Summer 2015

Common path pessimism removal in static timing analysis

Chunyu Wang

COMMON PATH PESSIMISM REMOVAL

IN

STATIC TIMING ANALYSIS

by

CHUNYU WANG

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER ENGINEERING

2015

Approved by

Yiyu Shi, Advisor
Minsu Choi
Jun Fan

ABSTRACT

Static timing analysis is a key process to guarantee timing closure for modern IC designs. However, additional pessimism can significantly increase the difficulty to achieve timing closure. Common path pessimism removal (CPPR) is a prevalent step to achieve accurate timing signoff. To speed up the existing exhaustive exploration on all paths in a design, this thesis introduces a fast multi-threading timing analysis for removing common path pessimism based on block-based static timing analysis. Experimental results show that the proposed method has faster runtime in removing excess pessimism from clock paths.

ACKNOWLEDGMENTS

My deepest gratitude goes first and foremost to my advisor Dr. Shi. Thank you for his constant encouragement and guidance.

Second, I would like to express my heartful gratitude to my committee members: Dr. Choi and Dr. Fan, who helped and instructed me a lot.

I also owe my sincere appreciation to my friends and lab mates, who gave me their help and time.

Finally, my thanks go to my beloved boyfriend Qifeng Chen and my family, who gave me loving consideration and great confidence.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

Page

1. INTRODUCTION

Static timing analysis (STA) is a process that verifies the timing performance of a design under worst-case conditions. In the modern IC design flow, STA is essential to identify timing critical paths for subsequent optimization, to determine operable clock frequencies, to prevent over-design, and to achieve design closure, while meeting stringent timing constraints. Rapid growing design complexities and increasing on-chip variations, however, complicate this analysis for nanometer design. These on-chip variations, including manufacturing process, voltage and temperature (PVT) variations, affect wire delays and gate delays in different portions of a chip. Although statistical timing analysis and multi-corner timing analysis have been proposed to handle these variations, not all sources of variability are accurately modeled.

To capture more accurate timing performance of a design, common path pessimism removal (CPPR) is prevalent to eliminate inherent but artificial pessimism in clock paths during timing analysis. For example, applying different conditions for the common part of the launch and capture clock paths is over pessimistic. This pessimism should be removed during timing analysis so that true critical paths can then be identified.

The challenge of CPPR is that the amount of pessimism to be removed is path-dependent. Existing solutions fall into two categories, critical-path-based approach and exhaustive search approach. The critical-path-based approach first identifies critical paths without CPPR consideration and then re-evaluates these identified paths with CPPR. Because the criticality of paths may be altered after CPPR consideration, this approach may miss true critical paths and generate optimistic results. In contrast, the exhaustive search generates the accurate solution by exploring all paths. The total number of paths has exponential growth with the circuit size. Thus, the exhaustive search approach is time-consuming especially for modern large-scale designs. To speed up the path retrieval, this thesis adopts a multi-threaded depth-first-search (DFS) method to achieve the exhaustive search approach of Common Path Pessimism Removal (CPPR).

The remainder of this paper is organized as follows: Section II introduces the problem formulation. Section III details timing analysis framework. Section IV shows experimental results. Finally, Section V concludes this work.

2. PRELIMINARIES AND PROBLEM FORMULATION

This section is organized as followed, it will talk about the static timing analysis (STA) in Subsection 2.1, in which will outline timing propagation in Subsection 2.1.1, and it will describe delay modeling in Subsection 2.1.2. Subsection 2.2 will cover the properties and the details of CPPR formulation.

2.1. STATIC TIMING ANALYSIS (STA)

A static timing analysis of a design typically provides a profile of the design's performance by measuring the timing propagation from inputs to outputs. Timing analysis computes the amount of time signals propagate in a circuit from its primary inputs (PIs) to its primary outputs (POs) through various circuit elements and interconnect. Signals arriving at an input of an element will be available at its output(s) at some later time. Each element therefore introduces a delay during signal propagation.

A signal transition is characterized by its input slew and output slew, which is defined as the amount of time required for a signal to transition from high-to-low or low-to-high. To account for timing modeling limitations in considering design and electrical complexities, as well as multiple sources of variability, such as manufacturing variations, temperature fluctuation and voltage drops, timing analysis is typically done using an early-late split, where each circuit node has an early (lower) bound and a late (upper) bound on its time. By convention, if the mode is not explicitly specified, both modes should be considered. Both slew and delay are computed separately on early and late modes. For example, in early mode, an output slew s_o^E is computed using the input slew taken from the early mode s_i^E, and similarly, in late mode, the output slew s_o^L is computed using s_i^L.

2.1.1. Timing Propagation. Starting from the primary input(s), the instant that a signal reaches an input or output of a circuit element is quantified as the Arrival Time (at). Similarly, starting from the primary output(s), the limits imposed for each arrival time to ensure proper circuit operation is quantified as the Required Arrival Time (rat). Given an arrival time and a required arrival time, the Slack at a circuit node quantified how well timing constraints are met. That is, a positive slack means the required time is satisfied, and a negative slack means the required time is in violation.

