statically scheduled to the

MPSoC. This scheduler keeps a track of data received from various PEs. When input dependencies of any of the scheduled tasks is satisfied it starts task execution.

In order to maintain a partial ordering of tasks, the scheduler also generates a set of *Lamport Time-stamp* [13]. It should be noted that, we will use the word "time stamp" to refer to the "Lamport Time-stamp" in the rest of this chapter. In order to keep track of the time-stamp, each PE holds a counter which is incremented at the end of every task execution. This counter value (Lamport time-stamp) is piggybacked with the data, when a PE sends a message. On receiving a message, the PE sets this counter to the maximum value among all the received time stamps. In case of an overflow, this counter restart from zero.

The time-stamps for task $t0$ is 'c' as shown in Fig. 4.3. It gets incremented to $c+1$, after being transmitted to tasks $t1$, $t2$ and $t3$ and after the above tasks have finished execution. A time stamp value of $c+1$ and $c+2$ is received by task $t6$, from $t3$ and $t4$, respectively. On the receipt of these time stamps for the task $t6$, it is set to the maximum value of $c+2$ and then incremented to $c+3$. At the end of a task graph iteration, it can be seen that the time-stamp increments by a value of 4 (changing from c to $c+4$).

4.4 Proposed Approach Overview

Consider a statically scheduled phase graph executing on the heterogeneous MPSoC. In this approach, we need to extract execution phases from the application and perform dynamic scheduling so as to minimize the task execution time. Our proposed approach can be divided into (i) phase detection (ii) phase graph extraction and (iii) dynamic scheduling. Next, we give a brief overview of each of the above steps.

- *Phase Detection:* Based on the observation that a typical application goes through phases of execution, we perform phase graph extraction. When an application goes through a phase, the same phase graph repeats for large num-

ber of times. Consequently, the same application statistics are repeated, in a given phase. In this step, phase detection is done by keeping track of these application phase statistics. This can be divided into the following sub-steps.

  - *Leader Election:* This one-time pre-processing is done before the application starts running. Leader election [13] is used to identify one of the PEs as a *Leader*. Any PE can act as a leader. A leader is responsible for phase graph extraction and dynamic scheduling. A PE which is not a leader is referred to as a *Follower*.
  - *Phase Extraction:* In this step, phase detection is done by each follower by periodically measuring the relative number of times various tasks are executed on the PE. Destination list and execution time measured on the PE are used to identify the task. If the same statistics are repeated for consecutive periods, a phase is detected at the follower. On successful detection of a *local phase* the follower informs the leader. A phase is detected globally if all the followers detect a phase.

- *Phase Graph Extraction:* As soon as a phase is detected globally, the leader informs all the followers by broadcasting a global phase detected message. This causes the followers to start sending *Execution Status Word* (ESW) to the leader at the beginning and end of every task execution. Task ordering and dependency information for each of the tasks executed on the system, is represented in the ESWs. The phase graph is extracted by the leader based on the dependency information in the received ESWs. This step is divided into the following set of sub-steps, as explained next.

  - *Data Dependency Extraction:* The leader uses the input/output dependency list present in each of the ESWs in order to extract dependencies among the tasks.

- *Phase Graph Generation:* This data dependency graph which was extracted by the leader in the previous step consists of multiple repetitions of the same phase graph. The leader has to determine the start node and the number of nodes in the phase graph, in order to extract the same. This is extracted from the dependency graph, by the leader, using a window based approach explained later.
- *Timing Extraction:* By measuring the time difference between the receipt of the ESWs sent at the beginning and end of a task, the execution time for the task is extracted. The of number of bytes of data transferred is used to measure the transfer time. The execution time of a particular task on all the other processors is determined by the leader, by scheduling duplicated tasks on idle processors and measuring the associated execution times. Hence, it obtains an estimate of execution times, for every task, on various processors with matching type.

- *Dynamic Scheduling:* The leader performs dynamic scheduling in order to reduce the total execution time of the phase graph, once phase graph is extracted successfully. We propose a novel non-cooperative game theory based approach for this purpose. Once a new schedule is determined, the leader sends this newly generated optimal schedule to the followers.

The next sections explains each of the above steps one by one in greater detailed.

4.5 Phase Detection

In this step, the detection of application phase behavior is done at run-time. It consists of the following two sub-steps which are explained next in detail.

4.5.1 Leader Election

This step is executed initially during the system boot-up. The process of phase detection, task graph extraction and dynamic scheduling are co-ordinated by the leader. As explained earlier, the PEs other than the leader are called followers. If a task is scheduled on the leader, it can also act as a follower. In that case, the leader will have to share its computation resources between system coordination and task execution. For leader election, various algorithms are available in literature [13]. We will not go into the depth of these algorithms in this text. Consequently, the readers are encouraged to go through the above references on their own.

4.5.2 Phase Detection

In this step, phase detection is performed by each of the followers independently. Once a phase is detected locally, the leader is informed by the follower. Now, the leader keeps a track of all the followers for which a phase was detected. When all followers have detected a phase, the leader multi-casts a phase detection successful message. Distributed phase detection is used here, in order to minimize the total number of additional messages needed to be transmitted. An alternate centralized phase detection approach will only involve the leader. Consequently, in the centralized case, a large number of messages will be transmitted to the leader. This is because, the leader will have to keep track of every task executing on the system in order to detect the presence of a stable phase.

The relative number of times, various tasks are executing on the follower, is tracked using the *Follower Phase Vector* (FPV). Each element of the follower phase vector is composed of *Task Information* and *Phase Count* fields. A task is identified using the task information field which consists of a tuple of task execution time and destination processor list. The destination processor list is the list of destination PEs for the current task, while the task execution time is measured in terms of the number of

execution clock cycles of the task on the follower. The count of the number of times a given task information is generated is stored in the phase count field. A pair of task information fields match if their execution time differ by less than 10% of the minimum and if they have the same destination list. This threshold value is used, in order to account for the small amount of variation in execution time of a task, each time it executes on the processor. Suppose for a task t, the execution times corresponding to the first and the second task executions on a particular processor is given by ex_t^1 and ex_t^2, respectively. Consequently, corresponding to the above two task executions, the task information fields are said to match if only if their destination list match and their time difference $|ex_t^2 - ex_t^1| < 10\% \cdot \min\{ex_t^2, ex_t^1\}$. It should be noted that, task information is not a unique task identifier. This is because two different tasks with the same destination list and same execution time will be indistinguishable. In this work, it is assumed that these characteristics of a task will change when a change of phase is encountered.

A new FPV is generated at every regular intervals known as *Sampling Interval*. A new FPV is generated when the time-stamp increases by an amount equal to the sampling interval value, with respect to the end of previous sampling interval. The newly generated FPV is compared with the previously generated FPV and the new FPV is copied to the previous, at the end of every sampling interval. A pair of FPVs matches, if all of their task information fields and count values match. When a pair of FPVs match consecutively for more than *Follower Phase Match Threshold*, a stable phase is detected in the application.

As soon as the follower detects a phase it informs the leader. The leader counts the number of times phase is detected at every processor by using a *Leader Phase Vector* (LPV). Each element of an LPV is formed using a processor identifier and a counter. The i^{th} element of LPV is incremented, if a phase is detected on the follower PE_i. The counts for the LPV elements increment, as the leader receives subsequent

phase detected messages from the corresponding followers. If the FPVs in consecutive sampling intervals don't match, the follower goes through a phase transition. In that case, it informs the leader about the end of previous phase and the leader resets all the LPV counters. When the count values for all the non-zero elements increase above *Leader Phase Match Threshold*, a global phase is detected.

Table 4.1. FPV generated for a sampling interval of 100

PE identifier	PR0		PR1		PR2		PR3L
Task Info.	3:0,1,2	3:1,3	4:3	4:0	5:2,3	3:0	6:1,2
Phase Count	25	25	25	25	25	25	25

Example 4.5.1. The FPVs generated at all the PEs is shown in Table 4.1. This is generated when the phase graph in Fig. 4.3 is run on a 4 processor MPSoC shown in Fig. 4.1, for a sampling interval of 100. The first row represents the processor on which the FPV is generated, while the task information and the phase count fields of the corresponding FPV are represented in the second and third rows. Here the PE $PR3$ has a dual role of a leader as well as a follower. Now $PR0$ is executing task $t0$ with an execution time of 3 after which it transfers data to $PR0$, $PR1$ and $PR2$. Hence, for $PR0$, the task information field is given by 3(execution time): 0(destination $PR0), 1(PR1), 2(PR2)$. Similarly, the task information vectors for $PR1$, $PR2$ and $PR3$ can be obtained.

The number of times a particular task information is encountered between two consecutive sampling intervals is shown by the phase count value in Table 4.1. Consider the scenario where a new sampling interval has started when task $t0$ is executing on $PR0$. At this time, the initial time-stamp value of c is temporarily stored in a register at $PR0$. It can be seen that, this value increases to $c+4$ for the next phase graph iteration, starting from an initial time-stamp c for the task $t0$. Thus the time-stamp value increases by 4 for every phase graph iteration.

Now, at each of the processors, the current and the initial time-stamp values is compared to the sampling interval. When the above difference becomes greater than or equal to the sampling interval of 100, then end of the current sampling interval is reached. A processor $PR0$ running task $t0$ and $t1$, encounters a sequence of time-stamps $\{c, c+1\}$, $\{c+4, c+5\}$ and $\{c+8, c+9\}$ for the first, second and the third iteration, respectively. As explained earlier, the initially stored time-stamp at the beginning of the sampling interval is c. Now, when the time-stamp at $PR0$ becomes $c+100$ for which the difference $((c+100)-c) = 100$, the sampling interval at $PR0$ ends. This happens at the end of 25 phase graph iterations. As tasks $t0$ and $t1$ execute once for every iteration, their corresponding phase count value increments by 1 for each of the iterations. Hence, at the end of a sampling interval, for $t0$ and $t1$, the total phase count is 25.

Current phase vectors are compared with the previous once at the end of every sampling interval. In this example, as the same phase vector is generated every sampling interval (25 iterations), it can be seen that all the elements of the phase vector match exactly. When we get 3 consecutive matches or after 75 phase graph iterations, a follower phase is detected.

The follower phase is detected every 75 phase graph iterations, as shown by the example in Table 4.1, after which, each of the followers send a phase detected message to the leader. As a result, the leader increments the phase detected counter for the corresponding processor in the LPV. The count values for all the processors in the LPV at the end of 375 phase graph iterations becomes $375/75 = 5$, corresponding to the leader phase match threshold of 4. Consequently, leader phase is detected as this value is greater than the phase count threshold. ■

For the follower phase vector, there is a limit on the maximum size. The FPV is pruned by removing elements with smallest count values, if the size increases beyond this limit. The size of LPV at the leader is same as the number of PEs in the system.

Consequently, it can be seen that, a very small amount of memory is required to store these phase vectors.

4.6 Phase Graph Extraction

Once the leader detects a phase, it informs all the followers about it. The followers transmit dependency information of the tasks that are executing on them, to the leader in the form of ESWs, after they receive the successful phase detection message. These ESWs are used by the leader for phase graph extraction. The above step is divided into the following sub-steps:

4.6.1 Data Dependency Extraction

On receiving the phase detected message, the follower starts sending ESWs to the leader at the beginning and end of every task execution. These ESWs generated at the beginning and end of task execution are called *Start ESW* and *End ESW*, respectively. The start ESW contains a task identifier which consists of a combination of processor identifier, task counter and time-stamp. Similarly, the end ESW consists of an *Input Dependency List.* in addition to the task identifier present in the start ESW. The input dependency list contains a list of task identifiers which were received by the current task from its input dependencies. Moreover, after receiving phase found message, each follower also starts incrementing a task counter. Whenever a new task starts its execution on the processor, this counter is incremented. The task counter is used for task graph extraction by the leader, as it acts as a unique identifier for every task that is executed on the processor.

The start and end ESWs generated for task $t4$ from the Fig. 4.3, is shown in Fig. 4.4. The generation of start and end ESWs at the beginning and end of $t4$, is shown in part (a), while part (b) shows the format of start ESW. For $t4$, the start ESW consists of the task identifier which has the processor name ($PR3$), task counter

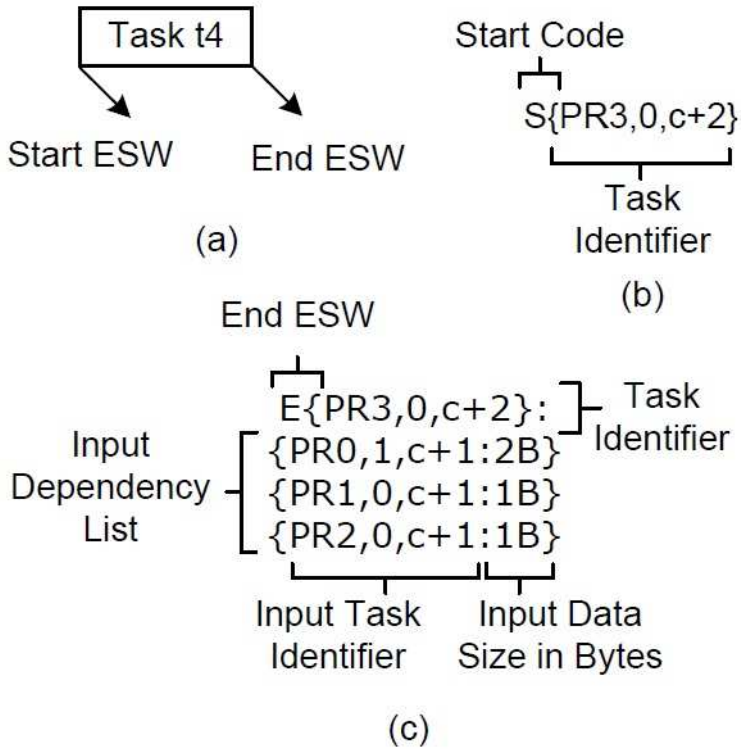


Figure 4.4. Start and end ESWs generated by task $t4$ (a) start and end ESW transmission (b) start ESW format (c) end ESW format

(0) and the time-stamp $(c + 2)$. As shown in part (c), the end ESW consists of the task identifier concatenated with the input dependency list. This task identifier is the same as the start ESW with the only difference that it is prefixed with an 'E'. The task identifiers received with the data from processors $PR0$, $PR1$ and $PR2$ executing tasks $t1$, $t2$ and $t3$, form the input dependency list. Moreover, it also holds the number of KiloBytes (KB) of data that is received from each processor. For example, it receives 2KB from $PR0$, 1KB from $PR1$ and 1KB from $PR2$.

The *Dependency List* is generated by the leader as it receives these ESWs. The nodes and the edges of this dependency list are the task identifiers of the received ESW and the input dependency information stored in the ESWs, respectively. These nodes of the dependency list are stored in increasing order of their time-stamps.

4.6.2 Phase Graph Extraction

The leader extracts the start node (node $t0$ in Fig. 4.3) and the total number of nodes in the phase graph (7 for task graph Fig. 4.3), in order to do phase graph extraction, as the followers may receive phase detection packet at any intermediate point of execution of the phase graph. Further, the phase found message may not be received by all the followers at the same time. Therefore, the dependency list generated at the leader will not necessarily start at the node $t0$. Thus, the leader has to determine the start node, in order to extract the final phase graph. Moreover, the total number of nodes in the phase graph, is to be also determined by the leader. Next, the heuristics for start node and phase graph size detection are explained.

1. *Start Node Extraction:*

The dependency list is generated on the fly by the leader as it receives ESWs from the follower. This is done by storing the received ESWs in the dependency list in the increasing order of time-stamps. Moreover, a list of nodes which are candidates for the start node are also identified. In the dependency list, a given node is a candidate for a start node if it is the only node with its time-stamp and there are no edges from the nodes having lower time-stamps, appearing before the given node, to those having a higher time-stamps appearing after the given node. A $TRUE$ value in the Boolean field called $nodeIsCandidate$, stored at every node in dependency list, is used to represent the edge condition. Start node detection is initiated only after the leader receives $2 \times extractionWeightLength$ ESWs (dependency list nodes), in order to make sure that all the nodes in the input and output dependencies of a given node have been stored in the dependency graph. In order to make sure that all the input dependencies of the currently arriving node are present in the dependency list, the first $extractionWeightLength$ received nodes are skipped. Similarly, the last $extractionWeightLength$ of nodes from the end are skipped to make sure that all the output dependencies of the given node have arrived. Hence, for start node detection

as well as task graph size extraction, only a section of the dependency list starting from $extractionWeightLength$ and ending at $extractionWeightLength$ nodes before the end of the dependency list is considered. These nodes, which are present in the above mentioned range, are called *Valid Nodes*.

The heuristic *bool* $get\_start\_node(currentNodeLoc, *startNodeLoc)$ as shown in the Pseudocode 4.1, is used for start node detection. This heuristic is called whenever the leader receives a new ESW/node. The index for minimum time-slot, for a newly arrived node, is determined in Line 2. We iterate through all the nodes with time-slots between $minTS$ and $currentTS$ and set the $nodeIsCandidate$ condition to $FALSE$, in Lines 4-6. This step has a time complexity of $O(n^2)$, for a dependency list of size n. The condition whether the $waitCounter$ is higher than $2 \times extractionWeightLength$ is checked in Line 7. Start node detection is performed in Lines 11 to 17, once this condition is $TRUE$. As all the output dependencies of the node are specified, we start from the node index $i = (dependencyList.length - extractionWaitLength)$. Then, the previous, current and the next time-slots are determined in Lines 12 to 14. This is followed by, start node detection, which is done by checking if the current node is a candidate node and whether the node is the only node in its time-slot, by comparing the current time-slot with previous and next in Line 15. Once, the start node is determined, phase graph size extraction is done.

2. *Phase Graph Size Extraction:*

The leader uses the task information field of received ESW to generate a *Task Alias*. A combination of source processor identifier and the input dependency list obtained after removing the task count and time-stamp is used to form the task alias. Task alias is used to uniquely identify an ESW corresponding to a task when it is received next time by the leader. Thus, there is a task alias, for every task in

Algorithm 4.1 $get\_start\_node(currentNodeLoc, *startNodeLoc)$

Initialization:
$waitCounter = 0$

```
1: dependencyList[currentNodeLoc].nodeIsCandiate = TRUE
2: minTS = min_timeStamp_in_all_inputs(currentNodeLoc)
3: currentTS = dependencyList.timeStamp
4: for (i = 0; minTS < dependencyList[i] < currentTS; i++) do
5:    dependencyList[i].nodeIsCandiate = FALSE
6: end for
7: if waitCounter < (2 * extractionWaitLength) then
8:    waitCounter++
9:    return FALSE
10: else
11:    i = dependencyList.length − extractionWaitLength
12:    previousTS = dependencyList[i − 1].timeSlot
13:    currentTS = levelizedList[i].timeSlot
14:    nextTS = levelizedList[i + 1].timeSlot
15:    if (dependencyList[i].nodeIsCandiate == TRUE&&previousTS < currentTS&&
       currentTS < nextTS) then
16:       (*startNodeLoc) = i
17:       return TRUE
18:    end if
19: end if
```

the dependency list. The minimum period of the repetition pattern of the alias list determines the phase graph size.

In the following Pseudocode 4.2, *storedAliasLoc* and *newAliasLoc* points to the previously stored alias in the dependency list and to the alias corresponding to the node which has recently become valid, respectively. Every time a new alias becomes valid, these aliases pointed to by the two pointers are compared. The set of previously stored aliases, is compared with the newly arrived alias, to determine if a match happens. The location of the first match after a set of mismatches, is pointed to by $firstMatchLoc$. Both the *storedAliasLoc* and the $firstMatchLoc$ are initialized to the start node location *startNodeLoc* obtained from *get_start_node*() heuristic. Every time a new ESW arrives at the leader after start node detection, *get_task_graph_size*() is called. We obtain *storedAlias* and *newAlias* corresponding to the *storedAliasLoc*

and *newAliasLoc*, respectively, in Lines 1-3. The condition, if a pair of aliases match or not, is checked in Line 4. The *storedAliasLoc* and $firstMatchLoc$ are initialized to *startNodeLoc*, whenever there is a mismatch between *storedAlias* and *newAlias* values. Now, when the first match happens after a set of mismatches, $firstMatchLoc$ is set to *newAliasLoc*. This is followed by comparing the aliases at *storedAliasLoc* and *newAliasLoc* and subsequently incrementing *storedAliasLoc* whenever a match happens. Thus, in the first and the second repetition of the phase graph, *storedAliasLoc* and *newAliasLoc* point to corresponding alias. If all the aliases corresponding to *storedAliasLoc* and *newAliasLoc* match for *storedAliasLoc* starting from *startNodeLoc* to $firstMatchLoc - 1$, then the phase graph repetition is said to be extracted. $firstMatchLoc$ is set to *newAliasLoc* as seen in Lines 8 to 10, when *storedAliasLoc* increments and becomes equal to $firstMatchLoc$. Hence, whenever a match is detected, $windoMatchCount$ is incremented. The above process continues until $windoMatchCount$ crosses $windoMatchCountThreshold$ value as shown in Lines 11-14. At this time, the phase graph size of (*newAliasLoc* − *storedAliasLoc*) is obtained.

Start node determination has a complexity of $O(n^2)$ while phase graph size determination heuristic has an $O(n)$ complexity. For the dependency list, the memory complexity is given by:

$$O(2 \times extractionWaitLength \times phaseGraphSize.windowMatchThreshold) \quad (4.1)$$

A phase graph is not extracted, if the phase graph size is more than the available memory and the leader sends a message to the followers to stop generation of ESWs.

4.6.3 Timing Extraction

The task execution and transfer times are extracted by the leader, in this step. As explained earlier, the followers start sending start and end ESWs to the leader,

Algorithm 4.2 *get_phase_graph_size*($currentNodeLoc, *TGSize$)

Initialization:
$firstMatchLoc = startNodeLoc$
$storedAliasLoc = startNodeLoc$
$windowMatchCount = 0$

```
1: newAliasLoc = currentNodeLoc − extractionWaitLength
2: storedAlias = dependencyList[storedAliasLoc].alias
3: newAlias = dependencyList[newAliasLoc].alias
4: if (storedAlias == newAlias) then
5:    if (firstMatchLoc == storedAliasLoc) then
6:       firstMatchLoc = newAliasLoc
7:    else
8:       storedAliasLoc++
9:       if (firstMatchLoc == storedAliasLoc) then
10:         firstMatchLoc = newAliasLoc
11:         if (windowMatchCount > windowMatchThreshold) then
12:            (*TGSize) = (newAliasLoc − storedAliasLoc)
13:            return TRUE
14:         else
15:            windowMatchCount++
16:         end if
17:      end if
18:   end if
19: else
20:   storedAliasLoc = startNodeLoc
21:   firstMatchLoc = startNodeLoc
22: end if
23: return FALSE
```

when it receives a phase detected message. Corresponding to each of the PEs, the leader holds a set of timer registers. The corresponding timer register is reset, on receipt of the start ESW. Until the end ESW is received, it starts counting the leader clock cycles. Here we assume that, as compared to the total task execution time, the variation of the total transmission latency of subsequent transfers from any of the PEs to the leader is small.

The number of bytes of data which is received by the task, obtained from the ESW, determines the edge transfer times. It is assumed that, the transfer time is directly proportional to the number of data bytes transferred, for a uniform flit size

of the underlying NoC. Moreover, we also assume that the latency involved in setting up a path from source to destination is small as compared to the total transfer time.

We need to estimate the execution time of every task on all the processors of a particular type, in order to perform dynamic scheduling. The leader can determine execution times of every task on other processors of the same type, once the phase graph is extracted and execution times of each task statically scheduled to a processor is known. In each phase graph iteration, the leader schedules a copy of a given task on every other processor of the matching type whenever they are idle. These duplicate tasks only transfer the start and end ESWs to the leaders. Hence, no data is transmitted on the NoC by these duplicated tasks to the destination processor list of the task. Consequently, leader estimates execution time of the current task on all other processors, based on the received ESWs.

