DISTRIBUTED TIMING ANALYSIS

BY

TSUNG-WEI HAUNG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Martin D. F. Wong, Chair
Professor Deming Chen
Professor Rob A. Rutenbar
Professor Wen-Mei Hwu

ABSTRACT

As design complexities continue to grow larger, the need to efficiently analyze circuit timing with billions of transistors across multiple modes and corners is quickly becoming the major bottleneck to the overall chip design closure process. To alleviate the long runtimes, recent trends are driving the need of distributed timing analysis (DTA) in electronic design automation (EDA) tools. However, DTA has received little research attention so far and remains a critical problem. In this thesis, we introduce several methods to approach DTA problems. We present a near-optimal algorithm to speed up the path-based timing analysis in Chapter 1. Path-based timing analysis is a key step in the overall timing flow to reduce unwanted pessimism, for example, common path pessimism removal (CPPR). In Chapter 2, we introduce a MapReduce-based distributed Path-based timing analysis framework that can scale up to hundreds of machines. In Chapter 3, we introduce our standalone timer, OpenTimer, an open-source high-performance timing analysis tool for very large scale integration (VLSI) systems. OpenTimer efficiently supports (1) both block-based and path-based timing propagations, (2) CPPR, and (3) incremental timing. OpenTimer works on industry formats (e.g., .v, .spef, .lib, .sdc) and is designed to be parallel and portable. To further facilitate integration between timing and timing-driven optimizations, OpenTimer provides user-friendly application programming interface (API) for inactive analysis. Experimental results on industry benchmarks released from TAU 2015 timing analysis contest have demonstrated remarkable results achieved by OpenTimer, especially in its order-of-magnitude speedup over existing timers.

In Chapter 4 we present a DTA framework built on top of our standalone timer OpenTimer. We investigated into existing cluster computing frameworks from big data community and demonstrated DTA is a difficult fit here in terms of computation patterns and performance concern. Our specialized

DTA framework supports (1) general design partitions (logical, physical, hierarchical, etc.) stored in a distributed file system, (2) non-blocking IO with event-driven programming for effective communication and computation overlap, and (3) an efficient messaging interface between application and network layers. The effectiveness and scalability of our framework has been evaluated on large hierarchical industry designs over a cluster with hundreds of machines.

In Chapter 5, we present our system DtCraft, a distributed execution engine for compute-intensive applications. Motivated by our DTA framework, DtCraft introduces a high-level programming model that lets users without detailed experience of distributed computing utilize the cluster resources. The major goal is to simplify the coding efforts on building distributed applications based on our system. In contrast to existing data-parallel cluster computing frameworks, DtCraft targets on high-performance or compute-intensive applications including simulations, modeling, and most EDA applications. Users describe a program in terms of a sequential *stream graph* associated with computation units and data streams. The DtCraft runtime transparently deals with the concurrency controls including work distribution, process communication, and fault tolerance. We have evaluated DtCraft on both micro-benchmarks and large-scale simulation and optimization problems, and showed the promising performance from single multi-core machines to clusters of computers.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to express my special thanks to my adviser Prof. Martin D. F. Wong, who has mentored me for years. I would like to thank you for giving me a lot of insightful advice on my research and helping me to grow as a research scientist. This dissertation would not be possible without your advice and wisdom. I am also very thankful to all my doctoral committee, Prof. Deming Chen, Prof. Wen-Mei Hwu, and Prof. Rob Rutenbar. Their constructive comments and suggestions have proven to be extremely useful for this thesis.

I am extremely grateful for all my colleague in UIUC, who have always been supportive in both my research and my life. I want to thank Chun-Xun Lin for excellent teamwork on our research project. I want to thank Dr. Haitong Tian for sharing his career experience with me and Dr. Zigang Xiao for instructing me on how to use commercial EDA tools. I want to thank Dr. Li-Da Huang for giving insightful comments for several of my research topics, and kindly accommodating me when I first arrived in the United States. I want to thank visiting scholars, Prof. Fan Zhang, Prof. Chun-Yao Wang, Prof. Hung-Ming Chen, Yen-Chen Lai, and Dr. Deojkin Joo for helpful discussions on my research topics. Also, I want to thank Dr. Pei-Ci Wu, Dr. Ting Yu, Leslie Hwang, Daifend Guo, Tin-Yin Lai, and Iou-Jen Liu for making my PhD life colorful and enjoyable. I appreciate my industry partners, Dr. P. V. Srinivas, Dr. Ismail Bustany, Dr. Shankar Krishnamoorthy, Dr. Igor Keller, Dr. Sharad Mehrotra, Dr. Qiuyang Wu, Dr. Natesan Venkateswaran, Dr. Kerim Kalafala, Dr. Debjit Sinha, Dr. Jin Hu, and Dr. Myung-Chun Kim, for very helpful technical suggestions on my research. Furthermore, I appreciate Marco and Dr. Daniele Paolo Scarpazza at Citadel for broadening my vision to the research in the financial world.