Actual arrival time is defined as starting from the primary inputs, arrival times (at) are computed by adding delays across a path, and performing the minimum (in early mode) or maximum (in late mode) of such accumulated times at a convergence point. For example, let $at^E(A)$ and $at^E(B)$ denote the early arrival times at pins A and B, respectively, in Figure 2.1. The most pessimistic early mode arrival time at the output pin Y is:

$$at^E(Y) = \min(at^E(A) + d^E(A,Y), at^E(B) + d^E(B,Y)) \tag{1}$$

Conversely, in late mode, the latest time that a signal transition can reach any given circuit node is computed. Following the same example in Figure 2.1, the most pessimistic late mode arrival time at Y is:

$$at^L(Y) = \max(at^L(A) + d^L(A,Y), at^L(B) + d^L(B,Y)) \tag{2}$$

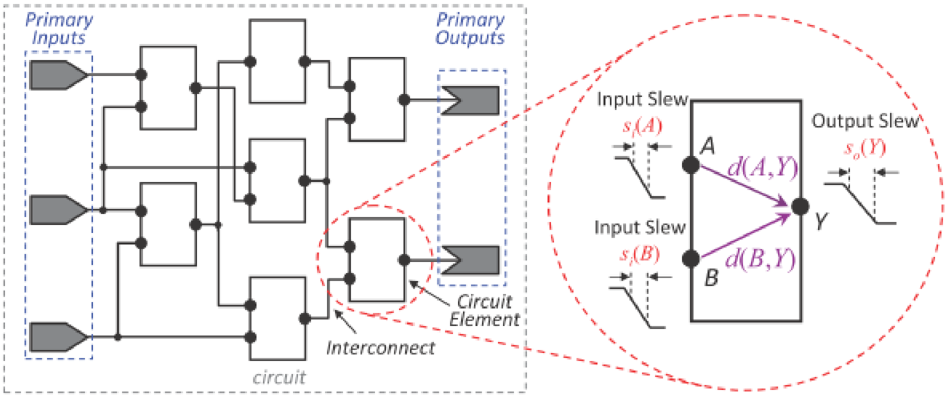

Figure 2.1. Delay Model Representation of A Circuit Element

Required arrival time is defined as starting from the primary outputs, required arrival times (rat) are computed by subtracting the delays across a path, and performing the maximum (minimum) in early (late) mode of such accumulated times at a convergence point. For example, a generic interconnect (input Y, output A, B), the most pessimistic early mode required arrival time at the output pin Y is:

$$rat^E(Y) = \max(rat^E(A) - d^E(Y,A), rat^E(B) - d^E(Y,B)) \tag{3}$$

Conversely, in late mode, the earliest time that a signal transition must reach a given circuit node is computed. Following the same example, the most pessimistic late mode required arrival time at the input pin Y is:

$$rat^L(Y) = \min(rat^L(A) - d^L(Y,A), rat^L(B) - d^L(Y,B)) \tag{4}$$

Slack, for proper circuit operation, must hold:

$$at^E \geq rat^E \tag{5}$$

$$at^L \leq rat^L \tag{6}$$

To quantify how well timing constraints are met at each circuit node, slacks can be computed based on Equations 5 and 6. That is, slacks are positive when the required times are met, and negative otherwise.

$$slack^E = at^E - rat^E \tag{7}$$

$$slack^L = rat^L - at^L \tag{8}$$

Slew propagation is defined as circuit element delays and interconnect delays are functions of input slew (s_i), subsequent output slew (s_o) must be propagated. This thesis assumes worst-slew propagation, that is, this thesis propagate the smallest (largest) slew in early (late) mode. Following the example in Figure 2.1, the early mode and late output slew at output pin Y are, respectively:

$${s_o}^E(Y) = \min({s_o}^E(A,Y), {s_o}^E(B,Y)) \tag{9}$$

$${s_o}^L(Y) = \max({s_o}^L(A,Y), {s_o}^L(B,Y)) \tag{10}$$

2.1.2. Circuit Element Modeling. This Subsection will discuss combinational and sequential circuit elements.

In a given combinational cell, for example, OR gate, let the delay d and output slew s_o, for a input/output pin-pair in Figure 2.2 be

$$\mathrm{d} = \mathrm{a} + \mathrm{b} \cdot C_L + c \cdot s_i \tag{11}$$

$$s_o = \mathrm{x} + \mathrm{y} \cdot C_L + z \cdot s_i \tag{12}$$

Here, a, b, c, x, y and z are cell-dependent constants, C_L is the output load at the output pin. For simplicity, this thesis assumes C_L to be sum of all capacitances in parasitic RC tree including cell pin capacitances at the taps.