4.7 Dynamic Task Scheduling

Once the task dependency information is extracted and execution times of each of the tasks on all processors with matching types are estimated in the phase graph extraction step, the leader performs dynamic scheduling. In this work, a *Non-cooperative Game Theory* based approach [80] is used to minimize the total schedule length of the phase graph. Our approach is inspired by that used by Puschini *et al.* [85] where a game theory based technique was developed for the generation of frequency assignments for tasks so as to minimize temperature in SoCs. Next, we formulate the dynamic task scheduling problem using a non-cooperative game theory based formulation as explained next.

Consider a set of N players $T = \{t_1, t_2, ..., t_i, ..., t_N\}$ who make decisions independently. Here, each player t_i has an associated type k. Moreover, there is a set of choices represented by $p_i = \{p_{i1}, p_{i2}, ..., p_{ij}, ..., p_{iM}\}$, corresponding to each player type. In this case, the number of possible choices of type k is represented by M.

For every player t_i of type k, there is an associated cost corresponding to each choice which is calculated based on the previous *Game Cycle*.

Table 4.2 shows the analogy between the phase graph scheduling problem and the game theoretic approach for dynamic scheduling.

Table 4.2. Comparison between non-cooperative game theory and dynamic task scheduling

Non-cooperative games	Dynamic Phase Graph Scheduling	Symbol
Players	Tasks	t_i
Type	Task Execution Type	k
Choices	Processors with matching type	p_{ij}
Game cycle	Single Phase Graph execution cycle	-
Cost Function	Total Schedule length	C_i
Objective	Minimize the total schedule length	-

Definition 4.7.1. The smallest duration within which all the players t_i have made a "move" involving the selection of a choice from the set of available choices p_{ij}, is called a *GameCycle* [85].

In the current game cycle, the choices are made, for each of the players t_i, in such a way that the total cost function is minimized. There is a real cost C_i, corresponding to every player t_i, which represents the cost associated with the set of choices for the player. Let p_{ij} be the choice made by the player t_i in the current game cycle and p^*_{ij} represents the choice made by other players in the previous game cycle. Then the cost associated with the current choice p_{ij} is given by $C_i = f\left(p^*_{1j}, p^*_{2j}, ..., p^*_{ij}, ..., p^*_{Nj}\right)$ where $f : P_1 \times P_2 \times ... \times P_N \rightarrow \mathbb{R}$. For each player, the objective of the game is to make choices so as to minimize its cost.

Definition 4.7.2. The set of choices made by all the players, for the above non-cooperative game, is a *Nash Equilibrium* if $C_i\left\{p^*_1, p^*_2, ..., p^*_i, ..., p^*_N\right\} \geq C\left\{p^*_1, p^*_2, ..., p_i, ..., p^*_N\right\} \forall p_i \in P_i$. Here, the Nash Equilibrium choices for all the players including t_i is given by $p^*_1, p^*_2, ..., p^*_i, ..., p^*_N$[85].

From the comparison shown in Table 4.2 it can be seen that there is a set of PEs P_i on which the task can be scheduled on to, for every task t_i of type k. Whenever a task is ready to execute, a processor is selected, which leads to the least increase in the schedule length (Cost C_i), which is calculated based on the current choice and the choices made by other processors in the previous game cycle. This scheduling technique is explained in the following sub-section.

Algorithm 4.3 $dynamic\_task\_schedule(currentTask, procName, startTime)$

Initialization:
$sameMinValueCount = 0$
$sameMinValueLimit = 5$
$globalMinCost = \infty$
$minCost = 0$

```
1: readyTasks = find_ready_tasks(currentTask)
2: for all taskName ∈ readyTasks do
3:    if (sameMinValueCount > sameMinValueLimit) then
4:       procName = gen_random_schedule(taskName)
5:       globalMinCost = ∞
6:       sameMinValueCount = 0
7:    else
8:       procName = get_sch_proc(taskName, procName, &minCost)
9:    end if
10:   send_schedule_to_proc(taskName, procName)
11:   if currentTask == startNode then
12:      minCost = ∞
13:      if (globalMinCost > startTime − prevStartTime) then
14:         globalMinCost = startTime − prevStartTime
15:         prevStartTime = startTime
16:         sameMinValueCount = 0
17:      else
18:         sameMinValueCount + +
19:      end if
20:   end if
21: end for
```

4.7.1 Dynamic Task Graph Scheduling Algorithm

The proposed game theoretic approach is explained in the Pseudocode 4.3. When the tasks have already sent the start and end ESWs at the end of a task execution,

the function *dynamic_task_schedule*() executes on the leader. The $startTime$ of the task is determined by the arrival time of the start ESW. The $find\_ready\_tasks()$ function in Line 1 is used to determine the total number of tasks which become available for execution (ready tasks) at the end of execution of the current task. This is followed by using the *get_sch_proc*() function in Line 8, to determine the processor on which to schedule the ready tasks and then sending the schedule to the processors using *send_scheduled_proc*() function in the Line 10. We check if the current phase graph iteration has ended by checking if the received task is the $startNode$ of the next iteration, in Line 11. The difference $startTime - prevStartTime$ is calculated to measure the actual task graph execution duration and the minimum value of this schedule length obtained is tracked by using the variable $globalMinCost$ in the Lines 13 and 14. The number of times this $globalMinCost$ did not change, is counted by the $sameMinValueCount$ in Line 18. In order to extricate the heuristic from local minimum, task perturbation is done when the minimum schedule length does not improve with subsequent task graph iterations. In Lines 3 and 4, the new schedule processor is determined randomly using *purturb_schedule*() if the $sameMinValueCount$ has gone above the $sameMinValueLimit$ value. This function generates a random number and, if the generated random number value is less than $taskPurturbThreshold$, then it perturbs the task schedule. The new scheduled processor is selected randomly, with uniform probability distribution, from the list of the available processors. Moreover, we also initialize the $globalMinCost$ and $sameMinValueCount$ in the Lines 5 and 6, respectively.

Based on the schedules of all the other tasks in the previous iterations, the function *get_new_cost*() calculates the total schedule length obtained for the current task t_i corresponding to the processor choice p_i. The complexity of above mentioned algorithms for an extracted phase graph with n nodes and m edges are shown in Table 4.3.

The above dynamic scheduling heuristic is executed for $maxGameCycles$ number of phase graph repetitions, after phase graph extraction. Among the above iterations, the schedule with minimum total execution time is saved. Until new phase is detected, this minimum schedule obtained from dynamic scheduling is used for subsequent iterations in the current phase. Hence, the proposed dynamic scheduling heuristic does NOT execute for millions of phase graph iterations. Therefore, the benefit of the minimum schedule length obtained within this duration, is reaped for rest of all the subsequent million phase graph iterations.

The processor assignment for the current task is determined by the *get_sch_proc*() function, which is shown in the Pseudocode 4.4. The processor which reduces $minCost$ among all processors with the same type as the current task, is searched. In the Lines 2-9, we iterate through all the processors with a matching type and evaluate the new schedule generated at each of the iterations. In Line 3, the $newSchedule$ variable holds the current task to processor assignment. The *get_new_cost*() function is used to find the cost for this assignment in Line 4. This cost is obtained by calculating the total schedule length of the phase graph for the current $newSchedule$. In the Lines 5 - 8, we keep track of the minimum cost and the corresponding processor assignment using $minCost$ and $minScheduleProcName$. Finally we return the new schedule for task $taskName$ in the Line 10. The original scheduled processor for the task, which is initialized in the Line 1, is returned if any of the processor choices do not reduce the minimum cost.

Table 4.3. Complexity of dynamic phase graph scheduling for a given $currentTask$

Function Name	Complexity	Comment
dynamic_task_schedule()	$O(\{m+n\} \cdot l \cdot r)$	r and l is the number of ready tasks and matching PEs
get_sch_proc()	$O(\{m+n\} \cdot l)$	
get_new_cost()	$O(m+n)$	-
purturb_schedule()	$O(c)$	c is a constant

Algorithm 4.4 $get\_sch\_proc(taskName, inpProcName, *minCost)$

```
1: minScheduleProcName = inpProcName
2: for all procName ∈ matchTypeProcList do
3:    newSchedule = {taskName, procName}
4:    newCost = get_new_cost(prevSchedule, newSchedule)
5:    if (minCost > newCost) then
6:       minCost = newCost
7:       minScheduleProcName = procName
8:    end if
9: end for
10: return minScheduleProcName
```

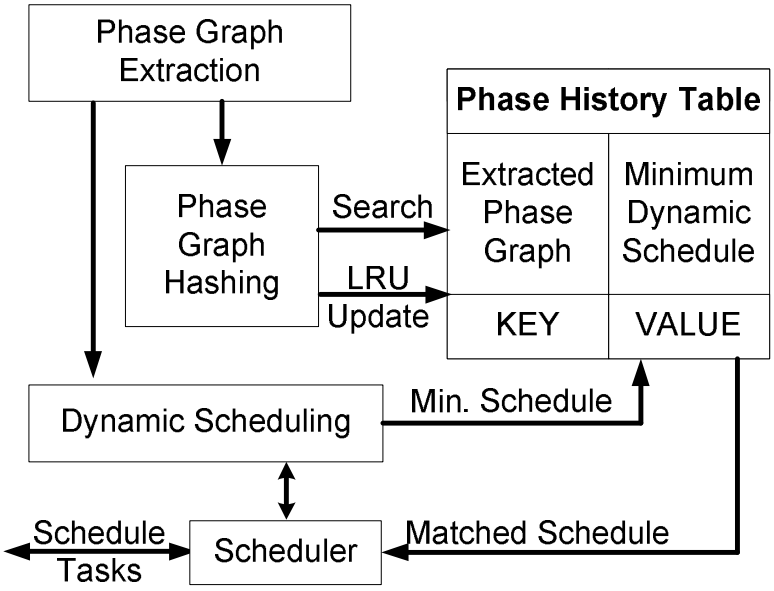


Figure 4.5. Phase graph hashing and search using phase history table

The captured phase graph, at the end of phase graph extraction, is stored in a *Phase History Table* (PHT) as shown in Fig. 4.5.

A string of aliases stored in the *dependencyList* is used to form the hashing function for the extracted phase graph. The minimum schedule obtained after dynamic scheduling is accessed using this stored value which acts as a hashing key. The extracted phase graph is compared with those stored in the PHT, in subsequent phases. If a match is found then, instead of invoking dynamic scheduling, the stored schedule corresponding to the matched phase graph is used. An entry of PHT is replaced

by using least recently used policy, if there is no match. The maximum PHT size is dependent on the available memory at the leader. In the next subsection the dynamic scheduling heuristic is explained with an example.

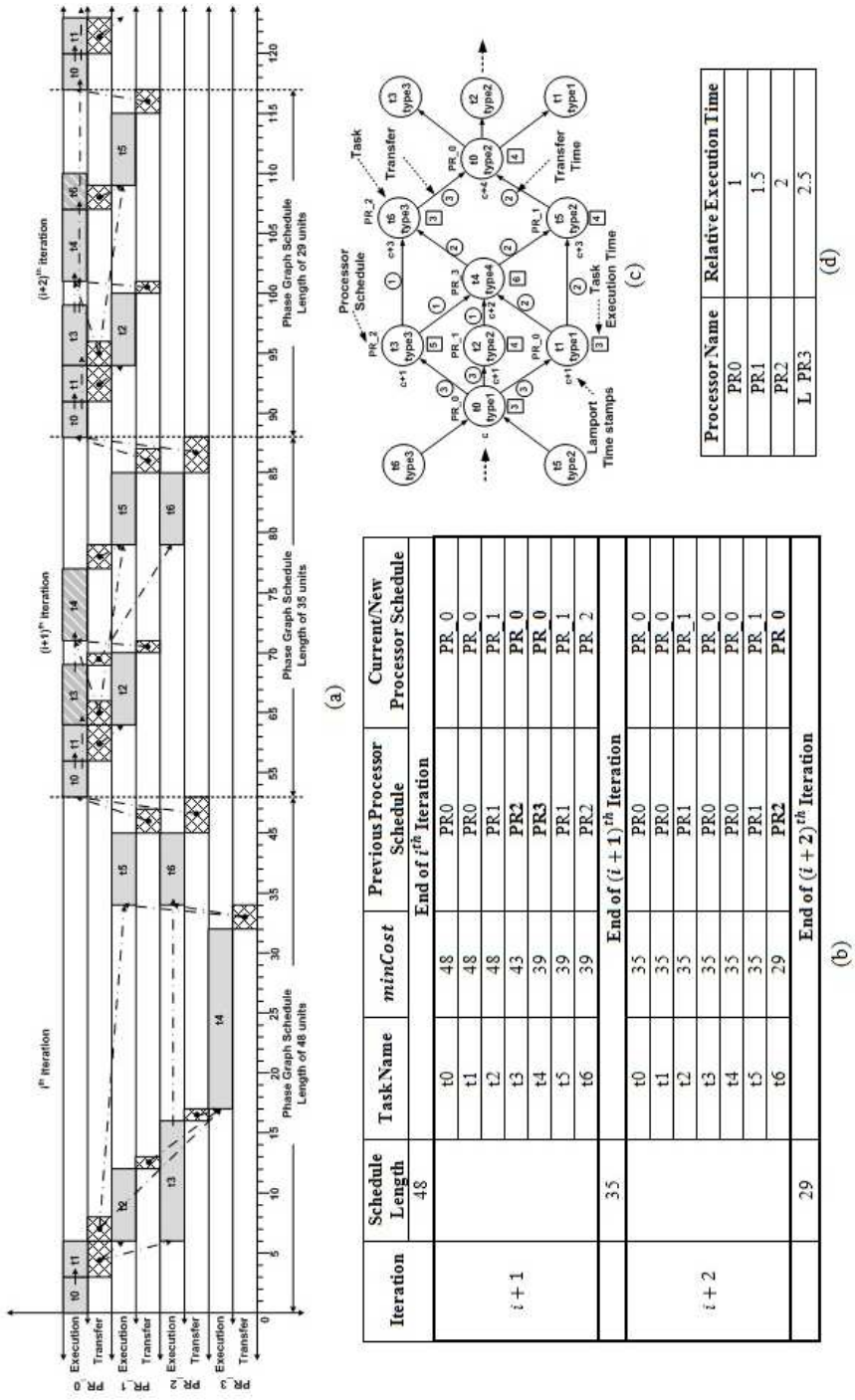
4.7.2 Dynamic Phase Graph Scheduling Example

Consider the phase graph in Fig. 4.3(shown again in the Fig. 4.6-c) executing on a 4 processor MPSoC shown in Fig. 4.1, with processor PR3 being the leader. The square boxes below the task nodes show the nominal execution times for the tasks.

Lets assume for this example that all the tasks and cores have the same execution type. Fig. 4.6-d shows the relative execution times for the PEs. Multiplying the nominal time with the *Relative Execution Time* for the PEs gives the actual execution time. For example, the task $t4$, with an execution time of 6, will take 6 units on $PR0$, 9 on $PR1$, 12 on $PR2$ and 15 on $PR3$.

A gradual decrease in total phase graph execution time with dynamic scheduling on the 4 core MPSoC system, can be seen from Fig. 4.6-a. Task execution and communication is represented by gray boxes and by boxes filled with crossed lines, respectively. Dotted arrows are used to represent communication between various tasks. The tasks which have been rescheduled, are shown by using gray boxes filled with diagonal lines (tasks $t3$ and $t4$ in the $(i+1)^{th}$ iteration). We can see that total phase graph schedule length decreases gradually from 48 to 29 within three phase graph iterations, with dynamic scheduling. The scheduling decisions are made for each of the iterations are shown in Fig. 4.6-b. When the leader receives the end ESW for the task $t0$ corresponding to the $startNode$, $dynamic\_task\_schedule()$ is called. This happens, at the beginning of $(i+1)^{th}$ iteration with the old scheduling length of 48 units for the i^{th} iteration.

In order to calculate $minCost$, the current task is scheduled on various processors and the total execution time for the phase graph is measured. For example, task $t4$



Iteration	Schedule Length	Task Name	*minCost*	Previous Processor Schedule	Current/New Processor Schedule
		End of i^{th} Iteration			
$i+1$	48	t0	48	PR0	PR_0
		t1	48	PR0	PR_0
		t2	48	PR1	PR_1
		t3	43	**PR2**	**PR_0**
		t4	39	**PR3**	**PR_0**
		t5	39	PR1	PR_1
		t6	39	PR2	PR_2
	35	End of $(i+1)^{th}$ Iteration			
$i+2$		t0	35	PR0	PR_0
		t1	35	PR0	PR_0
		t2	35	PR1	PR_1
		t3	35	PR0	PR_0
		t4	35	PR0	PR_0
		t5	35	PR1	PR_1
		t6	29	**PR2**	**PR_0**
	29	End of $(i+2)^{th}$ Iteration			

Processor Name	Relative Execution Time
PR0	1
PR1	1.5
PR2	2
L PR3	2.5

Figure 4.6. Dynamic scheduling for the example phase graph in Fig. 4.3 executing on a 4 core MPSoC (a) Task execution with dynamic scheduling showing the scheduling decisions made from iterations i to $i+2$ (b) Change in *minCost* with every scheduling decisions and the corresponding new schedule for the task (c) The phase graph in Fig. 4.3 shown again for convenience (d) Relative execution times of the PEs

scheduled on processor $PR0$, $PR1$, and $PR3$ will lead to a total execution time of 39, 42, 45 and 48, as the execution time changes from 6, 12, 15 and 18 respectively, keeping the scheduling for other tasks same as that in the i^{th} iteration. Consequently, $minCost$ is assigned to the new minimum execution time and the processor which gives minimum execution time ($PR0$ for execution time of 39) is selected. For the $(i+1)^{th}$ iteration, this is seen in the Line 5 in the Fig. 4.6-b.

Now, the minimum execution time of 47 occurs, for task $t6$ in iteration $(i+1)$, when it is scheduled on processor $PR0$. We stick with the processor $PR2$ originally scheduled for $t6$ in the i^{th} iteration, as this execution time is still greater than $minCost$ value of 39. The total schedule length is reduced to 29 when the task $t6$ is scheduled to the new processor $PR0$ in the next iteration.

Run-time variations in the processor performance, can also be handled by the proposed approach. The leader continues to measure the total execution time of the phase graph using the received start and end ESWs, once the phase graph is scheduled based on the minimum schedule length determined at the end of previous dynamic scheduling. Hence, even though a new phase is not detected, if the leader notices a change in the total schedule length of the executing phase graph, dynamic scheduling will be invoked, once this schedule length increase crosses a certain threshold.

A new schedule with smaller schedule length is obtained by invoking this dynamic scheduling only for $maxGameCycles$. Hence, dynamic scheduling will be automatically invoked if there is a change in execution times of any of the tasks, without any change in the proposed setup.

4.8 Results

In order to generate the following results, simulations were done for a 4x4 MPSoC architecture which uses a fully connected network for communication. Each of the processors can have one of total 6 different processor types. The number of proces-

sors per type is randomly selected between 2 to 4. As shown in Table. 4.4, relative execution times for the processors are generated randomly between 1 and 5. Here we assume that network delays remain constant. The proposed phase graph extraction and dynamic scheduling techniques are generic and independent of the network configuration. A set of 4 random benchmarks were generated using Task Graphs For Free (TGFF) [36] which is a freely available tool that generates pseudo random task graphs based on input parameters. Each benchmark has three different phases, each of which is a randomly generated phase graph.

Table 4.4. Relative execution times and execution types for various processors

Processor Name	Relative Exec. Time	Execution Type
$PR0$	1	$TYPE\_1$
$PR1$	2	$TYPE\_1$
$PR2$	3	$TYPE\_1$
$PR3$	5	$TYPE\_1$
$PR4$	1	$TYPE\_2$
$PR5$	2	$TYPE\_2$
$PR6$	3	$TYPE\_2$
$PR7$	4	$TYPE\_2$
$PR8$	1	$TYPE\_3$
$PR9$	2	$TYPE\_3$
$PR10$	1	$TYPE\_4$
$PR11$	3	$TYPE\_4$
$PR12$	2	$TYPE\_5$
$PR13$	4	$TYPE\_5$
$PR14$	1	$TYPE\_6$
$PR15$	3	$TYPE\_6$

A random selection of the number of nodes in the phase graphs was done between 30 to 55. Moreover, the execution and transfer times of tasks was generated with uniform probability between 80 and 120 units of time. Moreover, each of the nodes has an average input edge count of 3 and output edge count of 2. In order to assign the execution type to the tasks, random numbers are generated for task types between 1 and 6. Moreover, each benchmark has 3 phases repeating for randomly generated

number of iterations. The parameters used for simulation are shown in Table 4.5. The maximum phase graph size of 55 nodes is used to select these values.

Table 4.5. Parameters used for task graph extraction and dynamic scheduling

Comment	Step	Parameter	Value
System Parameters	-	Number of Processors	16
		Number of Processors types	6
Phase Graph Extraction	Phase Detection	Follower Sampling Interval	100
		Follower phase match threshold	3
		Leader phase match threshold	3
	Phase Graph Extraction	$extractionWaitLength$	30
		$windowMatchCountThreshold$	3
Dynamic Scheduling	Scheduling	$minConstantThreshold$	5
	Perturbation	$maxGameCycles$	100
		$taskPurturbThreshold$	0.2

4.8.1 Phase Detection

The simulation results for number of phase graph iterations required for phase detection by the leader are shown in Table 4.6. The benchmark and the corresponding phases are shown in Columns 1 and 2, respectively. The total number of repetitions for each phase which is generated randomly is shown by "Max. Phase Itrs." Column (Column 3). The phase graph size in terms of number of nodes + edges is shown in "PG Size" (Column 4). "Phase Ext. Itrs." (Column 5). shows the number of iterations required for phase detection. Number of iterations required for phase graph extraction as a percentage of the maximum phase graph iterations is given by "% of max. Phase Itrs." (Column 6). It can be seen that, we can accomplish phase detection within less than 160 phase graph iterations.

4.8.2 Phase Graph Extraction

The number of phase graph iterations required for phase graph extraction is shown in Table 4.7. The meaning of the Columns 1-4 in the table are the same as in Table 4.5. The total number of iterations for the phase graph extraction including those taken

Table 4.6. Number of cycles required for phase detection

BM	Phases	Max. Phase Itrs.	PG Size	Phase Ext. Itrs	% of max. Phase Itrs.
BM1	PH1	2512	72	153	6.09
	PH2	3852	72	136	3.53
	PH3	3567	73	153	4.29
BM2	PH1	3694	74	102	2.76
	PH2	3015	77	103	3.42
	PH3	3198	78	136	4.25
BM3	PH1	3582	84	103	2.88
	PH2	3246	92	108	3.33
	PH3	3198	78	136	4.25
BM4	PH1	3852	131	71	3.53
	PH2	3567	135	69	4.29
	PH3	3694	138	75	2.76

for successful phase detection is shown in "PG Ext. Itrs." (Column 5). The number of phase graph iterations at which timing extraction is successfully done is shown in "Timing Ext. Itrs." (Column 6). This value includes the total number of iterations for phase detection, phase graph extraction and timing extraction. The percentage total number of iterations required for the complete phase detection, phase graph extraction and timing extraction, is shown in the last column "% of max. Phase Itrs." (Column 7). It can be seen that, within less than 200 iterations, phase graph extraction is accomplished.

4.8.3 Dynamic Scheduling

Game theory based dynamic scheduling is started with a randomly generated initial schedule, after phase graph extraction. At the end of every phase graph iteration (Game cycle), a new schedule is generated. We apply the dynamic scheduling approach for a total of 100 phase graph iterations. The schedule responsible for minimum duration within 100 $maxGameCycles$, is extracted and used thereafter, in the subsequent iterations. The results for dynamic scheduling and the improvement in the schedule length obtained after phase graph extraction are shown in Table 4.8.