I am extremely lucky for meeting lots of friends in UIUC. My roommate Jhih-Chian Wu has been extremely helpful during my ups and downs

throughout my PhD life. I also want to thank my friends, Yingyan Lin, Billy Lee, Chen-Hsuan Lin, Chun-Yu Shao, Chichi Cheng, Hsien-Chih Chang, Yuguang Chen, Bei Yu, Xiaoqing Xu, Jong Bin Lim, Wei Zuo, Sitao Huang, Yi Liang, Ashutosh Dhar, Yihao Zhang, Yuting Chen, Pratik Lahiri, Jiangxiong Gao, Danny Kim, Xingkai Zhou, and many others.

Finally, I give my deepest gratitude to my parents and my little brother who have been always supportive throughout my whole life. I also want to thank my girlfriend for the great care in my life. I cannot express my love and gratitude for them in words.

TABLE OF CONTENTS

CHAPTER 1

COMMON PATH PESSIMISM REMOVAL

1.1 Introduction

The lack of accurate and fast algorithms for common path pessimism removal (CPPR) has been recently pointed out as a major weakness of existing static-timing analysis (STA) tools [1]. Conventional STA tools rely on conservative dual-mode operations to estimate early-late and late-early path slacks [2]. This mechanism, however, imposes unnecessary pessimism due to the consideration of delay variation along common segments of clock paths, as illustrated in Figure 1.1. This is because signal cannot simultaneously experience early-mode and late-mode operations along the physically common segment of the data path and clock path in the clock network. Unnecessary pessimism may lead to timing tests (e.g., setup check, hold check, etc.) being marked as failing whereas in reality they should be passing. Thus designers and optimization tools might be misled into an over-pessimistic timing report. Therefore, the goal of this chapter is to identify and eliminate unwanted pessimism during STA so as to prevent true timing properties of circuits from being skewed.

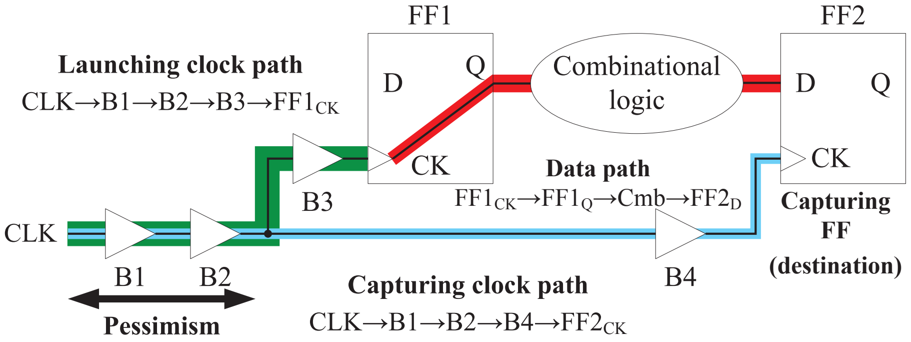


Figure 1.1: Common path pessimism incurs in the common path between the launching clock path and the capturing clock path.

The importance and impact of CPPR are demonstrated in Figure 1.2. It is observed that the number of failing tests was reduced from 642 to less than half after the pessimism was removed. Unwanted pessimism might force designers and optimization tools to waste a significant yet unnecessary amount of efforts on fixing paths that meet the intended clock frequency. Such a problem becomes even critical when design comes to deep submicron era where data paths are shorter, clocks are faster, and clock networks are longer to accommodate larger and complex chips. Moreover, without pessimism removal designers and CAD tools are no longer guaranteed to support legal turnaround for timing-specific improvements, which dramatically degrades the productivity. At worst, signoff timing analyzer gives rise to the issue of "leaving performance on the table" and concludes a lower frequency at which the circuits can operate than their actual silicon implementations [3].