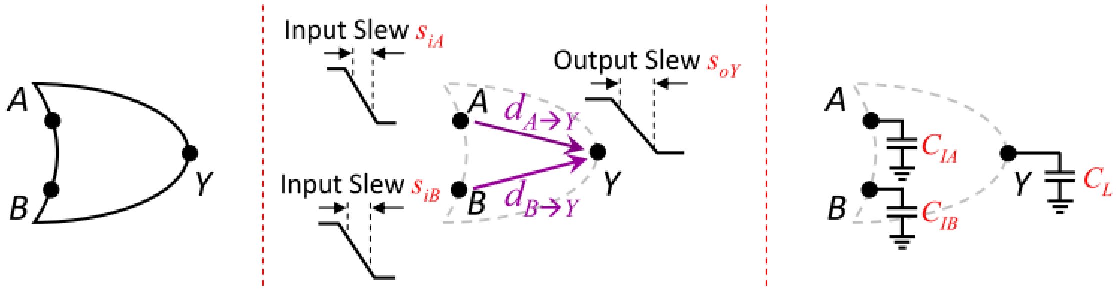

Figure 2.2. Combinational OR Gate, Timing Model and Capacitances

Sequential circuits consist of combinational blocks interleaved by registers, usually implemented with flip-flops (FFs). Typically, sequential circuits are composed of several stages, where a register captures data from the outputs of a combinational block from a previous stage, and injects it into the inputs of the combinational block in the next stage. Register operation is synchronized by clock signals generated by one or multiple clock sources. Clock signals that reach distinct flip-flops, for example, sinks in the clock tree, are delayed from the clock source by a clock latency l.

Set up and hold constraints. Proper operation of a flip-flop requires the logic value of the input data pin to be stable for a specific period of time before the capturing clock edge. This period of time is designated by the setup time t_{setup}. Additionally, the logic value of the input data pin must also be stable for a specific period of time after the capturing clock edge. This period of time is designated by the hold time t_{hold}. The flip-flop timing models are depicted in Figure 2.3.

Setup and hold constraints are respectively modeled as functions of the input slews at both the clock pin CK and the data input pin D as

$$t_{setup} = g + h \cdot s_{i}{}_{CK}^{E} + j \cdot s_{i}{}_{D}^{L} \tag{13}$$

$$t_{hold} = m + n \cdot s_{i}{}_{CK}^{L} + p \cdot s_{i}{}_{D}^{E} \tag{14}$$

Here, g, h, j, m, n and p are flop-specific parameters, $s_{i_{CK}}$ is the input slew at CK, and s_{i_D} is the input slew at D.

Signal propagation. Consider the standard signal transition between two flip-flops as illustrated in Figure 2.3. Assuming that the clock edge is generated at the source at time 0, it will reach the injecting (launching) flip-flop FF_1 at time l_i, making the data available at the input of the combinational block $d_{CK\to Q}$ time later. If the propagation delay in the combinational block is d_{comb}, then the data will be available at the input of the capturing flip-flop FF_2 at time $l_i + d_{CK\to Q} + d_{comb}$. Let the clock period to be a constant T. Then the next clock edge will reach FF_2 at time $T + l_o$. For correct operation, the data must be available at the input pin D of FF_2 t_{setup} time before the next clock edge. Therefore, at the data input pin D of FF_2, the arrival time and required arrival time are:

$$at_D^L = l_i^L + d_{CK\to Q} + d_{comb}^L \tag{15}$$

$$rat_{setup} = rat_D^L = T + l_o^E - t_{setup} \tag{16}$$

A similar condition can be derived for ensuring that the hold time is respected. The data input pin D of FF_2 must remain stable for at least t_{hold} time after the clock edge reaches the corresponding CK pin. Therefore, at the data input pin D of FF_2, the arrival time and required arrival time are:

$$at_D^E = l_i^E + d_{CK\to Q} + d_{comb}^E \tag{17}$$

$$rat_{hold} = rat_D^E = l_o^L + t_{hold} \tag{18}$$

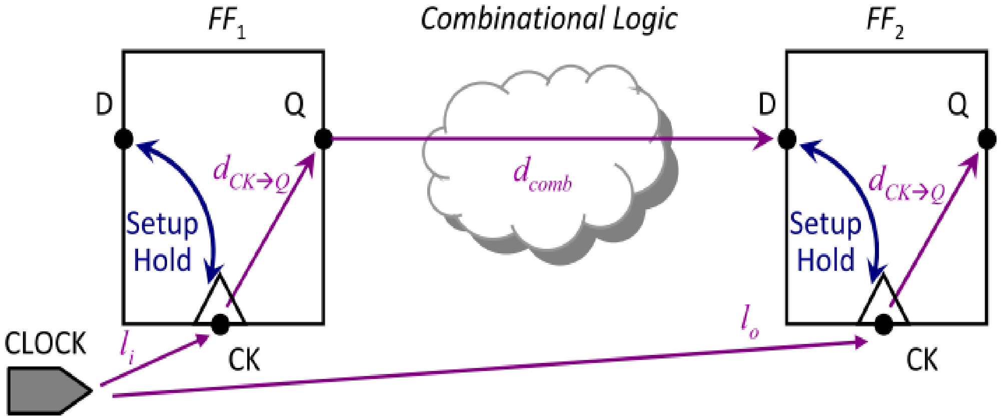

Figure 2.3. Two FFs in Series and Their Timing Models

2.2. COMMON PATH PESSIMISM REMOVAL (CPPR)

The early-late spilt-timing analysis inherently embeds unnecessary pessimism, which can lead to an over-conservative design. Analyze the example in Figure 2.4. The early (late) data's arrival time is compared with the late (early) clock's arrival time for the hold (setup) test. However, along the physically-common portion of the data path and clock path, the signal cannot simultaneously experience all the effects accounted for during early and late mode operation, for example, the signal cannot be both at high voltage and low voltage. Consequently, this unnecessary pessimism can lead to tests having negative slack. But in fact, they could be having positive slack. This unnecessary pessimism should thus be avoided when reporting final timing results.