Table 4.7. Number of phase graph iterations for phase graph extraction

BM Name	Phases	Max. Phase Itrs.	PG Size	PG Ext. Itrs	Timing Ext. Itrs	% of max. Phase Itrs.
BM1	PH1	2512	72	162	193	7.68
	PH2	3852	72	143	173	4.49
	PH3	3567	73	160	190	5.33
BM2	PH1	3694	74	109	137	3.71
	PH2	3015	77	109	141	4.68
	PH3	3198	78	143	175	5.47
BM3	PH1	3582	84	109	141	3.94
	PH2	3246	92	114	157	4.84
	PH3	2512	127	74	123	3.96
BM4	PH1	3852	131	77	127	3.61
	PH2	3567	135	77	129	4.47
	PH3	3694	138	84	137	5.39

The meaning of Columns 1-4 are same as in the previous tables. For the randomly generated initial schedule, "Init. Sch." (Column 5) shows the total schedule length (T_{init}) obtained. "Min. Sch." (Column 6) shows the minimum schedule (T_{min}) obtained from game theory Based Dynamic scheduling. "% Init. Sch." (Column 7) shows the percentage improvement in schedule length which is calculated using the following equation.

$$(T_{init} - T_{min}) \cdot \frac{100}{T_{init}}\% \tag{4.2}$$

The total number of iterations required for the end of dynamic scheduling, is shown in "Dyn. Sch. Itr."(Column 8). The iterations required for phase detection, phase graph extraction and dynamic scheduling are included in this. The total number of iterations is given by $maxGameCycles + Phase\ detection\ itrs. + task\ graph\ itrs$, as game theory based dynamic scheduling is run only for 100 iterations. Finally, in (Column 9) this count is shown as the "% max. Phase Itrs.". From the table, we can see that both phase graph extraction and dynamic scheduling is accomplished within less than 300 iterations. This is much less than the millions of iterations a phase

graph goes through. Moreover, the initial schedule length is reduced by 19% to 50%, due to the dynamic scheduling approach.

Table 4.8. Game theory based dynamic scheduling

BM	Ph.	Max. Phase Itrs.	PG Size	Init. Sch.	Min. Sch.	% Impr.	Dyn. Sch. Itrs	% of max. Phase Itrs.
BM1	PH1	2512	72	3874	3469	10.45	293	11.66
	PH2	3852	72	5173	4012	22.44	273	7.09
	PH3	3567	73	4176	2890	30.80	290	8.13
BM2	PH1	3694	74	7258	5126	29.37	237	6.42
	PH2	3015	77	6821	4658	31.71	241	7.99
	PH3	3198	78	4835	3675	23.99	275	8.60
BM3	PH1	3582	84	6668	4995	25.09	241	6.73
	PH2	3246	92	7176	4848	32.44	257	7.92
	PH3	2512	127	11032	7336	33.50	223	7.18
BM4	PH1	3852	131	11535	7296	36.75	227	6.45
	PH2	3567	135	10446	6923	33.73	229	7.93
	PH3	3694	138	11580	7553	34.78	237	9.32

A comparison of the schedule length obtained from dynamic phase graph scheduling procedure, with that obtained from static scheduling done by using ILP based approach (adapted from [105]), Greedy and Random approaches, is shown in Fig. 4.7. An absolute minimum schedule length is obained by solving the ILP formulation for the static phase graph scheduling. A timeout limit of 7200 seconds was used to run ILP for the phase graphs. In the greedy approach, the phase graph is propagated in topological order while scheduling the ready tasks to the fastest processor available. A ready task, in random approach, is schedule by randomly selecting a processor among all the available processors with the matching type. As shown in the Fig. 4.7, the random approach is repeated for 1000 iterations and the minimum schedule length obtained is reported. The total schedule length obtained from the above approaches is compared in the bar chart shown in the figure. The line (right Y axis) shows the phase graph size for various benchmarks in terms of nodes+edges.

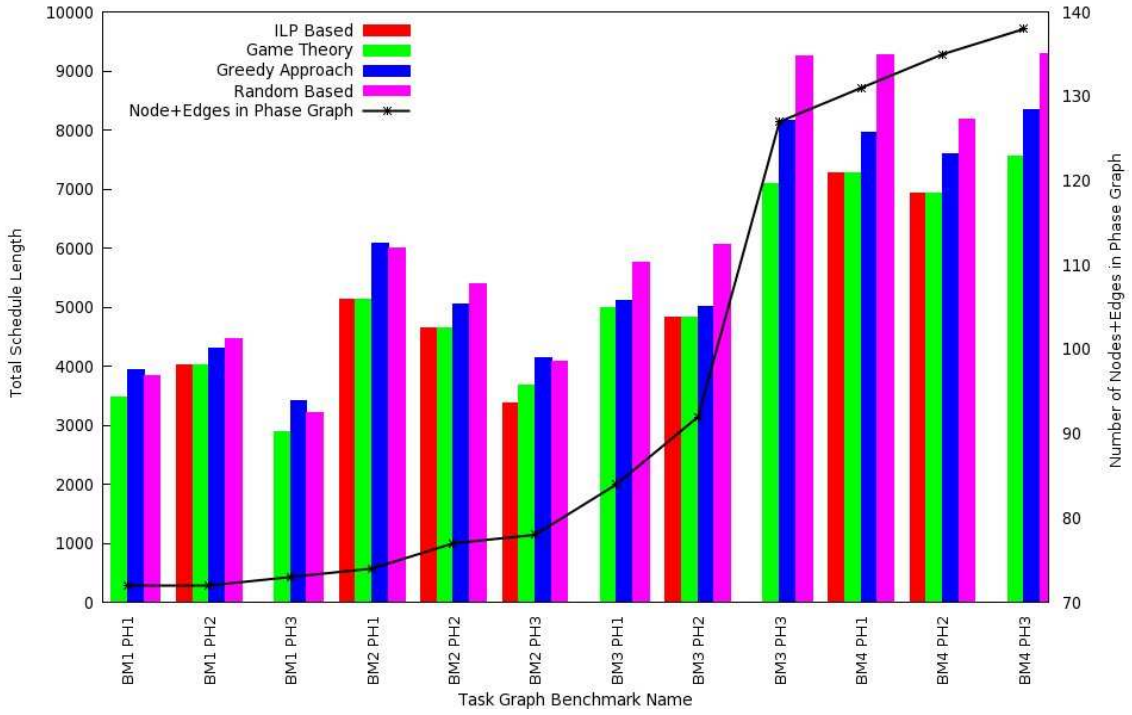


Figure 4.7. Comparison of minimum schedule length obtained from ILP, game theory, greedy and random approaches

It can be seen that the minimum schedule length obtained for most of the phases in the benchmarks, from dynamic schedule, is very close to that obtained from static scheduling done using ILP based absolute minimum. For most of the benchmarks, ILP based scheduling gives a timeout and thus no solution is obtained for these benchmarks as represented by the missing red bars corresponding to the ILP approach.

The Table 4.9 shows the time in seconds required to evaluate the final schedule length. A 1.86 GHz dual core machine with Intel Core 2 CPU and 2MiB cache, was used to run the simulations. It can be seen that ILP (Last Column) does not give any results as it times out for most of the cases (shown by italic times). Moreover, it takes a large amount of time, as compared to game theory based approach, to obtain an optimal result, for BM1 PH2 and BM2 PH3. The proposed game theory based approach produces a solution close to that obtained from ILP for all the benchmarks, as shown in Column 6. The solution is obtained from the game theory based approach

Table 4.9. Time in seconds for various approaches

BM Name	Phases	Task count	Greedy based	Random approach	Game theory approach	ILP based approach
BM1	PH1	72	0.01	150.07	15.25	*7211.35*
	PH2	72	0.02	144.79	13.66	87.93
	PH3	73	0.01	263.25	12.8	*7201.90*
BM2	PH1	74	0.02	103.89	9.1	0.33
	PH2	77	0.01	139.43	11.9	1.00
	PH3	78	0.06	200.29	15.46	127.50
BM3	PH1	84	0.04	225.63	15.91	*7201.51*
	PH2	92	0.07	306.45	22.35	4.34
	PH3	127	0.7	527.68	34.95	*7202.00*
BM4	PH1	131	0.27	492.43	34.74	4.17
	PH2	135	0.06	567.91	33.87	4.10
	PH3	138	0.25	765.87	40.11	*7210.37*

is faster than ILP, in all the cases. Consequently, for large benchmarks, the proposed approach is guaranteed to generate a solution, while the ILP times out for most cases. As compared to other approaches, the random simulation based approach (Column 5) takes much longer time and also gives the worst final schedule time among all the approaches. Morover, it can be seen that the greedy approach (Column 4) is the fastest but is sub-optimal for large phase graphs. Hence, we can conclude that, a good tradeoff between computational time and quality of the final schedule is obtained from the proposed game theory based approach. As compared to the greedy and random heuristics, our dynamic scheduling approach consistently gives a smaller schedule length. Moreover, dynamic scheduling converges much faster (100 iterations) as compared to the random simulation (1000 iterations).

The schedule length variation with game cycles (phase graph iterations) during dynamic scheduling for BM3 PH3 is shown in Fig. 4.8. The randomly generated schedule of with length 11032 is used as a starting point. Next, dynamic scheduling is used where a new schedule is generated at the end of the each iterations. The new schedule generated by random perturbation procedure, is represented using the

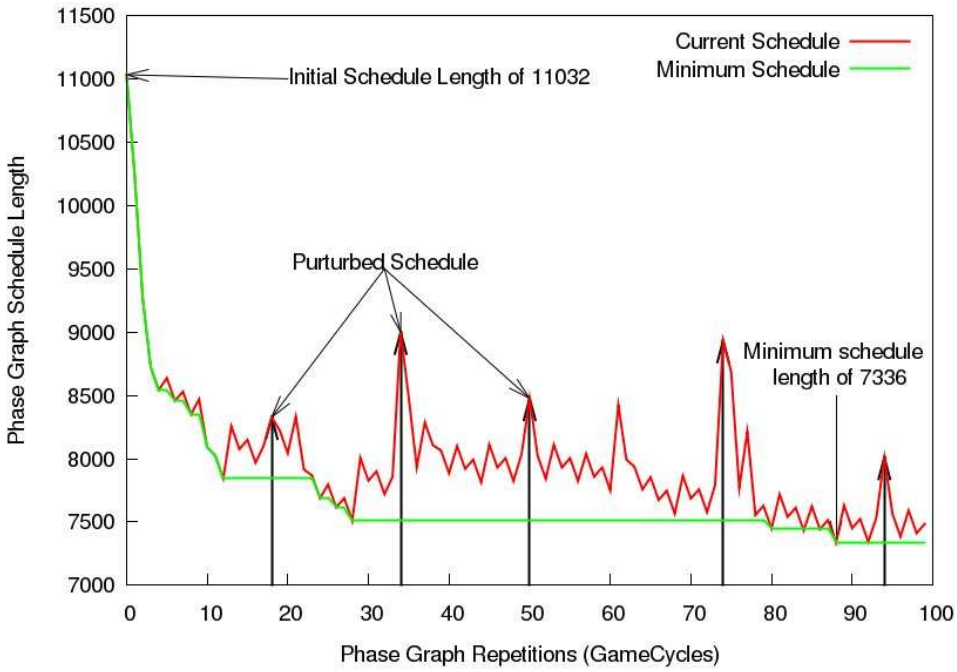


Figure 4.8. Schedule length convergence with game cycles for BM4 PH1

arrows, which is initiated when the minimum schedule length does not change for 5 consecutive phase graph iterations.

Starting from an initial schedule the dynamic scheduling approach converges very fast to a much lower value, as seen from the above figure. The schedule is perturbed in order to extricate the search from the local minima, when the minimum schedule length does not change for more than 5 consecutive phase graph iterations. Perturbation procedure, for most of the cases, increases the schedule length (shown by vertical arrows) to a large value which eventually converges to a much smaller value. Finally a minimum schedule length of 7336 is obtained, at the end of 100 iterations. In the subsequent phase graph iterations,until a new phase is detected, the schedule corresponding to this minimum value is used.

4.9 Conclusions

A novel technique for phase graph extraction, is presented in this work. We first determine if an application is exhibiting a phase behavior using execution history information. This is followed by phase graph extraction with timing information. The results show that our technique extracts a phase graph in fewer than 200 iterations which is several orders of magnitude smaller than the millions of iterations through which a phase graph goes through during the execution of an application. Dynamic scheduling is performed using the extracted phase graph in order to improve performance of subsequent iterations. In order to realize overall gain in performance, such scheduling must be performed very fast. Towards that goal, a novel game theory based dynamic scheduling approach is presented here. Our approach iteratively improves on the previous schedule to obtain a smaller schedule length, starting from an initial non optimal schedule. Typically in fewer than 100 iterations, the approach also reduces the initial schedule length by 19% to 50%. We compare our approach with ILP based static scheduling approach adapted from [105]. It is observed that, solutions comparable to ILP based static scheduling are obtained from the game theory based dynamic scheduling approach, albeit much faster. Moreover, due to the simplicity of the heuristic, we can easily integrate the proposed solution into an MPSoC with limited resources.

CHAPTER 5

THERMAL AWARE TASK GRAPH SCHEDULING

With device scaling, there has been a substantial increase in power density in the deep sub-micron era. This has lead to the formation of regions with very high temperatures known as thermal *Hotspots*. These thermal effects not only lead to a degradation in reliability and performance, but also a significant increase in the total cooling cost and worsening of non-ideal effects like leakage. It has been estimated that increasing the power dissipation above 20 to 30 Watts increases the cooling cost by more than $\$1/W$ [104].

High hotspot temperatures, if unchecked, will lead to reliability related failures [72] including higher rate of electro-migration and mechanical failures due to differential thermal expansion of various regions of the IC. Moreover, increase in gate delay and wire delay with temperature leads to performance degradation. It has been observed in [12] that a 20C increase in temperature leads to 5% to 6% increase in the Elmore delay of the circuit. Global signals like clock are most affected because of the variation in clock skew with temperature [12]. Spatial thermal gradients across various regions of a chip can be as large as 30C to 50C [72]. Spatial thermal gradients are responsible for mechanical failures due to different dielectric expansion in various regions of the IC. Temporal thermal gradients are generated by change in the workload power dissipation with time. These fast temperature changes of large magnitude lead to fatigue failure and deformation of the package [4]. Therefore, *Dynamic Thermal Management* (DTM) [16] is used in which the processor is run on a reduced voltage/frequency when the measured temperature increases above the DTM

threshold. This approach is reactive in nature and leads to performance reduction for tasks executing on the processor.

Based on the above observation, it can be seen that it is important to reduce the worst case temperatures. This work deals with run-time task graph scheduling for MPSoCs and aims at preventing temperature increase by dynamically predicting the temperature variation for various task schedules and performing task migration in order to minimize the total schedule length.

The rest of the work is organized as follows: We explore the previous work in the Section 5.1. Next, we present the preliminaries of wavelet transforms in Section 5.2. Section 5.3 gives an overview of the proposed approach. Then, each of the proposed steps are explained in Section 5.4 and Section 5.5. Section 5.6 presents the ILP formulation for temperature aware scheduling. Then results are presented in Section 5.7. Finally we conclude in Section 5.8.

5.1 Related Work

Various techniques for thermal aware scheduling and thread migration are presented in literature. Some of these techniques are proactive in nature [60][77][29][22], where temperature prediction is done to perform thread migration or DVFS, while others [32][31][85] are reactive in nature. Reactive techniques leads to faster thermal gradients, higher temperatures and performance loss. Most of the proactive techniques are more efficient but have much higher hardware/software cost.

Thermal aware thread migration for heterogeneous multiprocessors has been done by Khan *et al.* in [60] using an intermediate operating system layer known as Micro-visor. It maintains several data structures for thermal management and predicts the thermal mapping for the next epoch of computation based on the thread specific thermal demands. When micro-visor predicts a thermal hotspot, a preemptive thread migration is done. This technique uses very simple models for temperature prediction

where the future temperature is linearly extrapolated based on the current and the past temperature increase.

Coskun *et al.* in [32][31] present temperature aware task scheduling for MPSoCs. In [32] an ILP based thermal aware static scheduling approach is presented for various objectives of minimizing thermal hotspot temperatures, balancing thermal hotspots and energy consumption, and minimizing total energy. The solution of ILP formulation provides the required voltage and frequency settings for DVFS which is used for thermal management. The ILP based static scheduling approach is extended to static-dynamic policy [31] which is applied during run-time for the cases where the workload deviates from the statically estimated task graph. The task graphs considered in this approach are very small. Its a known fact that ILP does not scale very well for scheduling medium sized task graphs.

Dick *et al.* in [22] do ILP based temperature aware scheduling for hard real time deadlines. A single ILP formulation is done not only to represent the task graph dependency constraints but also for the package thermal constraints. As a steady state thermal model is used for the ILP formulation, they do not consider the effect of transient temperature variation. They also present a dynamic task scheduling technique based on temperature prediction. Static slack estimation is done for tasks and each ready task is scheduled to the processor which minimizes slack while preventing a thermal emergency. This approach is greedy in nature and hence is non optimal, as it does not search the available solution space. If a ready task cannot be scheduled without preventing thermal emergencies, delay insertion is done where a binary search based technique is used for calculation of task delay which will prevent thermal emergency. This technique is computationally expensive as it involves repeated delay insertion and thermal evaluation until the minimum delay preventing a thermal emergency is found.

Puschini *et al.* in [85] [86] present a non-cooperative game theory based approach for temperature aware run-time frequency assignment for processors while maintaining synchronization between the tasks. Here, each of the processors are players and they individually choose their actions independently so as to optimize their local set of goals. Following this strategy eventually leads to a global optimization. Temperature and synchronization are used as matrices to build the preference functions of processors. As their technique is reactive in nature, there is a lag between the temperature measurement and the corresponding action. In addition to this, they use a simplified thermal model to perform thermal simulation, which does not take transient temperature into account.

Murli *et al.* in [77] use convex optimization based method for temperature control. This technique is proactive in nature and controls core temperature while satisfying performance constraints of applications. Dynamic frequency scaling is used for thermal management where the core frequency assignments are made so that the maximum temperature does not go above the threshold until the next frequency assignment is done. It consists of an offline phase where thermal analysis is done for various frequencies and starting temperatures, and a convex optimization problem is solved to obtain a table of frequency assignments. In the on-line phase, the above generated table of frequency assignments is used to periodically perform DFS to prevent thermal emergencies. The offiline phase in this technique is compute intensive, as it requires the solution of convex optimization problem.

Reda *et al.* [29] propose a thermal prediction method based on program phase identification. They have an offline phase where the performance counter measurements are used to perform principle component analysis to capture the phase information for the application. This is followed by use of k-min clustering to obtain the global phase locations in the observed measurements. The above data is used to train a state-space model which captures the relation-ship between temperature,

performance counters and operating frequencies. At run-time, phase identification is done and the learned state-space model is used in conjunction with the temperature reading from thermal sensors to predict the future temperature and initiate DVFS based on the temperature prediction. The size of the tables generated from the offline phase can be very large. Moreover the offline phase is application dependent and time consuming as it involves solution to k-min clustering which is $NP - Hard$.

Ayoub *et al.* present a predictive thread migration for thermal management [14] in multicore processors. They use the band limited property of temperature variation to perform temperature prediction based on the previously measured temperatures. Temperature spectral limited bandwidth is used to estimate the predictor coefficients during design time. The temperature prediction in conjunction with workload characterization is used by the *Operating System* (OS) scheduler to perform thermal aware task scheduling. The OS scheduler only looks at the current set of ready tasks and does not perform look-ahead, as a result the final schedule obtained is not optimal.

Thus it can be seen that most thermal aware task scheduling algorithms are greedy or perform dynamic thermal management using DVFS. While DVFS causes performance degradation and as will be seen in the results, task migration becomes more effective with increase in the number of cores in an MPSoC. Moreover, thermal aware task migration is usually done in a greedy fashion which leads to a suboptimal solution. In addition to dynamic scheduling, the static solutions presented in the above techniques do not take transient thermal behavior into account. Moreover, in case of predictive techniques, offline pre-characterization required for temperature prediction step is compute intensive as it requires the solution of some optimization problem.

In this work we present (i) a run-time temperature prediction technique (ii) a run-time look-ahead based branch-and-bound scheduling heuristic which thermally evaluates various schedules using directed search to minimize the total schedule length (iii) finally a delay insertion technique to remove task execution overlaps leading to

thermal emergencies. The above solutions are leveraged by a wavelet based thermal modeling approach which does an application independent pre-characterization of the thermal system.

Ferzli *et al.* [39] have proposed a wavelet based technique to obtain a stimulus that causes worst case voltage droop at a given node in a power supply network. Current fluctuations generated by switching of logic gates are represented using wavelets. This is used to determine the worst case stimulus by using an ILP formulation. As underlying logical dependencies are not considered in this approach, the traces generated here are not functional.

Based on the above work Srinivasan and Ganeshpure [91] proposed a wavelet based thermal modeling approach to generate an input workload which maximizes the hotspot temperature at the target location. This approach is based on the fact that an application goes through stable phases during which its power dissipation remains constant. This thermal modeling approach is used to characterize the response of chip/heat sink thermal system to generate a set of temperature response values. These pre-characterized values are used by an ILP based technique to generate a combination of extracted workload phases which maximizes temperature at the target location. Based on this work, we use wavelet based technique for characterizing the thermal behavior of the system and for temperature prediction to enable task migration. In the next section we provide preliminary introduction to wavelet transforms.

5.2 Preliminaries

Wavelet transforms are used to represent the frequency components of an arbitrary waveform using a set of time limited functions known as wavelets. A wavelet can be scaled in order to increase its amplitude or shifted in time. These wavelets form a set of orthonormal basis functions for which the shift and scale values (wavelet coefficients) are obtained by using wavelet transform. The number of wavelet coefficients required

to represent the waveform depends on type of the wavelet and the characteristics of the input waveform. A large transform value is obtained at a particular shift if the shape of the wavelet matches with that of the input waveform at that particular time shift. In our work, *Haar* [10] wavelets are used to represent power dissipation at various processors. This is based on the assumption that the power dissipation of a task remains constant during its execution. A Haar wavelet $\Psi_{k\delta}(t)$ is a pulse defined by a pulse width parameter 'δ', and a time shift 'k' as shown in Fig. 5.1. This pulse varies in time from $k \cdot \delta$ to $(k+1) \cdot \delta$ with an amplitude of $1/\sqrt{\delta}$. The above technique is adapted from work by Srinivasan and Ganeshpure [91] which deals with generation of an input workload so as to maximize temperature at target location.

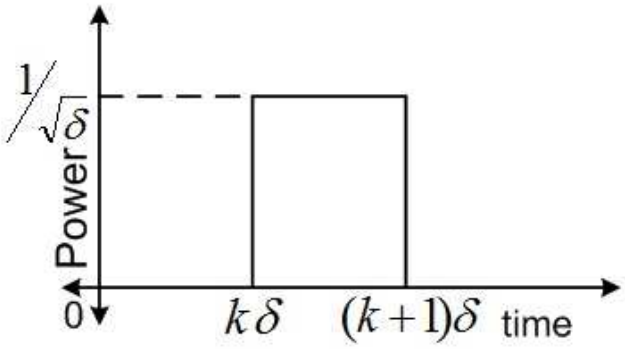


Figure 5.1. A Haar power wavelet with a scale 'δ' and a time shift '$k \cdot \delta$' [91]

$$\Psi_{k\delta}(t) = \frac{1}{\sqrt{\delta}} \cdot \psi\left(\frac{1 - k \cdot \delta}{\delta}\right) \tag{5.1}$$

It will be explained in the subsequent sections, how the power waveform is represented using a linear combination of shifted and scaled Haar power wavelets. A Haar wavelet which has 1W of power and duration δ is called a *Unit Power Wavelet*. The next section gives an overview of the proposed approach.

5.3 Proposed Approach Overview

Here we propose a greedy branch-and-bound search based heuristic for thermal aware scheduling. Our approach consists of a (i) pre-characterization stage, where the thermal behavior of the system is characterized using wavelet analysis and the power dissipation information is determined for the tasks in the task graph, and (ii) run-time stage where dynamic task migration is initiated to mitigate thermal emergencies. The above steps shown in the Fig. 5.2 are explained in brief below.