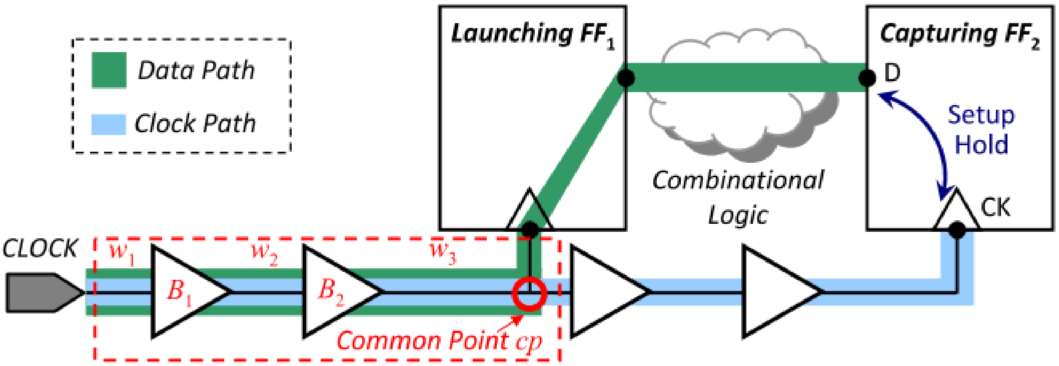

Figure 2.4. Clock Network Pessimism Incurs in the Common Path between the Launching Clock Path and the Capturing Clock Path

To the first order, the amount of pessimism for a given test can be approximated by the difference in the early and late arrival times at the common point. However, the common point is found by backwards tracking from the data and clock for the path with the worst slack. In the general case, there can be multiple paths converging at the data input of a flip-flop, and every path will have its own amount of undue pessimism. Therefore, to find the correct slack, we need to take the minimum slack found across all paths.

Hold tests. For tests that compare the data point against the clock point for the same cycle, for example, where data must be stable after the clock signal arrives at the capturing flip-flop, the total pessimism incurred is the difference between the early and late arrival times at the common point. That is, any on-chip variation incurred is spatially and temporally the same. For a hold test T_{hold} that has one data path that share common elements and common edges, the amount of credit given back is

$$credit^{hold} = at_{cp}^{L} - at_{cp}^{E} \tag{19}$$

Where CP is the last point before the data path and clock path diverge.

Setup tests. For tests that compare the data point against the clock point in subsequent cycles, for example, where the data must be stable before the clock signal arrives at the capturing flip-flop, the total pessimism incurred is the summation of the difference of early and late delays up through the common point. While the data and clock path share the same physical components, they are launched at different clock cycles. Therefore, if some on-chip variation, for example, temperature fluctuation occurs when the data is launched, but does not occur when the data is captured, the pessimism was now necessary, and cannot be removed. For a setup test T_{setup} that has one data path sharing common elements and common edges, the amount of credit given back is

$$credit^{setup} = \sum_{p \in P(V,E)} d_p^L - d_p^E \tag{20}$$

Where P(V,E) is the physically-common path between the data and clock paths. Here, V is the set of all common circuit elements, and E is the set of all common interconnect. Following the example in Figure 2.4, the amount of credit for the setup test is

$$credit^{setup} = \left(d_{w_1}^L - d_{w_1}^E\right) + \left(d_{B_1}^L - d_{B_1}^E\right) + \left(d_{w_2}^L - d_{w_2}^E\right) + \left(d_{B_2}^L - d_{B_2}^E\right) + \left(d_{w_3}^L - d_{w_3}^E\right) \tag{21}$$

Test credit or slack. As removing pessimism could require analyzing many paths, the post-CPPR test slack is the minimum slack of all paths that converge at the data point of the test.

Total test credit. As removing pessimism for each test could require investigating multiple paths, the amount of credit per test will be defined as the difference between the post-CPPR and pre-CPPR test slack. This thesis only considers hold and setup tests.

$$credit_{test}^{setup} = slack_{post-CPPR}^{setup} - slack_{pre-CPPR}^{setup} \tag{22}$$

$$credit_{test}^{hold} = slack_{post-CPPR}^{hold} - slack_{pre-CPPR}^{hold} \tag{23}$$

As mentioned above, common path pessimism removal is prevalent to eliminate artificially induced pessimism in clock paths during timing analysis. The common path pessimism removal problem can be formulated as: given a circuit with delay information, timing constraints, type of test, the required number of critical tests and the required number of critical paths, this thesis is to calculate the post-CPPR test slack.