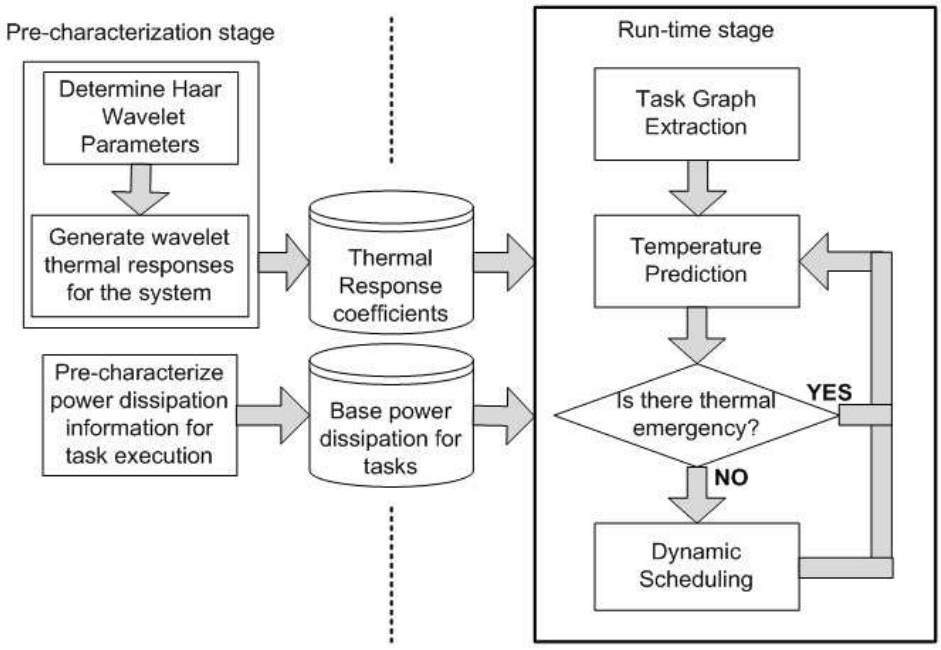


Figure 5.2. Pre-characterization and run-time stages in the proposed dynamic thermal aware task migration technique [91]

- *Pre-characterization:* This stage involves characterizing the thermal response of the processor die/package system using wavelet transforms. We use Haar wavelet to represent power dissipated at various *Processing Elements* (PEs). It should be noted that, we will interchangeably use the word "processor" and the acronym "PE" to represent a processor core in an MPSoC. The MPSoC is characterized by applying a Haar wavelet pulse of unit power value at each of the PEs one at

a time and obtaining the corresponding temperature variation at all the other PEs. This temperature response at various PEs for the current power source corresponding to the application of the power wavelet is stored in a *Thermal Response Table*, as shown in the central part of Fig. 5.2. In addition to this, the power dissipation of all the tasks in the task graph is also pre-characterized on various PEs using simulation and signal activity measurement. As a result, we obtain a *Power Estimation Table*, which holds the power dissipation estimates obtained for tasks scheduled on various processors. Therefore, at the end of pre-characterization we obtain (i) the thermal response table modeling the thermal behavior of the system and (ii) power estimation table for the task execution on various processors, generated from simulation. These are used in the run-time stage to perform dynamic thermal aware task scheduling.

- *Run-time:* In this step, we perform dynamic scheduling based on temperature prediction by using the thermal response tables generated in the pre-characterization stage. In our approach, dynamic scheduling is initiated only when a thermal emergency is predicted in the originally scheduled task graph. The search for a schedule continues until either a minimum schedule which prevents thermal emergencies is found, or if it fails to get such a schedule. In case of failure, delay insertion is initiated as a fallback action to eliminate thermal emergencies. Consequently, the run-time stage consists of the following steps:

  - *Temperature Prediction:* In this step, we predict the temperature variation of the executing task graph by determining the set of tasks which are deemed to be executed in the future known as *Future Tasks*. A set of L future tasks is extracted from the task graph and inserted into the *Look-ahead List*. The power values for these tasks estimated in the pre-characterization stage are used for temperature prediction.

The temperature profile at each of the PEs is represented using a linear combination of scaled and shifted versions of the thermal responses stored in the thermal response table. This future temperature profile at the end of execution of each of the L future tasks is generated at regular intervals to check for thermal emergencies. When a thermal emergency is predicted, dynamic task migration is initiated.

- *Task Migration:* A branch-and-bound heuristic is used for dynamic task scheduling so as to eliminate thermal emergencies while minimizing the total schedule length. The tasks in the look-ahead list are arranged in topological order and the branch-and-bound heuristic schedules each of these tasks onto the fastest processor which does not lead to a thermal emergency. If a solution is found for all the tasks in the look-ahead list or if none of the processor choices prevent thermal emergencies, then back tracking is done and branch-and-bound continues searching for an alternate schedule. The fact that branch-and-bound does not stop, even after a valid schedule is generated, prevents the search from being stuck in local minimum.

- *Delay Insertion:* In the case if branch-and-bound based task migration fails to find a valid schedule eliminating thermal emergencies, delay insertion is initiated in order to reduce temperature by removing task overlap. The thermal response table is used to determine the delay required to eliminate the temperature effect of one task on the other.

In the next sections we will explain each of the above steps in more detail:

5.4 Pre-characterization Stage

In this step, the thermal response of the system is characterized using wavelet transforms. This is done by applying a unit power wavelet to various PEs on the MPSoC and measuring the temperature at every other PE. This process is repeated for all PEs one by one to generate a thermal responses table. The response at time t on PE p_{j_2} due to a unit power wavelet applied at the PE p_{j_1} at time τ is denoted by $\Phi_{j_1 j_2}(t-\tau)$. Fig. 5.3 shows the thermal response for a unit power wavelet applied to a particular PE. The simulation time is divided into discrete time intervals which are integral multiples of the width of the unit power wavelet 'δ'. Consequently, the temperature response is also sampled at same discrete time intervals. The total number of samples taken for the temperature response is known as *Thermal Response Length*. For a thermal response length of W samples, the total duration of the thermal response is given by $W \cdot \delta$ and is measured from the start time of the application of unit power wavelet. The following Table 5.1 shows the thermal response table for an MPSoC with three processors.

Table 5.1. Thermal response table for an MPSoC with three PEs

Unit power wavelet applied at ..	Thermal response measured at ..	Thermal response values $t \in W$
$PR0$	$PR0$	$\phi_{00}(t)$
	$PR1$	$\phi_{01}(t)$
	$PR2$	$\phi_{02}(t)$
$PR1$	$PR0$	$\phi_{10}(t)$
	$PR1$	$\phi_{11}(t)$
	$PR2$	$\phi_{12}(t)$
$PR2$	$PR0$	$\phi_{20}(t)$
	$PR1$	$\phi_{21}(t)$
	$PR2$	$\phi_{22}(t)$

For an MPSoC with P PEs and a thermal response length of W sampled temperature values, the number of elements in the thermal response table is $O(P^2 \cdot W)$. The duration of unit power wavelet determines the size of thermal response table.

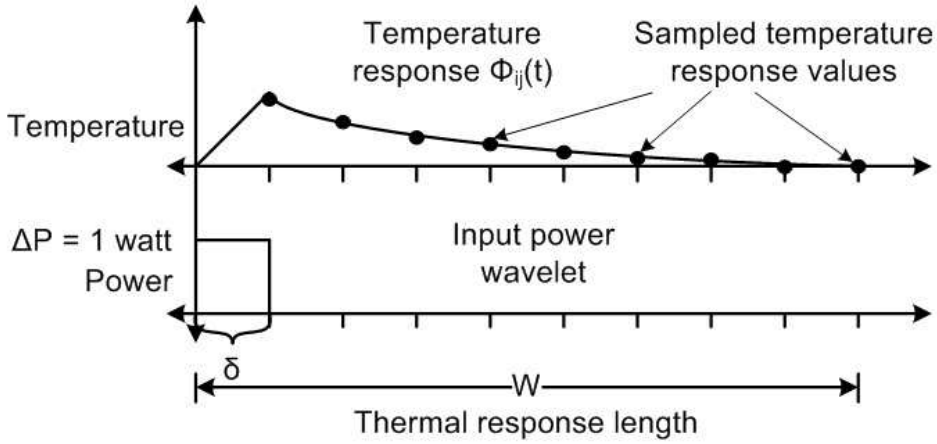


Figure 5.3. Input power wavelet and corresponding temperature response values

For each thermal response, the number of samples W increases linearly with the reduction in the power wavelet duration δ and so does the size of the thermal response table. Consequently, higher the number of samples taken, the more accurate is the temperature prediction. Moreover, for the same duration of the unit power wavelet, a larger W improves the accuracy of temperature prediction, but also increases its time complexity. In the subsequent text, the thermal response for a unit power wavelet as seen in Fig. 5.3 will be graphically represented by using a triangle.

In addition to characterizing the thermal behavior of the die package system, we also estimate the power dissipated for each of the tasks on various PEs of the MPSoC. This can be easily done by statically extracting the task graph for the application (Vallerio *et al.* [106]), followed by using an architecture simulator like SESC [82] to execute the task and finally measuring the power dissipated using a power model like Cacti [98]. This generates the power estimation table, which consists of a list of power estimates, dissipated by a task on the corresponding set of PEs with matching execution types.

Next, we explain the run-time stage, which uses the greedy branch and-bound heuristic for dynamic thermal aware task migration utilizing the thermal response and the power estimation tables, extracted in the present step, for temperature prediction.

5.5 Run-time Stage

Task scheduling for minimizing schedule length is an known $NP - Hard$ problem [105] [62]. The proposed branch-and-bound heuristic is based on a combination of look-ahead, task scheduling and temperature prediction to evaluate each of the scheduling decisions for the presence of thermal emergencies. The thermal response and power estimation tables generated during pre-characterization stage are used for temperature prediction. All the aspects of the proposed branch-and-bound heuristic are explained in detail in subsequent sections starting from temperature prediction.

5.5.1 Temperature Prediction

Our temperature prediction approach is based on the *Linear Time Invariance* [84] property of the thermal system.

Definition 5.5.1. A system with a transfer function h and a response given by $y(t) = h\left(x\left(t\right)\right)$ is linear time invariant if and only if any time shift (or scaling) of the input $x(t)$ leads to the equivalent shift (or scaling) of the output $y(t)$. Therefore, the properties given by the following set of equations are valid.

$$\text{Linearity Property:} \qquad y\left(t\right) = h\left(S \cdot x\left(t\right)\right) \tag{5.2}$$

$$\text{Time Invariance Property:} \qquad y\left(t-\tau\right) = h\left(x\left(t-\tau\right)\right) \tag{5.3}$$

Based on the known duality between heat flow and electric current, the thermal system consisting of the package and die can be represented by an RC network [52]. According to this duality, heat flow is analogous to flow of current and temperature is

analogous to node voltage. Moreover, thermal properties of the material, like thermal conductance and heat capacity, are analogous to electrical resistance and capacitance respectively. Consequently, the thermal system consisting of a die and heat sink can be represented using a finite element thermal *RC* network, where the input is a set of current sources representing the power dissipated at various units and the outputs are node voltages corresponding to the temperature at various locations. This *Thermal RC Network* can be modeled as a low pass filter which has a linear time invariance property [91][84]. Hence, according to the Definition 5.5.1, the temperature response of a time shifted (or scaled) unit power wavelet can be obtained by doing an equal amount time shifting (or scaling) of the original thermal response. We utilize this property for temperature prediction by representing the input power trace as a linear combination of shifted and scaled power wavelets.

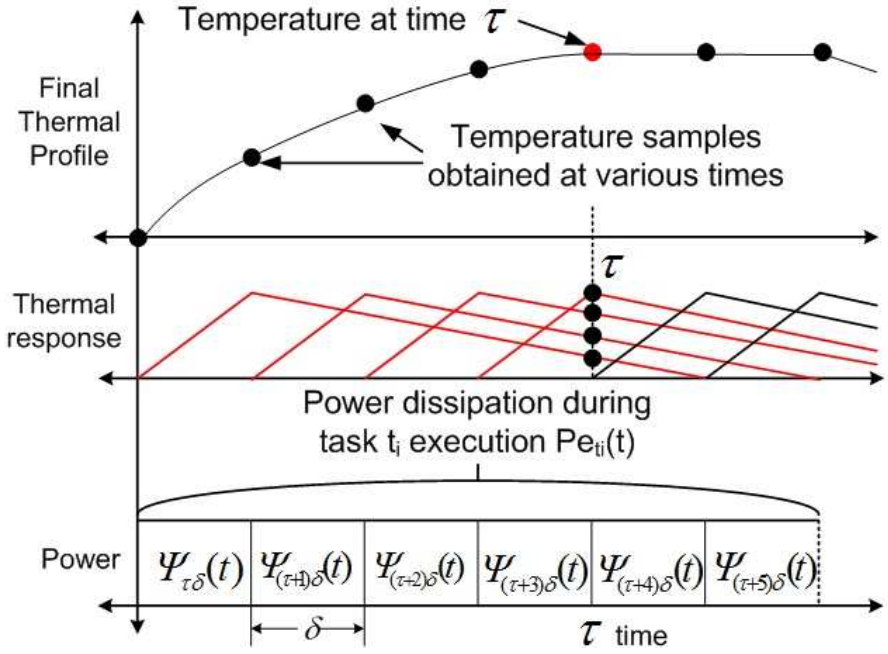


Figure 5.4. Representation of power dissipation profile and the corresponding temperature variation for a task t_i by using shifted and scaled unit power wavelets

Fig. 5.4 shows how the power dissipation profile of a task t_i can be represented as a linear combination of the set of shifted and scaled unit power wavelets $\Psi_{\tau\delta}(t)$. It

can also be seen that, time is discretized into integral multiples of unit power wavelet duration δ. The wavelets are scaled by the power values ρ_{t_i} corresponding to the estimate of the power dissipated by task t_i obtained from the power estimation table. Corresponding to each of the above shifted and scaled unit power wavelets $\Psi_{(k+\kappa)\delta}(t)$, there is a set of shifted and scaled thermal response values $\Phi_{j_1 j_2}(t-\tau)$ shown in the middle part of Fig. 5.4. The temperature at a given time τ is equal to the sum of all the thermal response curves with a non-zero temperature value at τ. This is shown by the rectangles in red in the figure. The corresponding temperature values are obtained at integral multiples of δ and, at the time τ, the temperature is given by the red dot.

Temperature prediction is done by determining the temperature at the end of every task execution using thermal response values. This is based on the observation that temperature is highest and hence, thermal emergencies can only occur, at the end time of a task execution. Firstly, we will explain how temperature measurement is done for a pair of tasks, then we will extend the approach for the more generic case consisting of multiple tasks.

5.5.1.1 For a pair of tasks

The thermal emergency check has to be done for each of the tasks in the task graph. A task which is currently being evaluated for thermal emergency is called the *Target Task*. For a given target task in the task graph, the target time is defined as the time at which temperature has to be determined. This time is equal to the end of execution time for the target task under consideration.

Definition 5.5.2. A task is said to be *Thermally Overlapping* at the target time, if the thermal response of the task has a non-zero value at a given *Target Time*. A task t_i with start and end times given by st_i and et_i, respectively, is thermally overlapping at target time tr if the following conditions are satisfied.

$$(tr - et_i) \leq W \tag{5.4}$$

$$(tr - st_i) > 0 \tag{5.5}$$

The temperature at target time is calculated by using a linear combination of the thermal responses of all thermally overlapping tasks, scaled by their power dissipation.

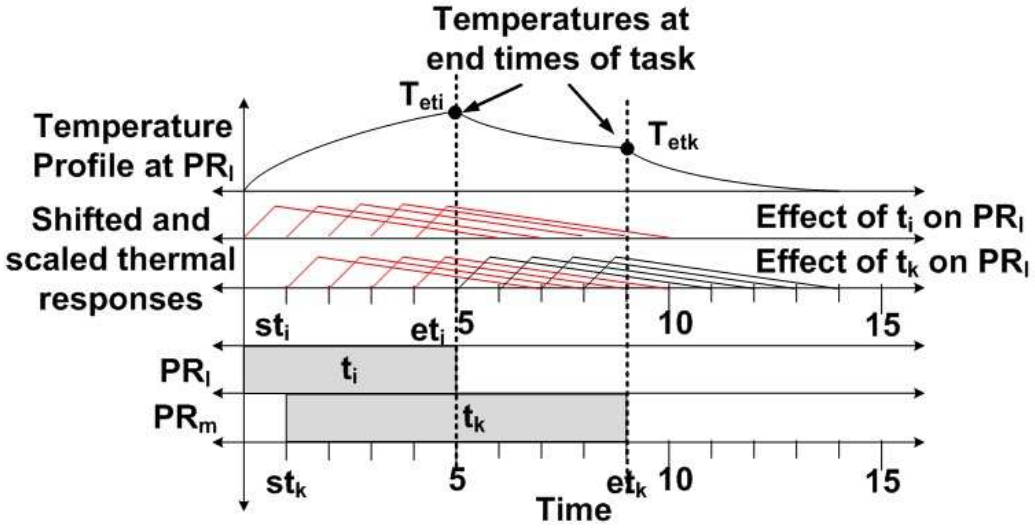


Figure 5.5. Temperature prediction for a pair of tasks

Consider a pair of tasks t_i and t_k executing on the PEs PR_l and PR_m, and dissipating power ρ_{il} and ρ_{km}, respectively. This is shown at the bottom two axes of the Fig. 5.5. Tasks t_i and t_k have start and end times of $\{st_i, et_i\}$ and $\{st_k, et_k\}$, respectively. Our goal here is to determine the temperature for the target task t_i at the corresponding target time et_i. The power dissipation profile of the tasks is decomposed into a set of shifted and scaled power wavelets. The triangles above the task execution are the thermal responses corresponding to various unit power wavelet shifts. It can be seen that tasks t_i and t_k are thermally overlapping at the target time of et_i. The thermal responses represented in red are the set of responses with non-zero value at et_i. The temperature $T^l(et_i)$ at the PE PR_l executing task t_i at target time et_i is the sum of the scaled thermal responses $T_i^l(et_i)$ corresponding to

the task t_i and that of the response $T_k^l\,(et_i)$ corresponding to the effect of task t_k at the target time et_i. This is given by the following set of equations.

$$T_i^l\,(et_i) = \sum_{\tau=et_i-W}^{et_i} (\rho_{il} \cdot \phi_{ll}\,(et_i - \tau) \cdot \alpha_{ii}(\tau)) \tag{5.6}$$

$$T_k^l\,(et_i) = \sum_{\tau=et_i-W}^{et_i} (\rho_{km} \cdot \phi_{lm}\,(et_i - \tau) \cdot \alpha_{ik}(\tau)) \tag{5.7}$$

$$T^l\,(et_i) = T_i^l\,(et_i) + T_k^l\,(et_i) \tag{5.8}$$

where:

$$\alpha_{ii}\,(et_i) = TRUE \;: \tau \geq st_i \tag{5.9}$$

$$\alpha_{ik}\,(et_i) = TRUE \;: \tau \geq st_k \;\&\; \tau < et_k \tag{5.10}$$

In the above, Equation 5.6 determines the thermal effect of task t_i at the target time et_i while Equation 5.7 determines that for task t_k, respectively. In Equation 5.6 we use the thermal response parameters ϕ_{ll} which represents the thermal response of a unit power wavelet applied at processor PR_l on itself, while ϕ_{lm} is the thermal response on PR_l due to a unit power wavelet applied on PR_m. We use a set of Boolean variables $\alpha_{ki}(\tau)(\alpha_{ii}(\tau))$ which are $TRUE$, only for the set of thermal response shifts falling within the execution time duration of the tasks $t_k(t_i)$ before the target time et_i. This is shown by constrains in the Equations 5.9 and 5.10. The total temperature is the sum of the above two temperature effects and is given by Equation 5.8. Moreover, we can also determine the thermal response profile by solving the Equations 5.6 to 5.10, at various target times corresponding to discrete time shifts. The top most curve of the Fig. 5.5 shows the temperature profile at PR_l. It can be seen that the temperature, at first rises to a much higher value, at the end time of task t_i, due to a combined effect of the power dissipated on tasks t_k and t_i. This finally stabilizes to a lower value at end of task t_k because of the effect of task t_k only. Similarly, we can

calculate the temperature profile at the processor PR_m considering the effect of both tasks t_i and t_k.

5.5.1.2 Temperature Prediction for a set of N Tasks

Consider a set of N thermally overlapping tasks which are going to be executed on an MPSoC consisting of P processors. Let t_i be the target task executing on the processor PR_l and the corresponding target time is et_i. In order to check if there is a thermal emergency at the target time, we need to determine the temperature at all the processors $T^l(et_i)$. This is done by writing the Equations 5.6 to 5.10 for each of the N thermally overlapping tasks, to generate a set of temperature parameters $T_k^l(et_i)$. Consequently, the temperature at the processor PR_l at target time et_i is given by the following equation.

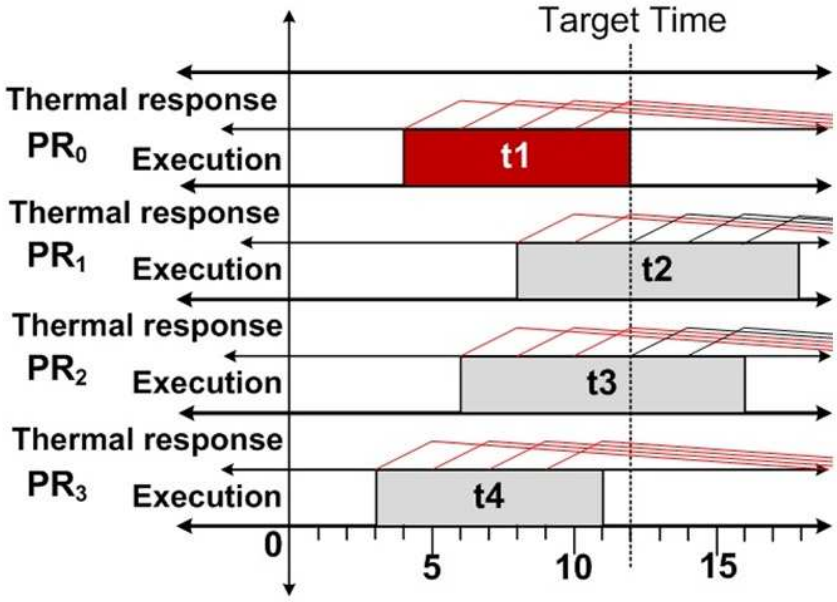


Figure 5.6. Thermal emergency check at end time of t_1 for a set of thermally overlapping tasks

$$T^l(et_i) = \sum_{k \in N} T_k^l(et_i) \qquad \forall l \in P \tag{5.11}$$

Fig. 5.6 shows the temperature prediction for a set of four thermally overlapping tasks executing on an MPSoC consisting of four processor cores. The current target time is the end time of task t_1. At the target time, we determine the temperature at all the PEs by using the Equations 5.6 to 5.10. Once this is done, the target time is shifted to the end time of task t_3 and the above procedure is repeated. Similarly, the above process is also repeated for the end times of tasks t_2 and t_4.

If the temperature corresponding to any of the tasks is found to be greater than the thermal threshold then a thermal emergency is predicted. Next, we explain the proposed branch-and-bound approach for thermal aware task scheduling.

5.5.2 Dynamic Thermal Aware Task Migration

Here we propose a look-ahead based branch-and-bound heuristic. The proposed heuristic is predictive in nature and consists of the following steps.

5.5.2.1 Look-ahead

At the beginning of execution of a new task, a look-ahead list of L tasks, which will be executed in the future, is generated. The size of look-ahead list is equal to the look-ahead length parameter L which determines the complexity of the search. This look-ahead list is arranged topologically based on the *level* of a task.

Definition 5.5.3. The level is a number associated with a task which is equal to the longest path from the task to any one of the nodes in the look-ahead list for which all the input dependencies have already been resolved.

The set of tasks with same level are arranged in the descending order of the power estimation obtained from the pre-characterization phase. Consequently, scheduling decisions are made in level order. For the tasks in the same level, scheduling decisions are first made for the tasks with higher power dissipation before considering those

with lower power dissipation. This task list is supplied to the branch-and bound heuristic as explained next.

5.5.2.2 Branch-and-bound

Determination of task to processor assignment which minimizes total schedule length while preventing thermal emergencies is an $NP - Hard$ problem. Dynamic scheduling involves solving this problem at run-time. Given the limited resources of the processors available on the MPSoC, it is infeasible to solve this problem in an exact fashion, within a reasonable amount of time. Various look-ahead based greedy heuristics have been proposed for the solution of this problem in the previous literature. These solutions are highly suboptimal, as they only look at the current set of ready tasks to determine the schedule. Moreover, these algorithms have a tendency of being stuck in local minimum. This is because, once a solution is found; they do not continue searching for an alternate improved solution. Here we present a look-ahead based branch-and-bound heuristic which comes out of the local minimum by continuing search even after a valid solution is found. In order to reduce the size of the search space the branch-and-bound operates only for the reduced set of tasks in the look-ahead list with length L.

The branch-and-bound heuristic consist of a *Decision List* which holds the set of tasks appearing in the look-ahead list. For each element of the decision list, there is a list of processors of the matching type on which the particular task can be scheduled. The decision list generated for a look-ahead of four tasks is shown in Fig. 5.7. There is a pointer variable called $currentTaskPtr$ which points to the current task for which a decision has to be made. In addition to this, there is a $currentProcPtr$ associated with each element of the decision list, which keeps track of the processor being currently evaluated for the task. A combination of $currentTaskPtr$ and

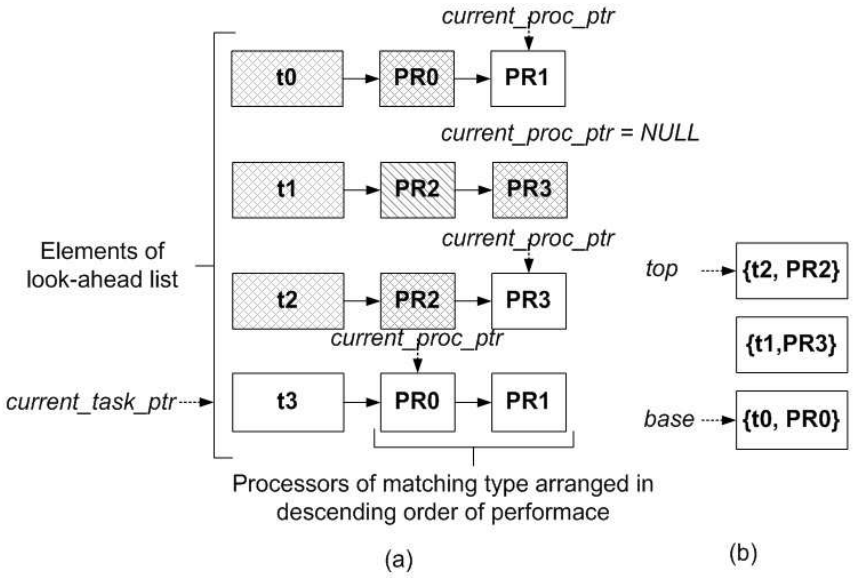


Figure 5.7. Data structures used in branch-and-bound: (a) decision list and (b) decision stack

$currentProcPtr$ is used for selecting a task to processor schedule, represented by the tuple $\{t_i, p_i\}$.

The task to processor schedule pair generated from the decision list is inserted into the *Decision Stack* that holds the list of all the tasks which are already scheduled to processors. The tasks and the processors represented by the hatched boxes in decision list of Fig. 5.7 represent those which are currently scheduled. The boxes with slanted lines represent the processors which have been already evaluated and have failed the thermal emergency check. Once all the processors corresponding to a particular task are evaluated, $currentProcPtr$ is reset to $NULL$. From the figure it can be seen that, decisions are made for the tasks t_0 to t_2 and the set of tuples $\{\{t_0, PR_0\}, \{t_1, PR_1\}, \{t_2, PR_2\}\}$ are pushed into the decision stack shown in Fig. 5.7(b). Thermal emergency check is done for the set of tasks currently present in the decision stack.

The flow chart in Fig. 5.8 shows the steps taken during the branch-and-bound heuristic. The capital letters in the top left corner of each of the boxes are used

here to explain the operation of the proposed heuristic. First, a task to processor mapping is extracted from the decision list in the Step A. This location is pointed by the $currentTaskPtr$ and the corresponding $currentProcPtr$. In Step B, this tuple is inserted into the decision stack, which is followed by thermal evaluation of all tasks to processor mappings present in the decision stack in Step C. If a thermal emergency is detected then the tuple $\{t_i, p_i\}$ is popped from the decision stack and the $currentProcPtr$ at the current location is incremented to evaluate the next fastest processor choice in the Steps D and E, respectively. If all the processor choices for the tasks have been evaluated, then we go no to the previous task in the decision list after resetting $currentProcPtr$ (Step F). If there are no more tasks to be evaluated in the decision list we report $FAILURE$ in Step G. Otherwise, the task present in the previous location is evaluated by decrementing $currentTaskPtr$ (Step G). This generates another task to processor mapping corresponding to the $currentTaskPtr$ and $currentProcPtr$ locations in Step A. The above loop continues until a schedule which does not lead to a thermal emergency, is found for the current task. In that case, we go to the next task location in the decision list by incrementing the $currentTaskPtr$ in Step J and selecting the first processor in the processor list of the new task in Step N. Finally, we go back to Step A where the tuple $\{t_i, p_i\}$ is evaluated for a thermal emergency. A solution is found (Step K), if we reach the end of the decision list which indicates that all the tasks in the decision list are scheduled, in which case, a $SUCCESS$ reported. We keep a track of the schedule which has the minimum schedule length by storing it in a temporary location (Steps L and M). This is followed by popping the decision stack and continuing search for another processor choice for the current task (Steps D onwards).

The implementation of the procedure $evaluate\_thermal\_emergency()$ which checks for the presence of a thermal emergency among the current set of scheduled tasks, is shown in the Pseudocode 5.1. It returns $TRUE$ if thermal emergency is not found

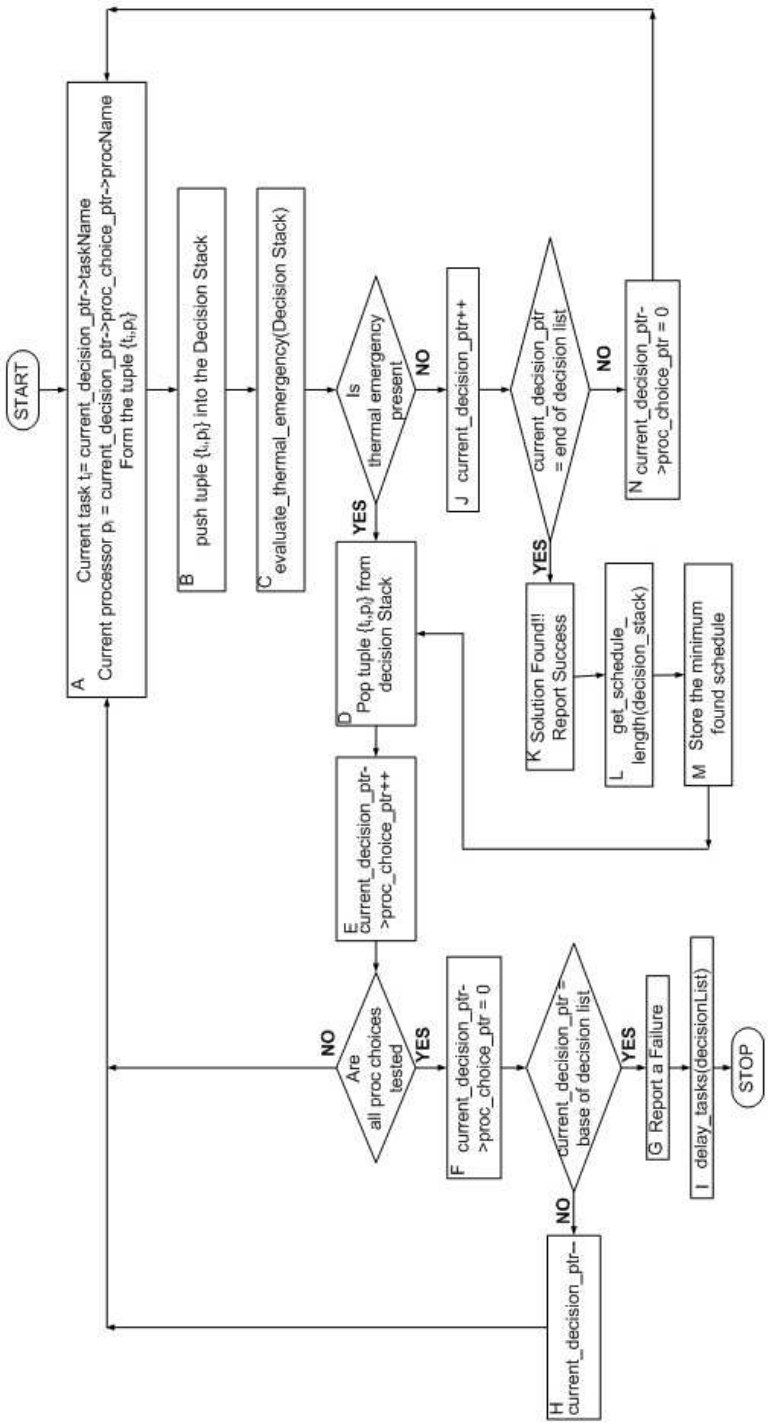


Figure 5.8. Flow chart representing branch-and-bound algorithm

and $FALSE$ otherwise. The input to the function is decision stack which holds the current set of tasks which are already scheduled. First start and end times of tasks

Algorithm 5.1 *evaluate_thermal_emergency*(*decisionStack*)

```
1: get_task_times (decisionStack, startTimeList, endTimeList)
2: for all taskIndex ∈ decisionStack.size() do
3:    tTime = endTimeList[taskIndex]
4:    for all taskIndex1 ∈ decisionStack.size() do
5:       sTime1 = startTimeList[taskIndex1]
6:       pName1 = decisionStack[taskIndex1]
7:       temperature = 0.0
8:       if thm_overlap(taskIndex, taskIndex1, tTime) then
9:          for all taskIndex2 ∈ decisionStack.size() do
10:            sTime2 = startTimeList[taskIndex2]
11:            eTime2 = endTimeList[taskIndex2]
12:            pName2 = decisionStack[taskIndex2]
13:            tPower2 = ret_task_power(taskIndex2, pName2)
14:            if thm_overlap(taskIndex1, taskIndex2, tTime) then
15:               for wvIndx = 0; wvIndx < wvCount; wvIndx++ do
16:                  if ((wvIndx · tStep) ≤ (tTime − sTime2)) & &((wvIndx · tStep) >
                     (tTime − eTime2)) then
17:                     temperature+ = get_wv_effect(pName1, pName2, pPower2, wvIndx)
18:                  end if
19:               end for
20:            end if
21:         end for
22:      end if
23:   end for
24: end for
25: if temperature > T_th then
26:    return FALSE
27: else
28:    return TRUE
29: end if
```

is determined using *get_task_times*(*decisionStack*, *startTimeList*, *endTimeList*). In the Pseudocode, the tasks are accessed by using indexes *taskIndex*, *taskIndex*1 and *taskIndex*2. In the following explanation, we would refer to a task using its index. Hence, in order to refer to a task present at location *decisionStack*[*taskIndex*1] we use *taskIndex*1. Start and end time generation is followed by checking for the presence of a thermal emergency for the end times of all the tasks in the outer loop from the Lines 2 to 24. The current target time '*tTime*' for each of the tasks *taskIndex* is determined in the Line 3. Now in the Lines 4 to 23 we check for each of the

tasks among the set of thermally overlapping tasks if there is a thermal emergency at $tTime$. The condition if tasks $taskIndex1$ and $taskIndex2$ are overlapping at the $tTime$ is checked by the function $thm\_overlap(taskIndex, taskIndex1, tTime)$. This function checks for the conditions given in the Equations 5.4 and 5.5 for pair of tasks $taskIndex$ and $taskIndex1$ at target time. Now for each of the tasks we determine the start time ($sTime1$) and the processor on which it is scheduled ($pName1$) in the Lines 5 and 6. The variable *temperature*, which stores the temperature of the task $taskIndex1$ at $tTime$ is also initialized. In order to calculate this temperature, we need to determine the temperature increase at $tTime$ due to the effect of all the other thermally overlapping tasks. This is done in the loop present in the Lines 9 to 21. The task which is obtained using $taskIndex2$ is called the overlapping task. For the overlapping task we determine the start and end times $sTime2$, $eTime2$, processor name $pName2$ and power dissipated $tPower2$. The power dissipation is obtained from the power estimation table using the function $ret\_task\_power()$ as shown in Line 13. Now, the temperature at $tTime$ is determined using the thermal response coefficients only if $taskIndex2$ overlaps at $tTime$ in Line 14. If they do overlap, we iterate through all the thermal response values which overlap at the $tTime$ in Line 15 to 19. Once the temperature at the current task $taskIndex1$ is determined, it is compared with the thermal threshold T_{th} to see if a thermal emergency is present or not. If a thermal emergency is detected, then we return a $TRUE$ value, otherwise a $FALSE$ value is returned. The overlap condition is checked in the Line 16 and the function $get\_wv\_effect()$ is used to determine the effect on $pName1$ due to power applied on $pName2$ at a time shift of $wvIndx$. It calculates each of the terms of the summation given in the Equations 5.6, 5.7 and 5.8. For a look-ahead length of L and a set of W termal response values, the run-time for the above algorithm is $O(L^3 \cdot W)$ making this the most compute intensive step in the proposed branch-and-bound heuristic.

Thus the proposed heuristic generates the following outcomes:

- *Success:* It returns a success if at least one schedule is found which eliminates thermal emergencies

- *Failure:* A failure is reported when a schedule which causes thermal emergencies cannot be determined after exhaustively searching all the possibilities

In case of failure, the following algorithm is used to perform delay insertion as explained in the next section.

5.5.2.3 Delay Insertion

Thermal emergencies are caused by thermally overlapping tasks and can be eliminated by removing the overlap by delaying selected tasks. Tasks are delayed such that the thermal effect of one of the task on the other is reduced until a thermal emergency is eliminated. The following Pseudocode 5.2 describes the function *void delay_tasks*(), which performs delay insertion to remove thermal emergencies. Firstly, it performs thermal evaluation of the tasks present in the decision list. This part is given in the Lines 2 to 23. This is similar to the the Pseudocode *evaluate_thermal_emergency*(), which checks for thermal emergencies, with a difference that, for each $taskIndex1$ it generates a list of tasks which are responsible for thermal emergency at $taskIndex1$. This is followed by delaying all the $taskIndex2$ which cause a thermal emergency at $taskIndex1$ until the thermal emergency is removed. We do not delay the tasks $taskIndex$ and $taskIndex2$. This condition is checked in the Line 27. This is followed by the function $insert\_task\_delay(taskIndex2, tTime)$ which determines the minimum amount of delay required to completely remove temperature contribution of $taskIndex2$ on $taskIndex1$ at the target time $tTime$. We keep track of the reduction in the temperature by subtracting the removed $tempContribution$ from $temperature$. Finally, after delaying tasks we update the start and end times of all the tasks by using *update_task_time*() function and checking for thermal emergencies using the

updated time values. For a look-ahead limit of L and thermal response length of W, the complexity of the above algorithm is also $O\left(L^3 \cdot K\right)$.

Algorithm 5.2 *delay_tasks*(*decisionStack*)

```
1: get_task_times (decisionStack, startTimeList, endTimeList)
2: for all taskIndex ∈ decisionStack.size() do
3:    tTime = endTimeList[taskIndex]
4:    for all taskIndex1 ∈ decisionStack.size() do
5:       sTime1 = startTimeList[taskIndex1]
6:       pName1 = decisionStack[taskIndex1]
7:       temperature = 0.0
8:       tempcontributionList.clear()
9:       if thm_overlap(taskIndex, taskIndex1, tTime) then
10:         for all taskIndex2 ∈ decisionStack.size() do
11:            sTime2 = startTimeList[taskIndex2]
12:            eTime2 = endTimeList[taskIndex2]
13:            pName2 = decisionStack[taskIndex2]
14:            tPower2 = ret_task_power(taskIndex2, pName2)
15:            if thm_overlap(taskIndex1, taskIndex2, tTime) then
16:               for wvIndx = 0; wvIndx < wvCount; wvIndx + + do
17:                  if ((wvIndx · tStep) ≤ (tTime − sTime2)) & &((wvIndx · tStep) >
                     (tTime − eTime2)) then
18:                     temperature+ = get_wv_effect(pName1, pName2, pPower2, wvIndx)
19:                  end if
20:               end for
21:            end if
22:         end for
23:      end if
24:      tempContributionList.insert(taskIndex2, temperature)
25:      while temperature > T_th do
26:         taskIndex2 = tempContributionList.next()
27:         if taskIndex2 ≠ taskIndex1 & &taskIndex2 ≠ taskIndex then
28:            tempContribution = tempContributionList.next()
29:            insert_task_delay(taskIndex2, tTime)
30:            temperature = temperature − tempContribution
31:            update_task_times()
32:         end if
33:      end while
34:   end for
35: end for
```

While executing the above branch-and-bound heuristic, there is a chance that the temperature at processor core executing the heuristic itself increases and becomes high. In order to prevent thermal emergencies this processor has to keep track of its

own temperature. Branch-and-bound is terminated if the temperature comes close to the thermal emergency threshold. In that case only delay insertion is done and the execution of this heuristic is halted until the temperature cools down. Moreover, DVFS based thermal management techniques could be used in order to reduce temperature by reducing the core performance and hece the power dissipation.

Next we explain the ILP formulation which is used to obtain a minimum schedule length statically while preventing thermal emergencies.

5.6 ILP Formulation

Consider a task graph G consisting of set of tasks $t_i \in G$ scheduled on an MPSoC with P processors. For the task t_i executing on a processor p_j the power dissipation is given by ρ_{ij}. Each of the tasks has a type, and the task can only execute on a processor with the matching type. The problem of thermal aware scheduling consists of determining a schedule with minimum length, which prevents thermal emergencies. Therefore, the set of ILP constraints for the above goal, consist of the following parts.

- Formulation for schedule length minimization
- Formulation for temperature calculation in order to evaluate thermal emergencies, based on current schedule
- An objective function which determines the total schedule length of the task graph so as to minimize it.

Next we will discuss each of the above parts in detail.

5.6.1 Schedule Length Minimization

The ILP formulation for schedule length minimization is adapted from Tosun *et al.* in [105]. The readers are advised to refer to the above work for details about this ILP formulation.

5.6.2 Thermal Evaluation

Temperature calculation is done by using the thermal response table generated in the pre-characterization phase. In order to reduce the complexity of the formulation, the original set of thermal response values are approximated to a smaller set of response values. These new set of responses consists of just one base response given by $\Phi(t)$. The base response is selected among original set of thermal responses, as the one which has the highest maximum temperature value. Hence, the scaling values for the base response are given by the following set of equations.

$$\phi(t) \leftarrow \phi_{kl}(t): \quad \max_{t \in W} \{\phi_{kl}(t)\} \geq \phi_{mn}(t) \qquad \forall k, l, m, n \in P \tag{5.12}$$

All the other thermal responses are represented by scaling the base thermal response using scaling values γ_{kl}. The scale is determined as a ratio between the maximum temperature for the corresponding original thermal response to that of the base thermal response. Hence we have the following equation.

$$\gamma_{kl} = \frac{\max_{t \in W} \{\phi_{kl}(t)\}}{\max_{t \in W} \{\phi(t)\}} \tag{5.13}$$

Thus the wavelet coefficient set is given as a combination of the base response $\phi(t)$ and the set of scaling values γ_{kl}, representing the effect on processor p_l due to power dissipated on processor p_k. This can be seen in Table 5.2 which compares the original thermal response presented in Table 5.1 with the approximate one. Here, each of the thermal response values $\phi_{kl}(t)$ are replaced by the corresponding scaling factors γ_{kl}. It can be seen that, the size of the approximate thermal response table is $O(P^2 + W)$ as compared to $O(P^2 \cdot W)$ for the original thermal response table. This approximation leads to an over-estimation of temperature response. This is because, it does not consider the delay introduced in the temperature response of processor

PR_l as compared to that at the processor PR_k. Consequently, it assumes that the temperature responses at a pair of separate locations are varying simultaneously.