3. TIMING ANALYSIS FRAMEWORK FOR CPPR

Section 3 is organized as followed, Subsection 3.1 will cover the concepts, motivations and Pthreads programming, Subsection 3.2 will cover the algorithm of accomplishing CPPR problem.

3.1. OVERVIEW AND CONCEPTS

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. For example, a problem is broken into discrete parts that can be solved concurrently. The main reasons to use parallel computing are as follows: (1) save time. In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. (2) solve larger/more complex problem. Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

In shared memory multiprocessor architectures, threads can be used to implement parallel. A thread is defined as an independent stream of instructions that can be scheduled to run. That is, a procedure that runs independently from its main program. In Figure 3.1, it conceptually shows that threads are within the same process address space, thus, much of the information present in the memory description of the process can be shared across threads. Some information cannot be replicated, such as the stack (stack pointer to a different memory area per thread), registers and thread-specific data. These allow threads to be scheduled independently of the program’s main thread and possibly one or more other threads within the program. Explicit operating system support is required to run multithreaded programs. Fortunately, most modern operating systems support threads such as Linux and Windows. Operating systems may use different mechanisms to implement multithreading support. One particular threading implementation is POSIX threads (Pthreads). Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library.

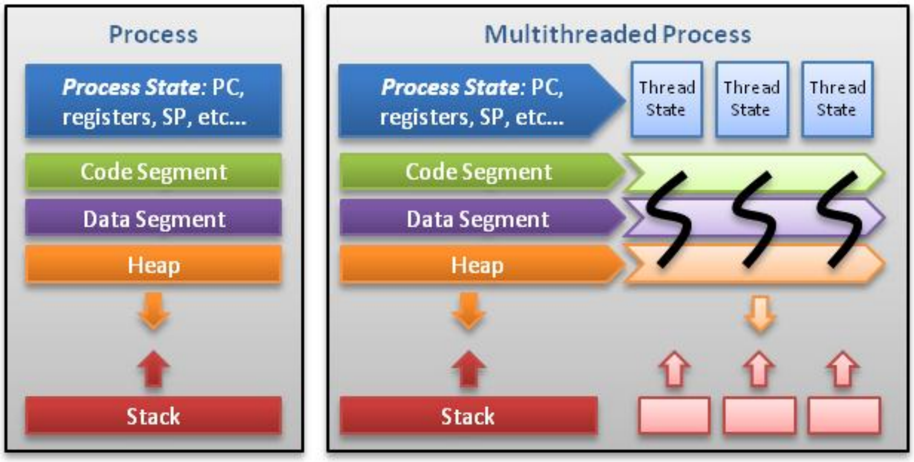

Figure 3.1. Information Shared within the Process among All Threads

3.1.1. Design Threaded Program. In general, in order for a program to take advantage of Pthreads, it must be organized into discrete, independent tasks which can execute concurrently.

For example, in Figure 3.2, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidate for threading. There are different ways to use threads in a program. Three common thread design patterns are presented:

Manager/worker: One thread dispatches other threads to do useful work which are usually part of a worker thread pool. This thread pool is usually pre-allocated before the manager begins dispatching threads to work. Although threads are lightweight, they still incur overhead when they are created.

Pipeline: Similar to how pipelining works in a processor, each thread is a part of a long chain in a processing factory. Each thread works on data processed by the previous thread and hands it off to the next thread.

Peer: The peer model is similar to the manager/worker model except once the worker pool has been created, the boss becomes another thread in the thread pool, and it thus, a peer to the other threads.

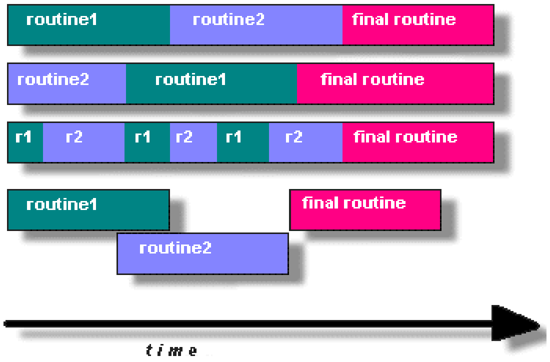

Figure 3.2. Threaded Program

All threads have access to the same global and shared memory. Besides, threads have their own private data. Figure 3.3 illustrate the shared memory model.