Table 5.2. The thermal response table shown in Table. 5.1 is compared with the approximate thermal response table used for ILP formulation

Unit power wavelet applied at ..	Thermal response measured at ..	Thermal response values $t \in W$	Scaling factors for approximated thermal response values obtained from base response $\phi(t) : t \in W$
$PR0$	$PR0$	$\phi_{00}(t)$	γ_{00}
	$PR1$	$\phi_{01}(t)$	γ_{01}
	$PR2$	$\phi_{02}(t)$	γ_{02}
$PR1$	$PR0$	$\phi_{10}(t)$	γ_{10}
	$PR1$	$\phi_{11}(t)$	γ_{11}
	$PR2$	$\phi_{12}(t)$	γ_{12}
$PR2$	$PR0$	$\phi_{20}(t)$	γ_{20}
	$PR1$	$\phi_{21}(t)$	γ_{21}
	$PR2$	$\phi_{22}(t)$	γ_{22}

Suppose we need to determine the thermal response at the end time of target task t_{i_1}. As temperature estimation is done only at the end of task execution, the target time for our case is the end time et_i. Execution of the pair of tasks t_{i_1} and t_{i_2} is thermally overlapping at the target time et_i if Equations 5.4 and 5.5 are satisfied. Rewriting Equations 5.4 and 5.5 the thermal overlap condition is given by the following set of equations.

$$et_{i_1} - st_{i_2} > 0 \tag{5.14}$$

$$et_{i_1} - et_{i_2} \leq W \tag{5.15}$$

We define shift Boolean variables $\alpha_{i_1 i_2 \tau}$ which are $TRUE$ for shift τ of the unit power wavelet representing task t_{i_2}, such that the thermal response overlaps at the target time et_{i_1}. Thus, for a pair of tasks $\alpha_{i_1 i_2 \tau}$ variables are generated for all the W shifts of the wavelet response. Moreover, we define variables $y_{i_1 i_2 \tau}$, which are $TRUE$

when Equation 5.14 is satisfied and $z_{i_1 i_2 \tau}$ which is $TRUE$ when Equation 5.15 is satisfied. Therefore we have the following set of constraints.

$$D \cdot (1 - y_{i_1 i_2 \tau}) + ((et_{i_1} - st_{i_2}) - \tau) \geq 1 \tag{5.16}$$

$$D \cdot y_{i_1 i_2 \tau} - ((et_{i_1} - st_{i_2}) - \tau) \geq 0 \tag{5.17}$$

$$D \cdot (1 - z_{i_1 i_2 \tau}) + ((et_{i_1} - et_{i_2}) - \tau + W) \geq 0 \tag{5.18}$$

$$D \cdot z_{i_1 i_2 \tau} + ((et_{i_1} - et_{i_2}) - \tau + W) \geq 1 \tag{5.19}$$

Consequently, $\alpha_{i_1 i_2 \tau}$ is $TRUE$ when both $y_{i_1 i_2 \tau}$ and $z_{i_1 i_2 \tau}$ are $TRUE$.

$$z_{i_1 i_2 \tau} + (1 - \alpha_{i_1 i_2 \tau}) \geq 1 \tag{5.20}$$

$$y_{i_1 i_2 \tau} + (1 - \alpha_{i_1 i_2 \tau}) \geq 1 \tag{5.21}$$

$$(1 - z_{i_1 i_2 \tau})(1 - y_{i_1 i_2 \tau}) + \alpha_{i_1 i_2 \tau} \geq 1 \tag{5.22}$$

Here, D is a large constant such that:

$$D > et_i : \quad \forall i \in G \tag{5.23}$$

Now we define $T^{ovlp}_{i_1 i_2}$, a set of real variables which represent the temperature at end of task t_{i_1} because of the overlap with task t_{i_2}. In calculating this temperature, we do not consider the scaling effect of power and the spatial location due to scheduling. Therefore, we have the following equation:

$$T^{ovlp}_{i_1 i_2} = \sum_{\tau = et_i - W}^{et_i} \phi\,(et_i - \tau) \cdot \alpha_{i_1 i_2 \tau} \tag{5.24}$$

Now, we define new set of real variables $T_{i_1 i_2 j_2}$ which represent the temperature at task t_{i_1} when the task t_{i_2} is scheduled on a processor p_{j_2}. If task t_{i_2} is scheduled

on processor p_{j_2} then the Boolean variable $s_{i_1 j_2}$ is $TRUE$. Therefore, in this case, we have $T_{i_1 i_2 j_2} = T^{ovlp}_{i_1 i_2}$. Hence, we obtain the following equations.

$$(T_{th} \cdot (1 - s_{i_1 j_2}) + T_{i_1 i_2 j_2}) \geq T^{ovlp}_{i_1 i_2} \tag{5.25}$$

$$(T_{th} \cdot (s_{i_1 j_2} - 1) + T_{i_1 i_2 j_2}) \leq T^{ovlp}_{i_1 i_2} \tag{5.26}$$

$$(T_{th} \cdot s_{i_1 j_2} + (1 - T_{i_1 i_2 j_2})) \geq 1 \tag{5.27}$$

$$\forall t_{i_1}, t_{i_2} \in G \qquad \forall p_{i_1}, p_{i_2} \in P$$

Here, T_{th} is the thermal emergency threshold. Now, we determine the temperature response on all the processors at the end time of the task t_{i_1}. In the above equations, we have not considered the power dissipated and the spatial response scaling for any of the tasks. Therefore, we define a real variable $Ts_{i_1 j}$ which represent the temperature at the processor p_j at the end time of task t_{i_1}.

$$Ts_{i_1 j} = \sum_{t_{i_2} \in G} \left(\sum_{p_{j_2} \in P} T_{i_1 i_2 j} \cdot \rho_{i_2 j_2} \cdot \gamma_{j_2 j} \right) \tag{5.28}$$

$$\forall p_j \in P \tag{5.29}$$

The temperature at task t_{i_1} is given by the variable T_{i_1} and is equal to the maximum value of the response among all the processors.

$$T_{i_1} \geq Ts_{i_1 j} \qquad \forall p_j \in P \tag{5.30}$$

Finally, the maximum temperature is given by T_{max} as follows:

$$T_{max} \geq T_{i_1} \qquad \forall t_{i_1} \in G \tag{5.31}$$

If T_{th} is the thermal emergency threshold, then T_{max} should not exceed it. Therefore we have the following constraints.

$$T_{max} < T_{th} \tag{5.32}$$

5.6.3 Objective Function

The objective function minimizes schedule length while preventing thermal emergency. We define a time t_{final} which is equal to the schedule length for the current task to processor allocation obtained from the ILP formulation done for schedule length minimization.

$$t_{final} \geq et_i \qquad \forall t_i \in G \tag{5.33}$$

Therefore, the objective function minimizing the total schedule length is given by the following equation.

$$Obj = minimize\,\{t_{final}\} \tag{5.34}$$

5.7 Results

In order to evaluate the proposed approach, task graphs were generated using Task Graphs For Free (TGFF) [36]. TGFF is a freely available tool which is used to generated pseudo-random task graphs based on the input parameters. Each task has a base power consumption which is the amount of power consumed if it is scheduled on the fastest processor. These power consumption values for the tasks were assigned randomly in a range varying from 10 to 15 Watts. These values are used to generate the power estimation table. HotSpot [101] temperature modeling tool is used here for thermal simulation of the power trace generated from task execution. The following Table 5.3 shows thermal parameters were used for the die package system.

Table 5.3. Thermal parameters of the die package system in Hotspot tool

Parameter	Value
Chip Thickness	$600um$
Substrate Convection Resistance	$0.01mK/W$
Substrate Heat Capacity	$1750000J/m^3K$
Heat Sink Thermal Resistance	$0.1K/W$
Heat Sink Thermal Capacity	$140J/K$
Ambient Temperature	$320K$
Base Processor Frequency	$2GHz$
Power Sampling Interval	$250us$

The MPSoC floorplan used in the following experiments is show in Fig. 5.9 below, where each of the processors has the same dimensions. This MPSoC consists of 12 processors each of which can have one of four different types. It can be seen that, all the processors in a column have the same type. Moreover, corresponding to each of the processors there is an associated *Relative Execution Time* for all the processors in a row, which determines the amount of time and power dissipated by a task on the particular processor. For example, consider a task taking x amount of time and dissipating a power of p watts on processor with a relative execution time of one unit. If this task is scheduled on a processor with a relative execution time of r, then it will take $r \cdot t$ amount of time and will dissipate p/r amount of power. Therefore, it can be seen that the processors in the row 1 and 2 are the fastest while those in the row 3 have half the performance and power dissipation. Therefore, any task migration should move a task to a processor amongst rows 1 and 2 in order to avoid thermal emergency and only if a thermal emergency cannot be avoided by doing so, then it should move the task to a slower processor in the row 3.

The above mentioned floorplan is useful in evaluating proposed task migration approach. Let there be a task causing thermal emergency, executing on any one of the processors in the first row. Thermal emergencies are caused not only due to the power dissipated in the given task, but also because of the power dissipated in the neighboring tasks which are thermally overlapping with the current task. This

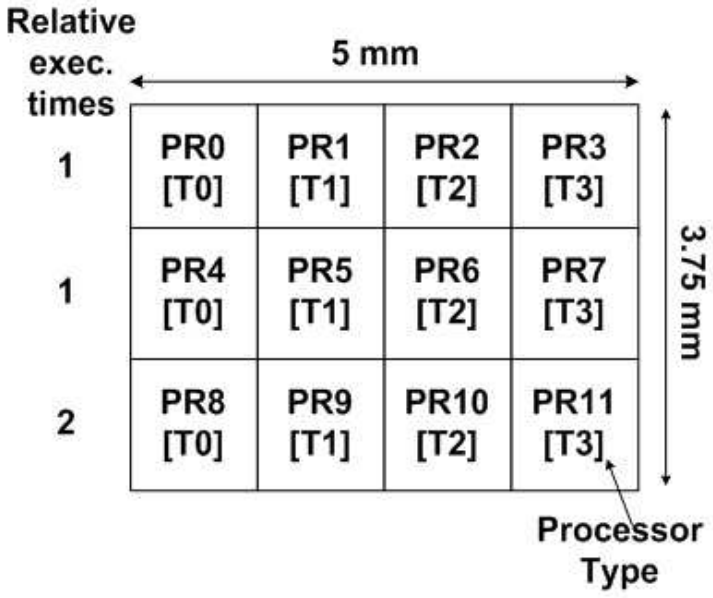


Figure 5.9. MPSoC floorplan with 12 processor cores

thermal emergency can be eliminated by rescheduling it to a processor in the second or the third row. Rescheduling the task to a processor in the second row may help reduce the temperature by increasing the distance between the thermally overlapping tasks, while rescheduling to a processor in the third row prevents thermal emergency by reducing the power to half of the maximum power at the cost of a two fold increase in execution time. A good thermal aware scheduling approach should select the former solution most of the times and resort to the latter only if the former fails to prevent the thermal emergency.

Next we will present results for pre-characterization and run-time stages.

5.7.1 Pre-characterization Stage

The thermal behavior of die and package is characterized in this stage. This is done by applying a unit power wavelet at each processor and measuring the corresponding thermal response at all the other processors. This is shown in Fig. 5.10, where the unit power wavelet is applied to the processor $PR0$ at the bottom of the figure and

the corresponding temperature variation at all the other processors is measured. The pulse width for Haar wavelet is set to $625\mu s$. Thus, for all the subsequent simulations, time is discretized to $625\mu s$. The corresponding thermal response obtained is sampled at time instances which are integral multiples of the unit power wavelet pulse width. This leads to the generation of the set of thermal responses $\phi_{0j}(t) : \forall j \in p$ on processor p_j corresponding to the power applied at $p_0(t)$. This is repeated for all the other processors to generate the thermal response table $\phi_{ij}(t)$. Moreover, from the figure we can see that thermal response length of $W = 40$ *samples* is used here.

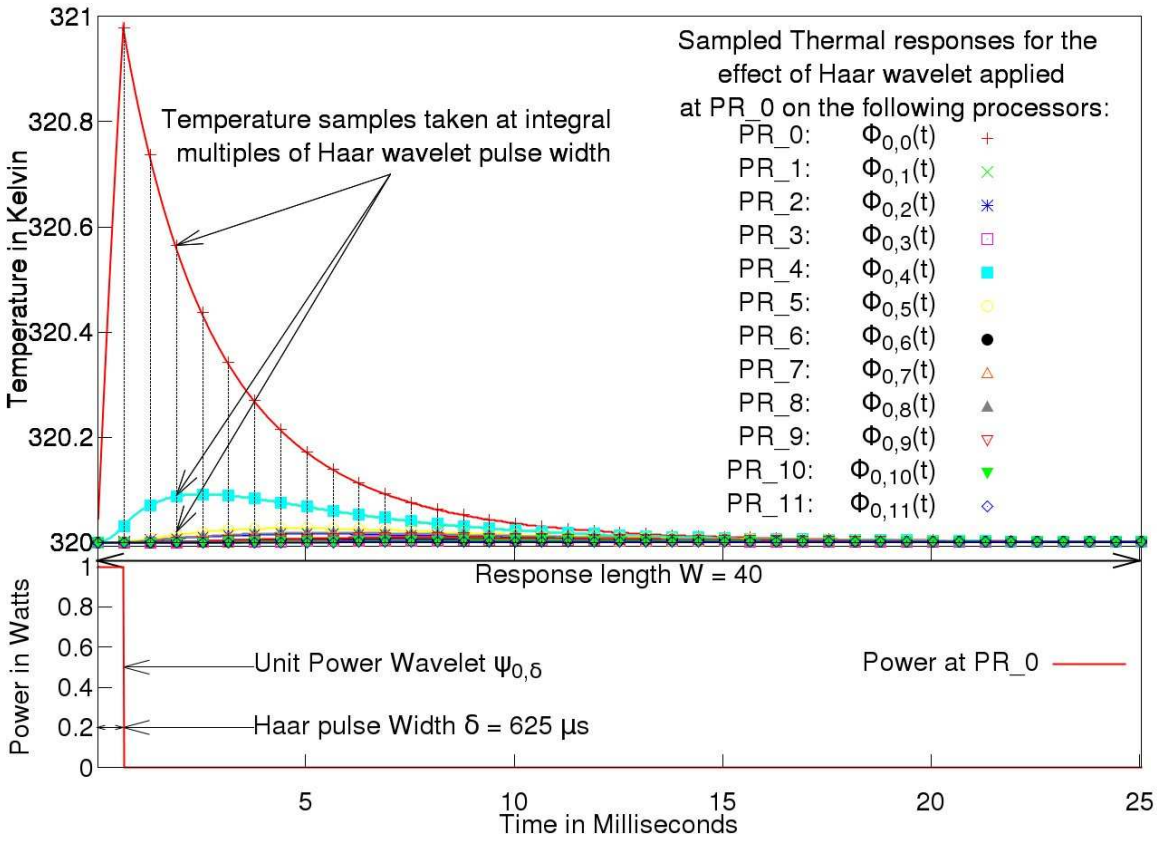


Figure 5.10. Thermal response coefficients generated for a unit power wavelet applied at PR0

5.7.2 Run-time Stage

In run-time stage, task migration is done based on temperature prediction using thermal response table generated during pre-characterization. If a thermal emergency is predicted, then branch-and-bound based heuristic is invoked to minimize the schedule for the set of tasks in the look-ahead list, so as to eliminate the thermal emergencies. The quality of the final schedule and the temperature prediction depends on the length of the look-ahead list used in this approach.

5.7.2.1 Look-ahead Length Selection

The amount of look-ahead used for task migration determines the quality of the final schedule. Looking ahead for a small number of tasks helps us in dividing the scheduling problem for the whole task graph into smaller sub-problems which can be solved with much less computational resources. Consequently, it should be noted that the final solution obtained is sub-optimal. For a look-ahead list consisting of L tasks, where each task can be scheduled on p processors, the complexity of task scheduling is exponential with respect to look-ahead length. This is given by $O\left(p^L\right)$. This implies that the number of times *evaluate_thermal_emergency*() function is called is given by $O\left(p^L + p^{L-1} + p^{L-2}...\right)$. For MPSoC architecture shown in Fig. 5.9 each task can be schedule on 3 processors with the matching type. As a result, for look-ahead length of L tasks, the number of times thermal emergency check is done is $O\left(3^L + 3^{L+1} + 3^{L+2}...\right)$. Due to limited computation capacity of each of the processors in the MPSoC, searching for a large search space is not feasible. We make a conservative assumption that each of the processors cannot handle more than 2000 possible calls to *evaluate_thermal_emergency*(). The maximum value of $L = 6$ for which the number of calls to *evaluate_thermal_emergency*() is less than 2000.

5.7.2.2 Temperature Prediction

In our approach, temperature prediction is done only at end times of the task execution. Due to the limited length of the thermal responses corresponding to various tasks, there is an error induced in temperature prediction with increase with look-ahead length. In order to evaluate the accuracy of temperature prediction, the predicted temperature values were compared with those obtained from running the same power trace using Hotspot. Table 5.4 shows the temperature prediction error.

Table 5.4. Temperature prediction error

Average Error	0.88
Standard Deviation	0.83
Max. Error	1.39

Temperature variation for a task graph referred to as BM6 with 30 tasks executing on the MPSoC shown in Fig. 5.9, is shown in the Fig. 5.11. The thermal emergency threshold is set to $380K$. It can be seen that, the execution of the task graph causes thermal emergencies at three different locations. These thermal emergencies are referred as $TE1$, $TE2$, and $TE3$, respectively. All the tasks in this case are scheduled among the processors $PR0$ to $PR3$. This gives an original schedule length of 10.7 seconds. The temperature predicted at the end of task graph execution is represented by the black step curve. The predicted values are represented using circles at the end of the corresponding task executions. These predicted values are compared with the actual temperature and the temperature prediction error e is calculated as a difference between the actual and the predicted temperatures. The calculated error values are shown only for the tasks which cause thermal emergencies. It was observed that temperature prediction is optimistic; the actual temperature could be slightly higher than the predicted value which causes a $+ve$ error. The inaccuracy is eliminated by reducing the actual thermal emergency threshold by 2 degrees.

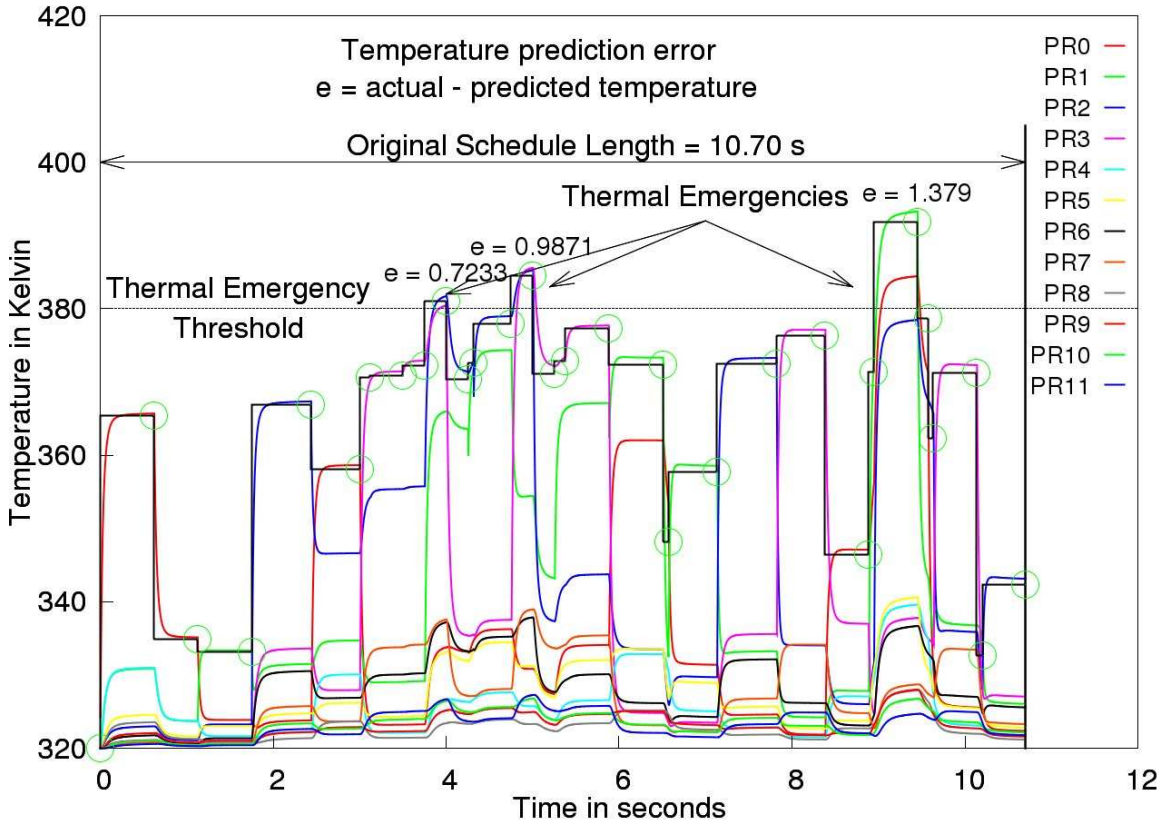


Figure 5.11. Temperature variation for BM6 task graph with 6 number of tasks with the corresponding temperature prediction values

5.7.3 Effect of Core Count on Task Migration

The following experiment was done in order to evaluate the importance of task migration with increase in the total number of processors in the MPSoC. We compared our results with *Dynamic Frequency Scaling* (DFS) based technique. The task graph (BM6) with 30 tasks used in the previous experiments is also used here.

As shown in the Fig. 5.12, the number of cores in the MPSoC is varied in steps of 4 starting from 4 to 16. The relative execution time of all the processors is set to one unit so that the task migration becomes the only reason responsible for schedule length improvement. The initial schedule consists of assigning tasks among the processors $PR0$ to $PR3$ which leads to the set of thermal emergencies as shown in the Fig. 5.11.

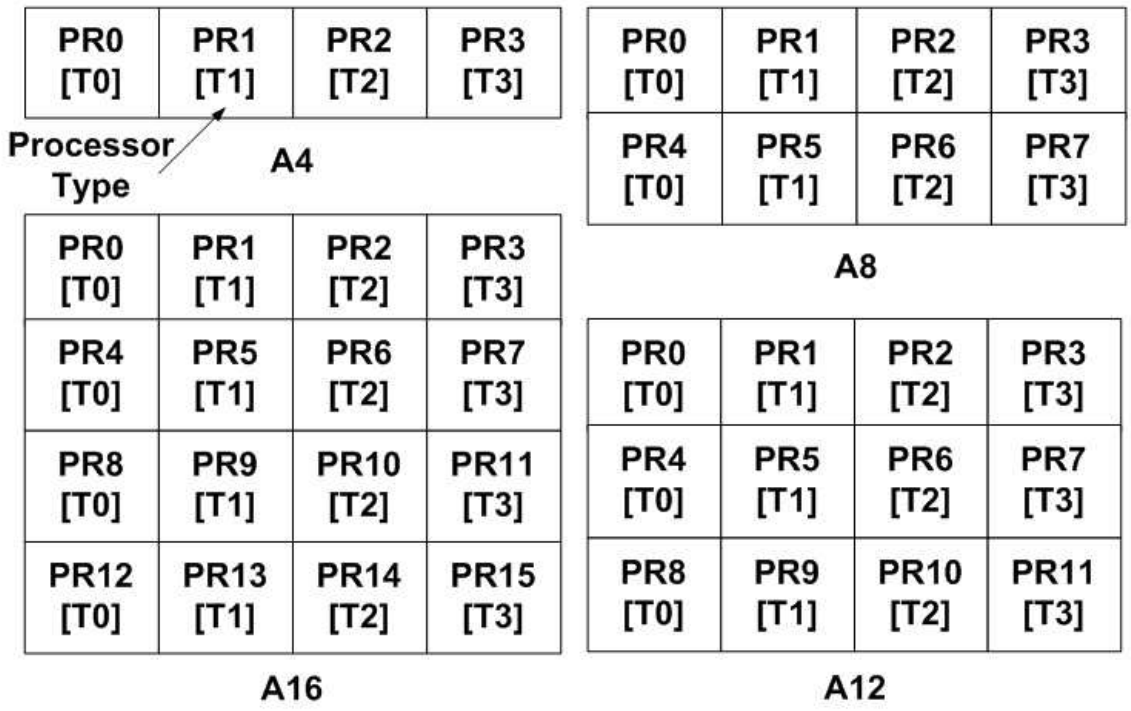


Figure 5.12. MPSoCs with 4, 8, 12 and 16 processors represented by using the architecture names A4, A8, A12, and A16 respectively

Branch-and-bound heuristic is run with a look-ahead limit of 6 tasks. If no solution preventing thermal emergency is found, then delay insertion is invoked.

Table 5.5. Dynamic frequency scaling parameters

Number of Frequency Levels:	2
Frequency Level 1:	Nominal Frequency
Frequency Level 2:	Half of Nominal Frequency
Temperature sampling interval:	$250\mu s$
Thermal emergency threshold:	$380K$
DFS threshold:	$378K$

Table 5.5 shows the various DFS parameters used in our experiments which are obtained based on [16]. As DFS does not predict temperature, in order to avoid a thermal emergency, it has to be conservatively initiated at a lower temperature of $378K$ as compared to the thermal threshold. The DFS approach samples the temperature at every $250\mu s$ period and takes action if the temperature is found to

be higher than DFS threshold. The action involves reducing the frequency to half of the nominal frequency. As a result some tasks whose temperature is higher than the DFS threshold but lower than thermal emergency threshold also get throttled and their performance gets penalized. Therefore, the total schedule length obtained from DFS can be higher than that obtained by migration.

Table 5.6 shows the schedule length variation with the number of cores. DFS gives a schedule length of 10.735 seconds. This does not change with the number of cores as the task schedule is static for DFS. We compare the effectiveness of task migration and delay insertion with increase in the number of cores.

Table 5.6. Schedule length variation with the number of cores

Original Schedule length 10.70 *seconds*
Schedule Length from DFS 10.734 *seconds*

Architecture	Schedule length in seconds	Action Taken
A4	11.55	Delay Insertion
A8	10.95	Delay + Task Migration
A12	10.7	Task Migration
A16	10.7	Task Migration

From the table it can be seen that DFS does better for the 4 core case because there is no scope for task rescheduling, hence the only action taken by the proposed approach is delay insertion. This gives a large schedule length as compared to DFS. With increase in the number of processors in the subsequent cases, a combination of task migration and delay insertion is required which eventually gives a shorter schedule length as compared to DFS. For the 8 core case, a combination of task migration and delay insertion does not give a schedule length smaller than DFS. Consequently, for the 12 and 16 core cases, as task migration is the only action taken the total schedule length obtained reduces below that obtained from DFS. This shows that with increase in the number of cores in an MPSoC task migration can do better as compared to DFS based approach.

Fig. 5.13 to 5.15 show the execution of the above task graph on the 12 core architecture A12. Fig. 5.13 shows how dynamic frequency scaling eliminates thermal emergencies $TE1$ $TE2$ and $TE3$ shown in Fig. 5.11. As a result of DFS the original schedule length of 3.70 seconds is increased to 3.735 seconds.

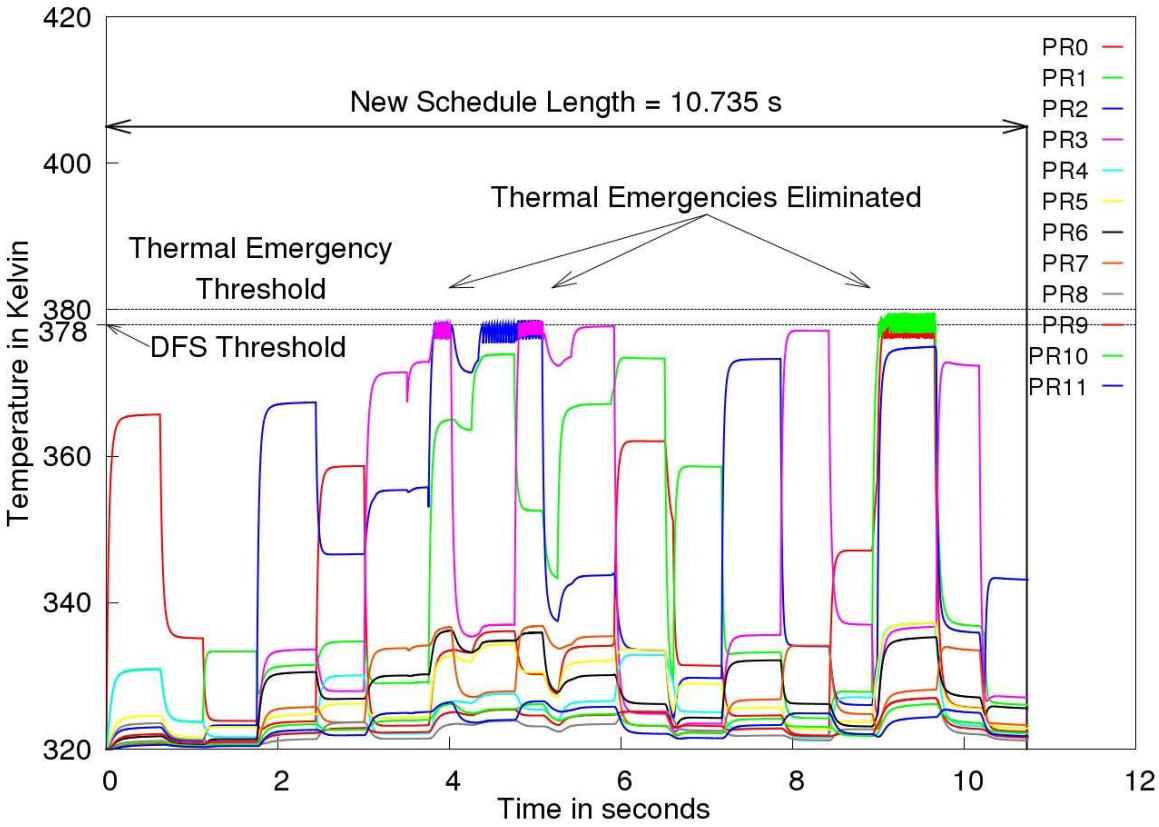


Figure 5.13. Elimination of thermal emergencies using DFS

Fig. 5.14 shows how thermal emergencies are eliminated when task migration is done. The schedule length of 3.70 seconds obtained from task migration is less than that obtained from DFS. Fig. 5.15 shows how the tasks are rescheduled in order to prevent thermal emergencies. The tasks that are rescheduled are shown in red boxes. The blue boxes show the time at which various thermal emergencies occur. It can be seen that the thermal emergency $TE1$ is avoided by moving the tasks t_{10} and t_9 away from task t_7. Similarly $TE2$ is avoided by moving the tasks t_{17} away from t_{18} and t_{12}. Finally, thermal emergency $TE3$ is avoided by rescheduling task t_{24}. As the resultant

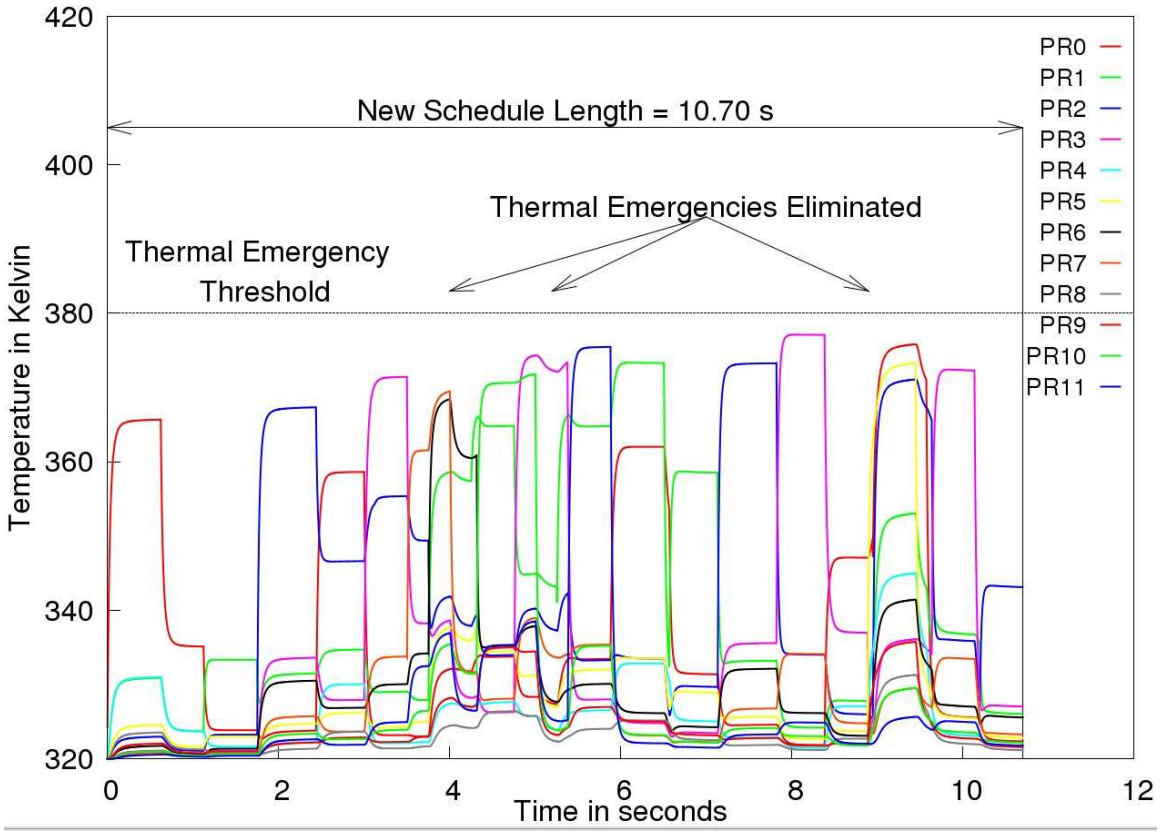


Figure 5.14. Elimination of thermal emergencies from run-time task migration

schedule length is same as the original schedule length of 10.70 seconds our proposed approach is able to prevent thermal emergencies without schedule length increase.

5.7.4 Comparison with ILP Based Static Approach

The following experiment was done in order to compare the proposed branch-and-bound heuristic for task migration with the exact solution obtained from ILP. The number of nodes in the task graphs is varied randomly from 5 to 11. *Gnu Linear Programming Kit* (GLPK) [5] was used to solve the ILP formulation. The branch-and-bound heuristic was run with a look-ahead length of 6 tasks. The twelve core MPSoC architecture with the floorplan shown in Fig. 5.9 was used here.

Fig. 5.16 shows the comparison of the schedule length obtained from the ILP based formulation, the proposed task migration heuristic and DFS. It can be seen that the

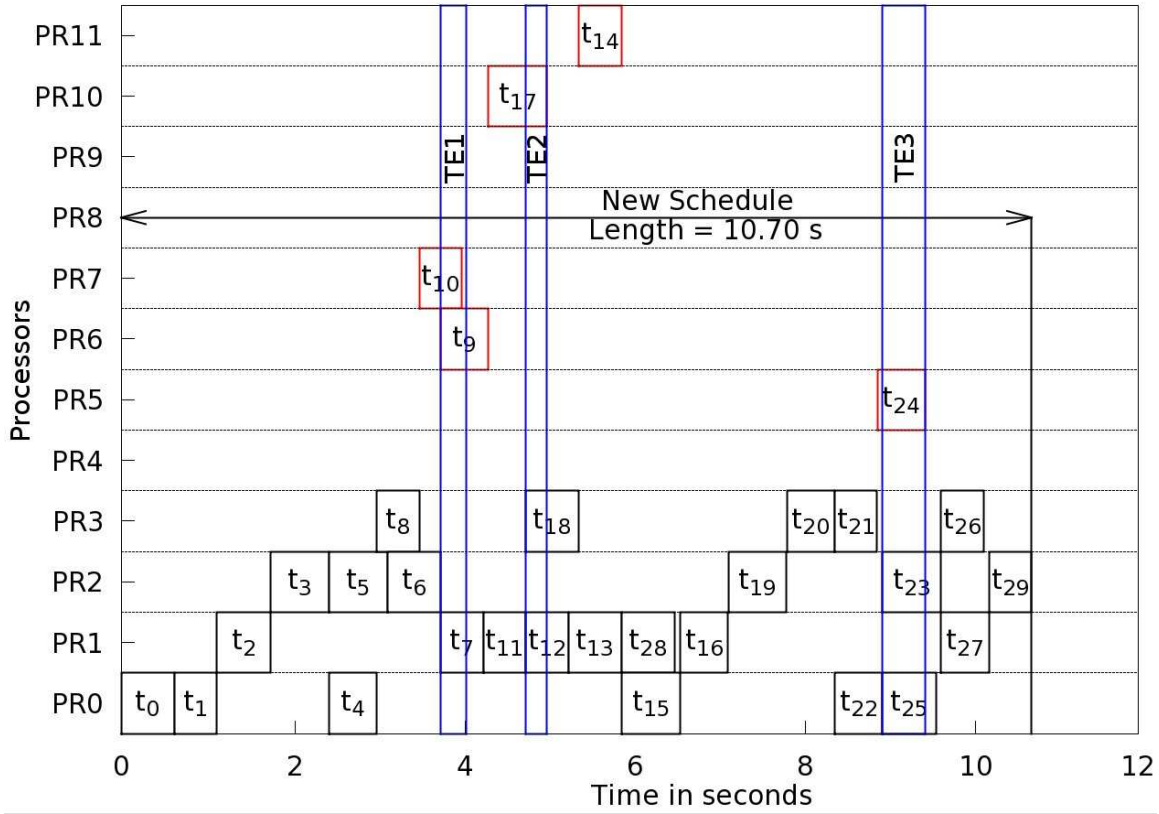


Figure 5.15. Effect of task migration on task execution

proposed task migration approach gives a schedule length very close to the absolute minimum obtained from the ILP based approach. For BMi6 task migration gives a shorter schedule length as compared to ILP. This is because of the approximation done for the thermal response values in order to reduce the complexity of ILP formulation. As a result of which, the ILP ends up over-estimating the temperature. Consequently, ILP results in scheduling a task to a slower processor as it overestimates temperature at a faster processor.

5.7.5 Comparison of branch-and-bound with DFS

The proposed branch-and-bound algorithm was run for large task graphs on the MPSoC architecture shown in the Fig. 5.9. The original schedule was generated by using a greedy heuristic. For greedy scheduling, the task graph was propagated

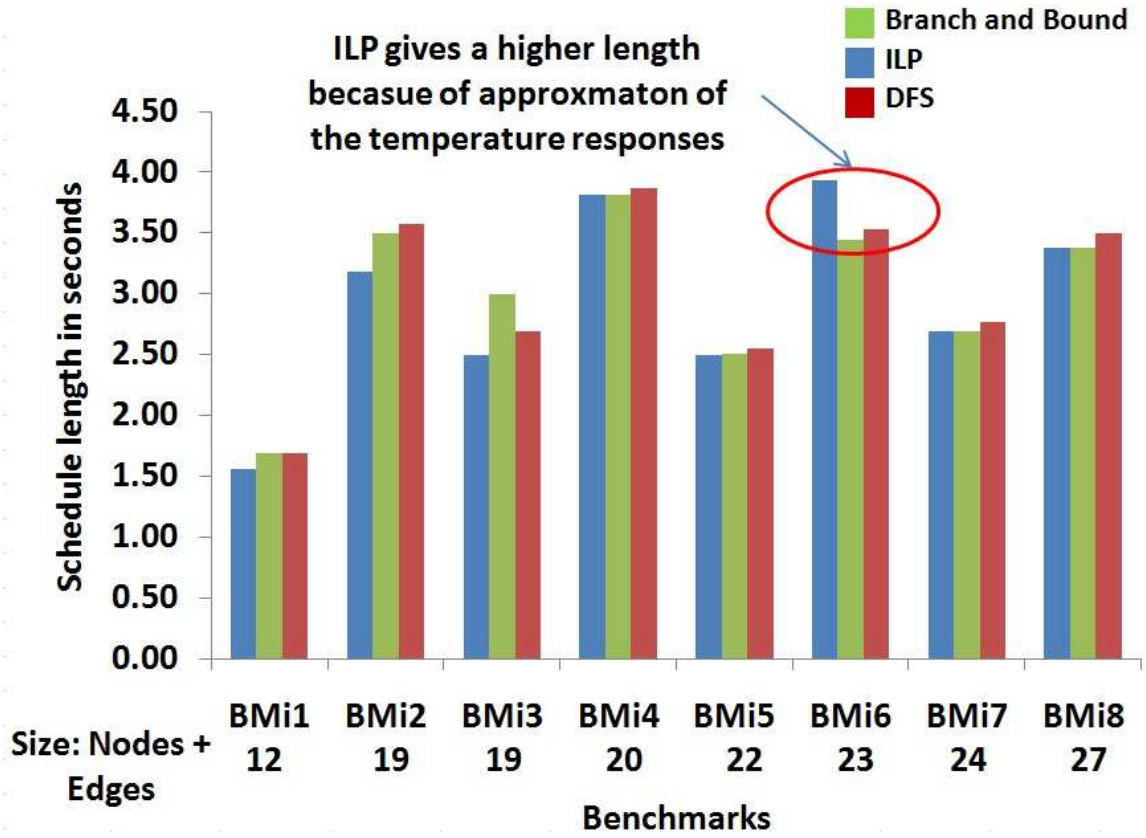


Figure 5.16. Comparison of ILP with branch-and-bound based heuristics

topologically and the ready tasks were scheduled to the fastest processor available. The original greedy schedule has thermal emergencies which are eliminated by running the proposed branch-and bound heuristic with a look-ahead length of 6 tasks. The results are compared with DFS as shown in Fig. 5.17.

It can be seen from Fig. 5.17 that the total schedule length obtained from the proposed approach is smaller than that obtained from DFS except for the BM1, BM4 and BM9. In this case branch-and-bound gives highest schedule length. DFS does better because the increase in task execution time gets absorbed in the available additional slack for the tasks. For rest of the other cases DFS give a higher schedule length than branch-and-bound.

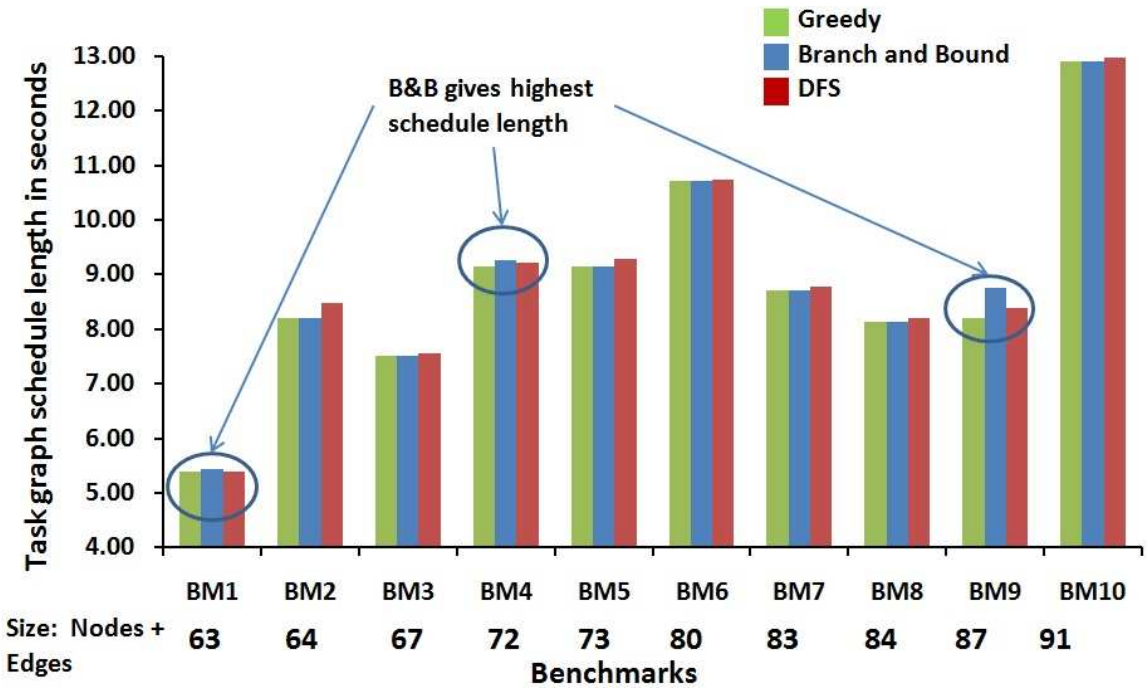


Figure 5.17. Comparison of schedule length for large task graphs obtained from branch-and-bound heuristic with greedy scheduling and DFS approaches

5.8 Conclusions

With high level of integration and increased power density thermal hotspots have become common. Hotspots can lead to reliability and performance issues and effect design convergence. In current generation ICs, DTM techniques like frequency and/or voltage scaling are used to control temperature by reducing power dissipation. In this work we propose a run-time look-ahead based branch-and-bound scheduling heuristic based on temperature prediction which is used to eliminate thermal emergencies while minimizing schedule length. In the case that task migration fails to prevent thermal emergency a delay insertion based technique is used to remove task thermal overlaps among the tasks. Finally an ILP based scheduling approach which achieves the above goals statically is also presented. The above solutions leverages upon a wavelet based thermal modeling approach where an application independent pre-characterization of the thermal system is done. We show that task migration becomes more effective

in eliminating thermal emergencies with increase in the number of processor cores in the MPSoC as compared to DFS. Results show that temperature prediction could be done with error within 2K. Moreover, schedule length obtained from the proposed dynamic scheduling approach was compared with that obtained from dynamic frequency scaling and the proposed ILP based approach. The results show that the dynamic task scheduling for most of the cases results in 2% to 5% smaller schedule lengths as compared to that obtained from frequency scaling and a maximum schedule length increase of 5% to 10% of that obtained from ILP.

CHAPTER 6

CONCLUSIONS

Due to device scaling, there has been a substantial increase in transistor count and density. In spite of this, extracting proportional improvement in performance with the available transistors has become a challenging task, as a result of various non-ideal effects arising due to shrinking geometries. These non-ideal effects manifest themselves in terms of performance degradation and reliability problems/faults in the design. This has lead to a substantial increase in verification/validation effort and cost. In order to achieve timely design convergence, these faults are either fixed at design time or tested during manufacturing time. In this work, we present heuristic solutions for various such problems and different levels of abstractions.

One of the major causes of design convergence problems in VLSI is capacitive crosstalk. Due to shrinking inter-wire distances with scaling, the number of crosstalk violations has been increasing. In this work, an Integer Linear Programming (ILP) based ATPG technique is presented in order to maximize crosstalk induced delay at the victim net for multiple aggressor crosstalk fault scenario. We presented an integrated approach which performs maximal aggressor excitation and fault propagation using a single ILP formulation and compare it with a novel partitioning based approach which deals with the above two problems separately. The above approaches are applied for both zero and unit delay models. The results indicate that percentage of total capacitance that can be switched varies from 75-100% for zero delay and 30-80% for variable delay case while propagating the fault effect to the primary output. It has been seen from the results that a better quality solution is obtained from the

integrated approach as compared to the partitioning based approach which is faster, as the number of equations is fewer.

Power supply switching noise has become one of the leading causes of signal integrity related failures, as a result of voltage scaling, in deep sub-micron designs. Traditional peak current estimation approaches involve addition of peak current associated with all the CMOS gates which are switching in a combinational circuit. In this work, an ILP based technique for generation of an input pattern pair so as to maximize switching supply currents for a combinational circuit in the presence of integer gate delays, is presented. Moreover, it is observed that the non-zero gate delay assumption not only improves the accuracy of the solution, but also helps us in reducing a single instance of the problem size by restricting the focus on only the set of active gates for a given time instant.

With high level of integration, Multi-Processor Systems on Chip (MPSoC) feature multiple processor cores and accelerators on the same die. Traditional approaches of hardware-software co-design for MPSoCs involve representing an application in the form of a task graph and employing static scheduling in order to minimize the schedule length. The drawback of static scheduling is that dynamic system behavior is not taken into consideration. In order to do dynamic scheduling we require the knowledge of the application task graph at run-time. In this work, a run-time task graph extraction heuristic to facilitate a novel game theory based dynamic scheduling is presented. The results show that our technique extracts a phase graph in fewer than 250 iterations which is several orders of magnitude smaller than the millions of iterations which a task graph goes through during the execution of an application. Moreover, typically in fewer than 100 iterations, the proposed game theory based dynamic scheduling approach is able to obtain a schedule length comparable to that obtained from ILP based absolute minimum.

Power density in modern VLSI ICs has gone up substantially, with increase in transistor density. This has lead to the creation of hotspots, which are regions with very high temperature. Excessive temperature at hotspots can lead to reliability and performance issues and affect design convergence. In this work, we propose a run-time look-ahead based task migration technique in order to utilize the multitude of cores available in an MPSoC to eliminate thermal hotspots. Our technique is based on temperature prediction leveraging upon a novel wavelet based thermal modeling approach. We also present a static ILP based scheduling approach which minimizes the total schedule length of a task graph, while preventing thermal emergencies. The results show that the dynamic task scheduling for most of the cases results in 2% to 5% smaller schedule lengths as compared to that obtained from frequency scaling and a maximum schedule length increase of 5% to 10% of the absolute value obtained from ILP based approach.

It can be seen that all the above problems involve optimization of some real physical quantity in the presence of constraints in various domains. These constraints are in the Boolean domain for crosstalk ATPG and switching current estimation problems at hardware level, while they involve integer and dependency constraints for dynamic task scheduling and thermal ware scheduling problems in software level. All of the above problems can be mapped to the class of constrained max-satisfiability problems, which are $NP - Hard$ in nature. This work provides and compares various heuristic/exact solutions to the above problems. The results show that ILP based exact solution leads to large computation time. Consequently, heuristic solutions provide a good trade-off between quality of the solution and the computation complexity.

PUBLICATIONS

- K. Ganeshpure, A. Sanyal, and Kundu, S. "A Pattern Generation Technique for Maximizing Switching Supply Currents Considering Gate Delays,". IEEE Transactions on Computers 99, PrePrints (2011).

- K. Ganeshpure, S. Kundu, "On Run-time Task Graph Extraction of SoC," SoC Design Conference (ISOCC), 2010 International

- K. Ganeshpure, S. Kundu, "On ATPG for Multiple Aggressor Crosstalk Faults," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.29, no.5, pp.774-787, May 2010

- K. Ganeshpure, S. Kundu, "On ATPG for Multiple Aggressor Crosstalk Faults in Presence of Gate Delays," International Test Conference, pp.1-7, 2007

- K. Ganeshpure, S. Kundu, "An ILP Based ATPG Technique for Multiple Aggressor Crosstalk Faults Considering the Effects of Gate Delays," VLSI Design conference 2009, , 233-238

- Kunal Ganeshpure, Sandip Kundu, "Automatic Test Pattern Generation for Maximal Circuit Noise in Multiple Aggressor Cross-Talk Faults," DATE, pp. 540-545, 2007

- K. Ganeshpure, A. Sanyal, and Kundu, S. "A Pattern Generation Technique for Maximizing Power Supply Currents," IEEE International Conference on Computer Design (ICCD) (2006), 387392.

BIBLIOGRAPHY

[1] "AMD Phenom Processors,". *http://www.amd.com*.