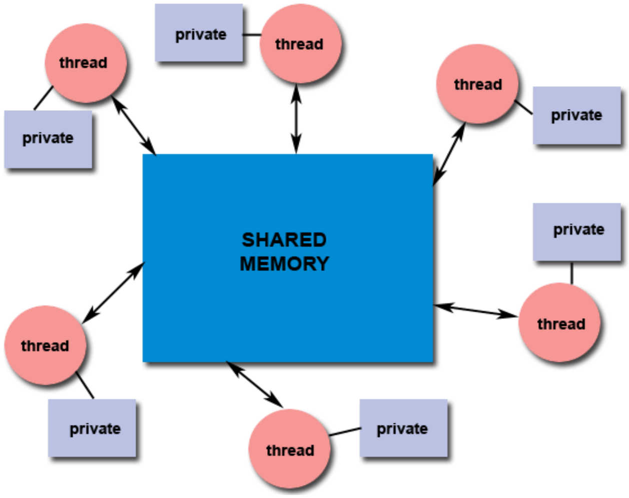

Figure 3.3. Shared Memory Model

Thread safeness refers to an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions. Threads may operate on disparate data, but often threads may have to touch the same data. It is unsafe to allow concurrent access to such data or resources without some mechanism that defines a protocol for safe access. Threads must be explicitly instructed to block when other threads may be potentially accessing the same resources. For example, suppose that an application creates several threads, each of which makes a call to the same library routine. This library routine accesses a global structure or location in memory. As each thread calls this routine, it is possible that they may try to modify this global structure/memory locations at the same time. If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe. See Figure 3.4.

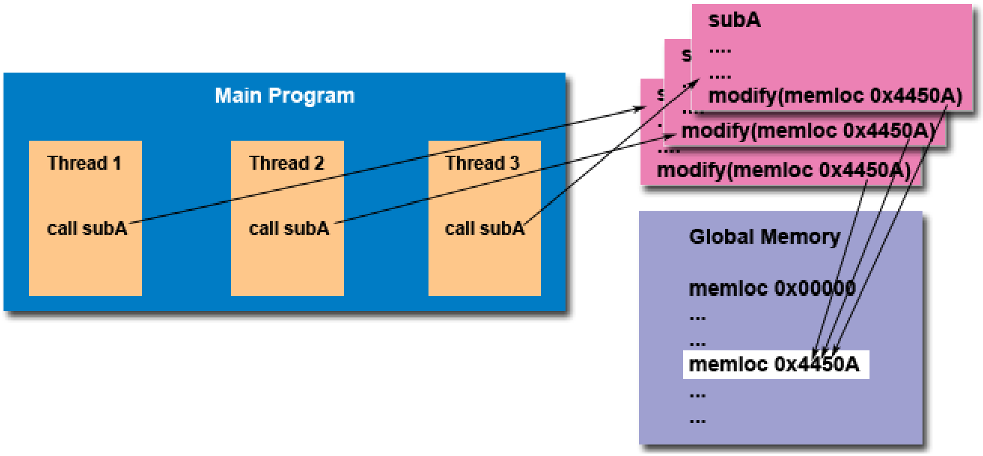

Figure 3.4. Thread Safeness Illustration

3.1.2. The Pthreads APIs. The subroutines which comprise the Pthreads API can be informally grouped into four major groups: Thread management, mutexes, condition variables and synchronization.

Figure 3.5 shows the Pthreads APIs:

Thread management: Routines that work directly on threads, for example, creating, detaching and joining. They also include functions to set/query thread attributes.

Mutexes: Routines that deal with synchronization, called a “mutex,” which is an abbreviation for “mutual exclusion.” Mutex functions provide for creating, destroying, locking and unlocking mutexes. One of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur. Mutex acts like a lock protecting access to a shared data resource. Only one thread can lock a mutex variable at any given time. A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable.
- Several threads attempt to lock the mutex.
- Only one succeeds and that thread owns the mutex.
- The owner thread performs some set of actions.
- The owner unlocks the mutex.

- Another thread acquires the mutex and repeats the process.
- Finally the mutex is destroyed.

Condition variables: Routines that address communications between threads that share a mutex. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data. A condition variable is always used in conjunction with a mutex lock. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included. Condition variables must be declared with type pthread_cond_t and must be initialized before they can be used. When it is no longer needed, pthread_cond_destroy() should be used to free a condition variable.

Synchronization: Routines that manage read/write locks and barriers. Synchronization is an enforcing mechanism used to impose constraints on the order of execution of threads, in order to coordinate thread execution and manage shared data. Synchronization mechanism has three types: mutexes, condition variable and joins.

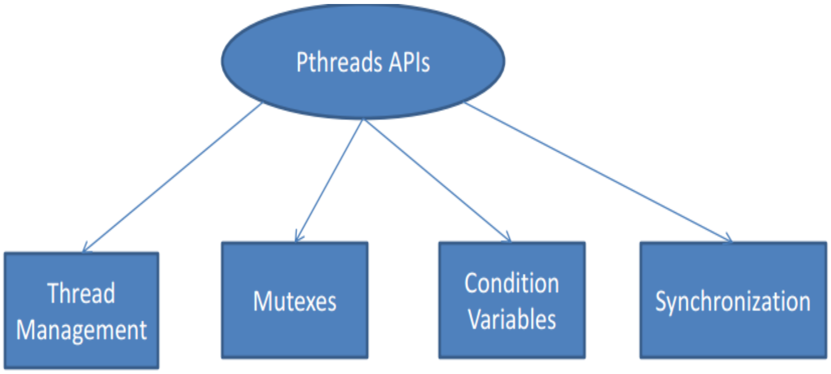

Figure 3.5. Pthreads APIs

3.2. DATA STRUCTURES AND ALGORITHM TO IMPLEMENT CPPR

3.2.1. Data Structures. In this thesis, the major three data structures created are Pin, Wire and Path.

Pin data structure will store timing information in the netlist such as the early mode arrival time, early mode required arrival time, early mode slack, late mode arrival time, late mode required arrival time, late mode slack. As the D pins and the CLK pins in a latch or in a flip-flop are especially important in the project, therefore the base Pin data structure will have two derived data structures: DPin, CKPin, which will store further timing information in the netlist such as the set up time and the hold time.