[2] "Android,". *http://www.android.com*.

[3] "Dell PowerEdge Server,". *http://www.dell.com/downloads/global/products/pedge/en/2800_specspdf*.

[4] "Failure Mechanisms and Models for Semiconductor Devices,". In *JEDEC publication JEP122C http://www.jedec.org*.

[5] "GNU Linear Programming Kit,". *http://www.gnu.org/software/glpk/*.

[6] "HSPICE Users Manual,". *http://www.synopsys.com/Tools/Verification/AMS-Verification/CircuitSimulation/HSPICE/Pages/default.aspx*.

[7] "ILOG CPLEX,". *http://www.ilog.com/products/cplex*.

[8] "Qualcomm Snapdragon,". *http://www.qualcomm.com/snapdragon/specs*.

[9] A. Krstic, Y. Jiang, and Cheng, K. "Pattern Generation for Delay Testing and Dynamic Timing Analysis Considering Power Supply Noise Effects,". *IEEE Transactions on Computer-Aided Design 20*, 3 (March 2001), 416–425.

[10] Addison, Paul S. *"The Illustrated Wavelet Transform Handbook,"*. Instutute of Physics Publishing Bristol and Pheladelphia.

[11] Ahmad, I., Ranka, S., and Khan, S.U. "Using Game Theory for Scheduling Tasks on Multi-Core Processors for Simultaneous Optimization of Performance and Energy,". In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* (2008), pp. 1 –6.

[12] Ajami, A.H., Banerjee, K., and Pedram, M. "Modeling and Analysis of Non-uniform Substrate Temperature Effects on Global ULSI Interconnects,". *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 24*, 6 (2005), 849 – 861.

[13] Andrew S. Tanenbaum, Maarten Van Steen. *"Distributed Systems: Principles and Paradigms,"*, 2 ed. Prentice Hall, 2008.

[14] Ayoub, Raid Zuhair, and Rosing, Tajana Simunic. "Predict and Act: Dynamic Thermal Management for Multi-core Processors,". In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design* (New York, NY, USA, 2009), ISLPED '09, ACM, pp. 99–104.

[15] Bai, Xiaoliang, Dey, S., and Krstic, A. "HYAC: A Hybrid Structural SAT Based ATPG for Crosstalk,". *Test Conference, 2003. Proceedings. ITC 2003. International 1* (30-Oct. 2, 2003), 112–121.

[16] Brooks, David, and Martonosi, Margaret. "Dynamic Thermal Management for High-Performance Microprocessors,". In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2001), HPCA '01, IEEE Computer Society, pp. 171–.

[17] Bushnell, M.L., and Agrawal, V.D. *"Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits,"*. Frontiers in electronic testing. Kluwer Academic, 2000.

[18] C. Tirumurti, S. Kundu, S. Sur-Kolay, and Chang, Y. "A Modeling Approach for Addressing Power Supply Switching Noise Related Failures of Integrated Circuits,". *Proceeding of Design, Automation and Test in Europe (DATE)* (2004), 1078–1083.

[19] Chai, D., and Kuehlmann, A. "Circuit-based Preprocessing of ILP and its Applications in Leakage Minimization and Power Estimation,". *IEEE International Conference on Computer Design* (2004), 387–392.

[20] Chai, D., and Kuehlmann, A. "A Fast Pseudo-Boolean Constraint Solver,". *IEEE Transactions on Computer-Aided Design 24*, 3 (March 2005), 305–317.

[21] Chakraborty, Koushik, Wells, Philip M., and Sohi, Gurindar S. "Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly,". In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ASPLOS-XII, ACM, pp. 283–292.

[22] Chantem, Thidapat, Dick, Robert P., and Hu, X. Sharon. "Temperature-aware Scheduling and Assignment for Hard Real-time Applications on MPSoCs,". In *Proceedings of the conference on Design, automation and test in Europe* (New York, NY, USA, 2008), DATE '08, ACM, pp. 288–293.

[23] Chen, Pinhong, and Keutzer, Kurt. "Towards True Crosstalk Noise Analysis,". In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design* (Piscataway, NJ, USA, 1999), IEEE Press, pp. 132–138.

[24] Chen, Wei-Yu, Gupta, Sandeep K., and Breuer, Melvin A. "Test Generation for Crosstalk-Induced Delay in Integrated Circuits,". In *ITC '99: Proceedings of the 1999 IEEE International Test Conference* (Washington, DC, USA, 1999), IEEE Computer Society, p. 191.

[25] Chen, Weiyu, Breuer, Melvin A., and Gupta, Sandeep K. "Analytic Models for Crosstalk Delay and Pulse Analysis Under Non-Ideal Inputs,". In *Proceedings of the IEEE International Test Conference* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 809–818.

[26] Chen, Weiyu, Gupta, Sandeep K., and Breuer, Melvin A. "Test Generation in VLSI Circuits for Crosstalk Noise,". In *ITC '98: Proceedings of the 1998 IEEE International Test Conference* (Washington, DC, USA, 1998), IEEE Computer Society, p. 641.

[27] Chen, Ya-Shu, Shih, Chi-Sheng, and Kuo, Tei-Wei. "Dynamic Task Scheduling and Processing Element Allocation for Multi-Function SoCs,". In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. 13th IEEE* (2007), pp. 81 –90.

[28] Chowdhury, S., and Barkatullah, J. "Estimation of Maximum Currents in MOS IC Logic Circuits,". *IEEE Transactions on Computer-Aided Design 9*, 6 (June 1990), 642–654.

[29] Cochran, Ryan, and Reda, Sherief. "Consistent Run-time Thermal Prediction and Control Through Workload Phase Detection,". In *Proceedings of the 47th Design Automation Conference* (New York, NY, USA, 2010), DAC '10, ACM, pp. 62–67.

[30] Cong, Jason, Liu, Bin, and Zhang, Zhiru. "Scheduling with Soft Constraints,". In *Proceedings of the 2009 International Conference on Computer-Aided Design* (New York, NY, USA, 2009), ICCAD '09, ACM, pp. 47–54.

[31] Coskun, A.K., Rosing, T.S., and Whisnant, K. "Temperature Aware Task Scheduling in MPSoCs,". In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07* (2007), pp. 1 –6.

[32] Coskun, Ayse Kivilcim, Rosing, Tajana Simunic, Whisnant, Keith A., and Gross, Kenny C. "Temperature-Aware MPSoC Scheduling for Reducing Hotspots and Gradients,". In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference* (Los Alamitos, CA, USA, 2008), ASP-DAC '08, IEEE Computer Society Press, pp. 49–54.

[33] D. Suryanarayana, R. Hsiao, T. Gall, and McCreary, J. "Enhancement of Flip-Chip Fatigue Life by Encapsulation,". *IEEE Transactions on Components, Packaging, and Manufacturing Technology 14*, 1 (March 1991), 218–223.

[34] Denning, Peter J. "The Working Set Model for Program Behavior,". *Commun. ACM 11* (May 1968), 323–333.

[35] Dhodapkar, Ashutosh S., and Smith, James E. "Managing Multi-configuration Hardware via Dynamic Working set Analysis,". *SIGARCH Comput. Archit. News 30* (May 2002), 233–244.

[36] Dick, R.P., Rhodes, D.L., and Wolf, W. "TGFF: Task Graphs For Free,". In *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on* (Mar. 1998), pp. 97 –101.

[37] Een, N., and Sorensson, N. "An Extensible SAT-Solver,". *International Conference on Theory and Applications of Satisfiability Testing (SAT)* (2003), 502–508.

[38] F. Aloul, A. Ramani, I. Markov, and Sakallah, K. "Generic ILP versus Specialized 0-1 ILP: an Update,". *International conference on Computer Aided Design* (2002), 450–457.

[39] Ferzli, Imad A., Chiprout, Eli, and Najm, Farid N. "Verification and Co-design of the Package and Die Power Delivery System using Wavelets,". *Trans. Comp.-Aided Des. Integ. Cir. Sys. 29* (January 2010), 92–102.

[40] Fiduccia, C. M., and Mattheyses, R. M. "A Linear-Time Heuristic for Improving Network Partitions,". In *DAC '82: Proceedings of the 19th conference on Design automation* (Piscataway, NJ, USA, 1982), IEEE Press, pp. 175–181.

[41] Fortet, R. "Applications de l'algebre de Boole en Recherche Operationelle,". *Revue Francaise de Recherche Operationelle, 1960,Vol. 4 4* (1960), 17–26.

[42] Ganeshpure, K., and Kundu, S. "On ATPG for Multiple Aggressor Crosstalk Faults,". *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 29*, 5 (May 2010), 774–787.

[43] Ganeshpure, K. P., and Kundu, S. "Automatic Test Pattern Generation for Maximal Circuit Noise in Multiple Aggressor Crosstalk Faults,". *Design, Automation and Test in Europe Conference and Exhibition 0* (2007), 105.

[44] Ganeshpure, K. P., and Kundu, S. "An ILP Based ATPG Technique for Multiple Aggressor Crosstalk Faults Considering the Effects of Gate Delays,". 233–238.

[45] Ganeshpure, Kunal, and Kundu, Sandip. "On ATPG for Multiple Aggressor Crosstalk Faults in presence of Gate Delays,". *Test Conference, 2007. ITC 2007. IEEE International* (Oct. 2007), 1–7.

[46] Ganeshpure, Kunal, and Kundu, Sandip. "On Run-time Task Graph Extraction of SoC,". pp. 380–383.

[47] Gao, D.S., Yang, A.T., and Kang, S.M. "Modeling and Simulation of Interconnection Delays and Crosstalks in High-speed Integrated Circuits,". *Circuits and Systems, IEEE Transactions on 37*, 1 (Jan 1990), 1–9.

[48] Gupta, S., and Najm, F. "Energy and Peak-Current Per-Cycle Estimation at RTL,". *IEEE Transactions on VLSI Systems 11*, 4 (August 2003), 525–537.

[49] H. Kriplani, F. N. Najm, and Hajj, I. N. "Pattern Independent Maximum Current Estimation in Power and Ground Buses of CMOS VLSI Circuits: Algorithms, Signal Correlations, and their Resolution,". *IEEE Transactions on Computer-Aided Design 14*, 8 (August 1995), 998–1012.

[50] Hauck, S., and Borriello, G. "An Evaluation of Bi-partitioning Techniques,". *Advanced Research in VLSI, 1995. Proceedings., Sixteenth Conference on* (Mar 1995), 383–402.

[51] Hrishikesh, M.S., Jouppi, N.P., Farkas, K.I. Burger, D., Keckler, S.W., and Shivakumar, P. "The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays,". pp. 14–24.

[52] Huang, Wei, Stan, Mircea R., Skadron, Kevin, Sankaranarayanan, Karthik, Ghosh, Shougata, and Velusam, Sivakumar. "Compact Thermal Modeling for Temperature-aware Design,". In *Proceedings of the 41st annual Design Automation Conference* (New York, NY, USA, 2004), DAC '04, ACM, pp. 878–883.

[53] Isci, Canturk, Martonosi, Margaret, and Buyuktosunoglu, Alper. "Long-term Workload Phases: Duration Predictions and Applications to DVFS,". *IEEE Micro: Special Issue on Energy Efficient Design 25* (2005), 39–51.

[54] Jagau, U. "SIMCURRENT-An Efficient Program for the Estimation of the Current Flow of Complex CMOS Circuits,". *IEEE International Conference on Computer-Aided Design* (1988), 208–211.

[55] Jiang, Y., and Cheng, K. "Vector Generation for Power Supply Noise Estimation and Verification of Deep Submicron Designs,". *IEEE Transactions on VLSI Systems 9*, 2 (April 2001), 329–340.

[56] K. Ganeshpure, A. Sanyal, and Kundu, S. "A Pattern Generation Technique for Maximizing Power Supply Currents,". *IEEE International Conference on Computer Design (ICCD)* (2006), 387–392.

[57] K. Ganeshpure, A. Sanyal, and Kundu, S. "A Pattern Generation Technique for Maximizing Switching Supply Currents Considering Gate Delays,". *IEEE Transactions on Computers 99*, PrePrints (2011).

[58] K. Roy, S. Mukhopadhyay, and Mahmoodi-Meimand, H. "Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicron CMOS Circuits,". *Proceedings of the IEEE 91*, 2 (2003), 305–327.

[59] Kahng, Andrew, Liu, Bao, and Xu, Xu. "Constructing Current-Based Gate Models Based on Existing Timing Library,". In *International Symposium on Quality Electronic Design* (March 2006), pp. 37–42.

[60] Khan, Omer. "A Hardware/Software Co-Design Architecture for Thermal, Power, and Reliability Management in Chip Multiprocessors,". *Open Access Dissertations. Paper 169* (2010).

[61] Klaiber, A. "The Technology Behind Crusoe Processors,". *Transmeta white paper* (January 2000).

[62] Krishna, C. M. *"Real-Time Systems,"*, 1st ed. McGraw-Hill Higher Education, 1996.

[63] Krstic, A., Liou, Jing-Jia, Jiang, Yi-Min, and Cheng, Kwang-Ting. "Delay Testing Considering Crosstalk-Induced Effects,". *Test Conference, 2001. Proceedings. International* (2001), 558–567.

[64] Kundu, R., and Blanton, R.D. "Timed Test Generation for Crosstalk Switch Failures in Domino CMOS,". *VLSI Test Symposium, 2002. (VTS 2002). Proceedings 20th IEEE* (2002), 379–385.

[65] Kundu, S., Zachariah, S.T., Chang, Yi-Shing, and Tirumurti, C. "On Modeling Crosstalk Faults,". *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 24*, 12 (December 2005), 1909–1915.

[66] Larrabee, Tracy. "Test Pattern Generation using Boolean Satisfiability,". *IEEE Transactions on Computer-Aided Design 11* (1992), 4–15.

[67] Lawler, E.L. "Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints,". In *Algorithmic Aspects of Combinatorics*, P. Hell B. Alspach and D.J. Miller, Eds., vol. 2 of *Annals of Discrete Mathematics*. Elsevier, 1978, pp. 75 – 90.

[68] Lee, H.K., and Ha, D.S. "ATALANTA: An Efficient ATPG for Combinational Circuits,". *Technical Report, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia* (1993), 93–12.

[69] Lee, Kyung Tek, Nordquist, C., and Abraham, J.A. "Automatic Test Pattern Generation for Crosstalk Glitches in Digital Circuits,". *VLSI Test Symposium, 1998. Proceedings. 16th IEEE* (Apr 1998), 34–39.

[70] Li, Wing Ning, Reddy, Sudhakar M., and Sahni, Sartaj. "On Path Selection in Combinational Logic Circuits,". In *DAC '88: Proceedings of the 25th ACM/IEEE conference on Design automation* (Los Alamitos, CA, USA, 1988), IEEE Computer Society Press, pp. 142–147.

[71] Liu, Ai-Hsin, and Dick, Robert P. "Automatic Run-time Extraction of Communication Graphs from Multithreaded Applications,". In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis* (New York, NY, USA, 2006), CODES+ISSS '06, ACM, pp. 46–51.

[72] Lu, Zhijian, Huang, Wei, Ghosh, Shougata, Lach, John, Stan, Mircea, and Skadron, Kevin. "Analysis of Temporal and Spatial Temperature Gradients for IC Reliability,". *CS-2004-08* (March 2004).

[73] M. Gschwind, P. Hofstee, B. Flachs M. Hopkins Y. Watanabe, and Yamazaki, T. "A Novel SIMD Architecture for the Cell Heterogeneous Chip-Multiprocessor,". *Hot Chips 17* (August 2005).

[74] M. Hsiao, E. Rudnick, and Patel, J. "Peak Power Estimation of VLSI Circuits: New Peak Power Measures,". *IEEE Transactions on VLSI Systems 8*, 4 (2000), 435–439.

[75] Manich, S., and Figueras, J. "Maximizing the Weighted Switching Activity in Combinational CMOS Circuits under Variable Delay Model,". *European Design and Test Conference 0* (1997), 597.

[76] Mogul, J.C., Mudigonda, J., Binkert, N., Ranganathan, P., and Talwar, V. "Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems,". *Micro, IEEE 28*, 3 (may-june 2008), 26–41.

[77] Murali, Srinivasan, Mutapcic, Almir, Atienza, David, Gupta, Rajesh, Boyd, Stephen, Benini, Luca, and De Micheli, Giovanni. "Temperature Control of High-performance Multi-core Platforms using Convex Optimization,". In *Proceedings of the conference on Design, automation and test in Europe* (New York, NY, USA, 2008), DATE '08, ACM, pp. 110–115.

[78] Nabavi-Lishi, A., and Rumin, N. "Delay and Bus Current Evaluation in CMOS Logic Circuits,". *IEEE International Conference on Computer-Aided Design* (August 1992), 198–203.

[79] Najm, F., and Zhang, M. "Extreme Delay Sensitivity and the Worst-case Switching Activity in VLSI Circuits,". *Design Automation Conference* (1995), 623–627.

[80] Nash, John. "Non-Cooperative Games,". *The Annals of Mathematics 54*, 2 (1951), pp. 286–295.

[81] Omer Khan, Sandip Kundu. "Microvisor: A Runtime Architecture for Thermal Management in Chip Multiprocessors,". *Transactions on High-Performance Embedded Architectures and Compilers* (2009).

[82] Ortego, P. Sack. "SESC: Super ESCalar Simulator,". In *Technical Report* (2004).

[83] Paul, Bipul C, and Roy, Kaushik. "Testing Crosstalk Induced Delay Faults in Static CMOS Circuits Through Dynamic Timing Analysis,". *Test Conference, International* (2002), 384.

[84] Proakis, J.G., and Manolakis, D.G. *"Digital Signal Processing,"*. Pearson Prentice Hall, 2007.

[85] Puschini, D., Clermidy, F., Benoit, P., Sassatelli, G., and Torres, L. "Game-Theoretic Approach for Temperature-Aware Frequency Assignment with Task Synchronization on MPSoC,". In *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on* (dec. 2008), pp. 235 –240.

[86] Puschini, D., Clermidy, F., Benoit, P., Sassatelli, G., and Torres, L. "Temperature-Aware Distributed Run-Time Optimization on MPSoC Using Game Theory,". In *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual* (april 2008), pp. 375 –380.

[87] Q. Wu, Q. Qiu, and Pedram, M. "Estimation of Peak Power Dissipation in VLSI Circuits using the Limiting Distributions of Extreme Order Statistics,". *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 20*, 8 (2001), 942–956.

[88] Rabaey, Jan M., Chandrakasan, Anantha, and Nikolic, Borivoje. *"Digital Integrated Circuits,"*, 2 ed. Prentice Hall, January 2003.

[89] Rao, V., Navet, N., Singhal, G., Kumar, A., and Visweswaran, G.S. "Battery Aware Dynamic Scheduling for Periodic Task Graphs,". In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*.

[90] S. Devadas, K. Keutzer, and White, J. "Estimation of Power Dissipation in CMOS Combinational Circuits Using Boolean Function Manipulation,". *IEEE Transactions on Computer-Aided Design 11*, 3 (March 1992), 373–383.

[91] S. Srinivasan, K. Ganeshpure, S. Kundu. "Maximizing Hotspot Temperature: Wavelet based Modelling of Heating and Cooling Profile of Functional Workloads,". *Quality Electronic Design (ISQED), 2011 12th International Symposium on* (March 2011), 1–7.

[92] Sagahyroon, A., and Aloul, F. "Using SAT-based Techniques in Power Estimation,". *Microelectronics Journal 38*, 6 (2007), 706–715.

[93] Sanyal, A., Ganeshpure, K., and Kundu, S. "An Improved Soft-Error Rate Measurement Technique,". *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 28*, 4 (April 2009), 596–600.

[94] Sasaki, Y., and De Micheli, G. "Crosstalk Delay Analysis using Relative Window Method,". *ASIC/SOC Conference, 1999. Proceedings. Twelfth Annual IEEE International* (1999), 9–13.

[95] Sasaki, Y., and Yano, K. "Multi-Aggressor Relative Window Method for Timing Analysis Including Crosstalk Delay Degradation,". *Custom Integrated Circuits Conference, 2000. CICC. Proceedings of the IEEE 2000* (2000), 495–498.

[96] Sherwood, T., Sair, S., and Calder, B. "Phase Tracking and Prediction,". In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on* (june 2003), pp. 336 – 347.

[97] Shimizu, Kazuya, Itazaki, Noriyoshi, and Kinoshita, Kozo. "Built-In Self-Test for Crosstalk Faults in a Digital VLSI,". *Systems and Computers in Japan, 2002, Vol. 33, No. 13* (2002), 35–47.

[98] Shivakumar, Premkishore, Jouppi, Norman P., and Shivakumar, Premkishore. "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model,". Tech. rep., 2001.

[99] Smith, James E., and Dhodapkar, Ashutosh S. "Dynamic Microarchitecture Adaptation via Co-designed Virtual Machines,". *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International 1* (Feb 2002), 198–199.

[100] Spencer, R. R., and Ghausi, M. S. *"Introduction to Electronic Circuit Design,"*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[101] Stan, Mircea R., Skadron, Kevin, Barcella, Marco, Huang, Wei, Sankaranarayanan, Karthik, and Velusamy, Sivakumar. "Hotspot: A Dynamic Compact Thermal Model at the Processor Architecture Level,". *Microelectronics Journal 34* (2003), 1153–1165.

[102] Tayade, R., and Abraham, J.A. "Critical Path Selection for Delay Test Considering Coupling Noise,". pp. 119–124.

[103] Theocharides, Theocharis, Michael, Maria K., Polycarpou, Marios, and Dingankar, Ajit. "Towards Embedded Run-time System Level Optimization for MPSoCs: On-Chip Task Allocation,". In *Proceedings of the 19th ACM Great Lakes symposium on VLSI* (New York, NY, USA, 2009), GLSVLSI '09, ACM, pp. 121–124.

[104] Tiwari, Vivek, Singh, Deo, Rajgopal, Suresh, Mehta, Gaurav, Patel, Rakesh, and Baez, Franklin. "Reducing Power in High-Performance Microprocessors,". In *Proceedings of the 35th annual Design Automation Conference* (New York, NY, USA, 1998), DAC '98, ACM, pp. 732–737.

[105] Tosun, Suleyman, Mansouri, Nazanin, Kandemir, Mahmut, and Ozturk, Ozcan. "An ILP Formulation for Task Scheduling on Heterogeneous Chip Multiprocessors,". In *Computer and Information Sciences ISCIS 2006*, Albert Levi, Erkay Savas, Hsn Yenign, Selim Balcisoy, and Ycel Saygin, Eds., vol. 4263 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 267–276.

[106] Vallerio, K.S., and Jha, N.K. "Task Graph Extraction for Embedded System Synthesis,". In *VLSI Design, 2003. Proceedings. 16th International Conference on* (2003), pp. 480 – 486.

[107] Wang, C., and Roy, K. "Maximum Power Estimation for CMOS Circuits using Deterministic and Statistic Approaches,". *International Conference on VLSI Design* (1995), 364–369.

[108] Wang, Yi, Liu, Duo, Wang, Meng, Qin, Zhiwei, and Shao, Zili. "Optimal Task Scheduling by Removing Inter-Core Communication Overhead for Streaming Applications on MPSoC,". In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium* (Washington, DC, USA, 2010), RTAS '10, IEEE Computer Society, pp. 195–204.

[109] Werner, C., Göttsche, R., Wörner, A., and Ramacher, U. "Crosstalk Noise in Future Digital CMOS Circuits,". In *DATE '01: Proceedings of the conference on Design, automation and test in Europe* (Piscataway, NJ, USA, 2001), IEEE Press, pp. 331–335.

[110] Weste, N. H., and Harris, D. *"CMOS VLSI Design: A Circuits and Systems Perspective,"*, 2 ed. Addison Wesley, January 2005.

[111] Y. Jiang, A. Krstic, and Cheng, K. "Estimation for Maximum Instantaneous Current Through Supply Lines for CMOS Circuits,". *IEEE Transactions on VLSI Systems 8*, 1 (February 2000), 61–73.