The Wire data structure functions as edge in a graph data structure, it defines the connectivity info in the netlist, and it also stores delay information for Pin to Pin such as each gate and each wire in the netlist, as in the STA process, not only the early mode delay but also the late mode delay is necessary.

The Path data structure will be used for the set up test and hold test in the CPPR process and it will store the early mode pre-CPPR slack as well as the late mode pre-CPPR slack.

3.2.2. Algorithm. As it mentioned in the previous section, this project mainly consist of three major steps. First of all, after parsing the inputs to a directed acyclic graph using the data structure mentioned above, it should perform Static Timing Analysis to get the pre-CPPR slack, and then it will use Depth First Search to enumerate all the data path and clock path between primary input (s) and a latch/FF or latch/FF and latch/FF or latch/FF and primary output (s). The final step will be perform set up test or hold test and generate corresponding test report according to user’s input according to the path information got from the second step. Furthermore, the project will use parallel computing technique in the input parsing, the path enumeration step and the report generation step.

To be more specific, the whole process is as followed:

- Input parsing: parse the netlist and corresponding timing information into a directed acyclic graph (DAG).

- Forward propagation: perform forward propagation in DAG from primary inputs to primary outputs to store the early mode and late mode arrival time.
- Backward propagation: perform backward propagation in the DAG from primary outputs to primary inputs to store the early mode and late mode required arrival time.
- Clock path enumeration: use depth first search (DFS) algorithm to find all the paths starting from the clock source to each latch/FF's CLK pins.
- Data path enumeration: use depth first search (DFS) algorithm to find all the paths starting from the primary inputs to each latch/FF's D pins.
- CPPR tests: find the late clock path and early clock path for a CLK pin of a latch/FF, then go through all paths to its corresponding data input pins. Calculate the setup test and hold test post-slack result and keep it.
- Test report generation: output the most critical paths of latch/FF from the number of tests according to user's input. The report will cover the post-CPPR slack after the setup test or the hold test and the specific critical path information.

This thesis uses parallel computing to fasten the exhaustive method, follows are the specific information on where to use the multi-threaded programming:

- After the DAG was built, during the forward propagation step, backward propagation, clock path enumeration and the D path enumeration, parallel computing could be used to run simultaneously because they are independent of each other.
- The CPPR tests for each latch/FF are independent, therefore, parallel computing will also be used in this step.
- After CPPR was performed, the path information of each latch/FF are fixed and because the path of different latch/FF are independent of each other, the test report generation could be run in parallel.

4. EXPERIMENTAL RESULTS

This section shows experimental results of presented algorithm which was implemented in the C++ programming language and complied with g++. The program was executed on a platform with 2.5GHz dual-core I5 CPU and with 8GB memory under MacOS 10.9.5.

This experiment are conducted on thirteen benchmarks circuits from 2014 Tau Contest, the statistics of each benchmark are shown in Table 4.1, ranging from small to large scale design. '#PIs' means the number of primary inputs, '#POs' means the number of primary outputs, '#FFs' means the number of flip-flops, '#Gates' means the number of combinational gates, and '#Paths' means the number of paths in a benchmark circuit. The number of paths for each benchmark varies from single digit to millions in ascending order. The experiment performs a comparison between the efficiency of the multi-threaded method and traditional single-threaded method.

The experiment takes in two files: a delay file and a timing input file. The delay file will describe the timing behavior of the circuit. The timing input file will describe the initial timing conditions which will be used during timing propagation.

The delay file contains the description of the circuit cells and the delays for direct connection. The circuit topology is implicitly derived from this file. Each line will represent a delay for each pin-pair in the design. Each line will contain two numbers: one for early and one for late mode. All numerical results should be given in seconds and printed in scientific notation. All keywords and variable fields should be separated by a white space.

Timing input file will contain the relevant timing information needed to propagate timing.

The output file should contain the paths for each type of test specified in accordance with the input commands.

As a comparison, this experiment has recorded runtime of the single-threaded solution and the multi-threaded solution on both set up test and hold test on benchmarks from Table 4.1. Note that the experiment sets the number of test to 10 and number of path to 10. The results from both the single-threaded solution and multi-threaded solution have been verified and turn out to be correct. For the runtime of this two solutions, from Table 4.2, it can be seen that for the same case and same solution, the runtime of hold test is usually a little less than the runtime of set up test, that is because the set up test has to go through every element in the path while the hold test only needs to find the merge point on the path. After comparison of the runtime of these two solutions, it can be seen that after applying multi-threaded computing technique, the runtimes of code are much faster than the single-threaded solutions, the saved time can be very large when the case is significantly large, especially when the path number of the case is very large.

To sum it up: the hold test is usually faster than the set up test.
The parallel computing technique can significantly fasten the runtime in this project and will save a lot of time on the significant large cases.

Table 4.1. Benchmarks Statistics

Benchmarks	**#PIs**	**#POs**	**#FFs**	**#Gates**	**#Paths**
s27v2	6	1	3	39	23
s386v2	9	7	6	192	129
s510v2	21	7	6	306	277
s344v2	11	11	15	191	315
s349v2	11	11	15	203	322
s526v2	5	6	21	324	409
s1494v2	10	19	6	819	508
s400v2	5	6	21	241	539
s1196v2	16	14	18	660	676
wb_dmav2	217	215	523	4433	19775
aes_corev2	260	129	530	23726	1039968
systemcdesv	132	65	190	3713	1108437

Table 4.2. Runtime Comparison

Benchmarks	Type	Single-threaded	Multi-threaded	Effectiveness in Reduction Time
s27v2	setup	0.0080721	0.005964	26.2%
	hold	0.009847	0.006199	37.1%
s386v2	setup	0.038079	0.025469	33.3%
	hold	0.047396	0.024136	49.1%
s510v2	setup	0.076158	0.035799	53%
	hold	0.074274	0.03348	54.9%
s344v2	setup	0.081743	0.039056	52.1%
	hold	0.086128	0.038732	55.1%
s349v2	setup	0.07668	0.042288	44.9%
	hold	0.075227	0.0368	50.9%
s526v2	setup	0.11473	0.055571	51.6%
	hold	0.117818	0.050463	57.2%
s1494v2	setup	0.160954	0.076508	52.5%
	hold	0.14659	0.066055	54.9%
s400v2	setup	0.133062	0.066127	50.4%
	hold	0.132101	0.055824	57.8%
s1196v2	setup	0.16938	0.096221	43.2%
	hold	0.169076	0.081487	51.8%
wb_dmav2	setup	6.147627	2.281624	62.9%
	hold	5.553806	2.146346	61.4%
aes_corev2	setup	258.963142	157.224541	39.9%
	hold	253.931346	148.43621	41.6%
systemcdesv	setup	243.6763	148.418269	40%
	hold	231.7322	133.178243	42.5%

5. CONCLUSIONS

This thesis has presented a static timing analysis timer that can deal with common path pessimism removal problem. To speed up the existing approached which are predominated by exhaustive path search, this thesis applies an efficient search routine using a quick and efficient multi-threaded depth first search (DFS) algorithm to obtain an exact solution. Comparatively, experimental results have demonstrated the superior performance of the timer in terms of accuracy and runtime over the traditional method. The proposed method is highly scalable, very promising for large-scale designs.

Future works shall focus on the development of even more efficient algorithm for path-based CPPR. Studies in fast CPPR algorithms are still eagerly in demand especially when moves to multi-core or many-core area.

BIBLIOGRAPHY

[1] C. Visweswariah, et al., “First-order increment block-based statistical timing analysis,” in *DAC*, July 2004, pp. 331-336.

[2] J. Bhasker and R. Chadha, “Static Timing Analysis for Nanometer Design: A Practical Approach,” Springer, 2009.

[3] S. Cristian, N. H. Rachid, and R.Khalid, “Efficient exhaustive path-based static timing analysis using a fast estimation technique,” US patent 8079004, 2009.

[4] J. Hu, et al., “TAU 2014 contest on removing Common path pessimism during timing analysis,” in Proc. TAU Workshop, Mar. 2014, pp. 56-63.

[5] J. Zejda and P. Frain, “General framework for removal of clock network pessimism,” in ICCAD, Nov. 2002, pp. 632-639.

[6] C. M. Darsow and T. D. Helvey, “Implementing forward tracing to reduce pessimism in static timing of logic blocks laid out in parallel structures on an integrated circuit chip,” US patent 0023466, 2012.

[7] S. Singh, et al., “Common path pessimism removal for hierarchical timing analysis,” US patent 8572532, 2013.

[8] D. J. Hathaway, et al., “Network timing analysis method which eliminates timing variations between signals traversing a common circuit path,” US patent 5636372, 1997.

[9] K. Kalafala, et al., “System and method for correlated process pessimism removal for static timing analysis,” US patent 7117466, 2006.

[10] S. Sirichotiakul, V. Zolotov, R. Levy and D. Blaauw. “Driver modeling and alignment for worst-case delay noise,” in *DAC*, pp. 720-725, 2001.

[11] C. Visweswariah, K. Ravindran, K. Kalafala, S. Narayan and S. G. Walker. “First-order incremental block-based statistical timing analysis,” in *DAC*, pp. 331-336, 2004.

VITA

Chunyu Wang was born in Tianjin, China. She received her Bachelor's Degree in Electrical Engineering at Beijing University of Chemical Technology in 2012. In August 2015, she received her Master's Degree in Computer Engineering from Missouri University of Science and Technology.