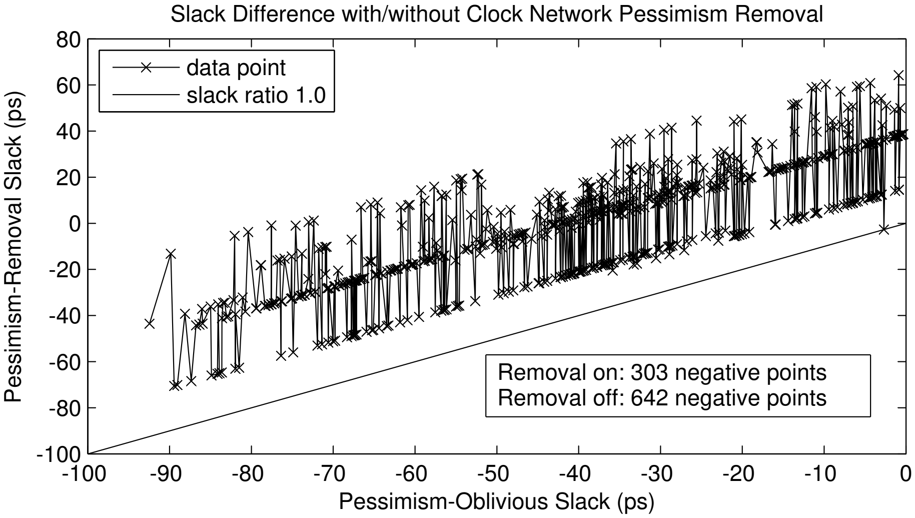


Figure 1.2: Impact on common path pessimism from a circuit in [4].

State-of-the-art CPPR algorithms are dominated by straightforward path-based methodology [5, 6, 7]. Critical paths are identified without considering the pessimism first. Then for each path the common segment is found by a simple walk through the corresponding launching clock path and capturing clock path. Finally, slack of each path is adjusted by the amount of pessimism on the common segment. The real challenge is the amount of pessimism that needs to be removed is path-specific. The most critical path prior to pessimism removal is not necessarily reflective of the true counterpart (see the line plot in Figure 1.2), revealing a potential drawback that path-based methodology has the worst-case performance of exhaustive search space in peeling out the true critical paths. Accordingly, prior works are usually too slow to handle complex designs and unable to always identify the true critical path accurately [4].

In this chapter we introduce UI-Timer 1.0, a powerful CPPR algorithm which achieves high accuracy, ultra-fast runtime, and low memory requirement. UI-Timer 1.0 is the preliminary version of OpenTimer and its details can be referred to [8]. Our contributions are summarized as follows: (1) We introduce a theoretical framework that maps the CPPR problem to a graph search formulation. The mapping allows the true critical path to be directly identified through our search space, rather than the time-consuming yet commonly applied strategy which interleaves the search between slack computation and pessimism retrieval. (2) Unlike predominant explicit path search, we represent the path implicitly using two efficient and compact data structures, namely suffix tree and prefix tree, and yield a significant saving in both search space and search time. (3) The effectiveness and efficiency of our timer have been verified by the TAU 2014 CAD contest [4]. Comparatively, UI-Timer 1.0 confers promissing results over existing timers in terms of accuracy and runtime. The source code of our timer has been released to the public domain [9], which can be an indicator assisting researchers in discovering and optimizing the performance bottleneck of their tools.

1.2 Static Timing Analysis

STA is a method of verifying expected timing characteristics of a circuit. The dual-mode or early-late timing model is the most popular convention because it provides both lowerbound and upperbound quantities to accounts for various on-chip variations (OVC) such as process parameter, e.g., transistor width, voltage drops, and temperature fluctuations [2]. In contrast to statistical STA (SSTA) where process variations are modeled as random variables, the early-late timing model has deterministic behaviors and thus enables lower computational complexity for timing propagation. The earliest and latest timing instants that a signal reaches are quantified as earliest and latest *arrival time* (at), while the limits imposed on a circuit node for proper logic operations are quantified as earliest and latest *required arrival* *time* (rat). The verification of timing at a circuit node is determined by the largest difference or *worst slack* between the required arrival time and signal arrival time. We focus on two primary types of timing verification – *hold test* and *setup test* for a specified data point at a flip-flop (FF). The hold test and setup test are two safe timing guards that constrain the earliest required arrival time and the latest required arrival time for a data point, respectively. Considering a timing test t, the following equations are applied for STA [4].

$$rat_t^{early} = at_o^{late} + T_{hold},\ rat_t^{late} = at_o^{early} + T_{clk} - T_{setup} \tag{1.1}$$

$$slack_{worst}^{hold} = at_d^{early} - rat_t^{early},\ slack_{worst}^{setup} = rat_t^{late} - at_d^{late} \tag{1.2}$$

Notice that T_{clk} is the clock period, T_{hold} and T_{setup} are values of hold and setup constraints, and o and d are respectively the clock pin and the data pin of the testing FF. In general, the best-case fast condition is critical for hold test and the worst-case slow condition is critical for setup test. For a data path feeding the testing FF, a positive slack means the required arrival time is satisfied and a negative slack means the required arrival time is in a violation.

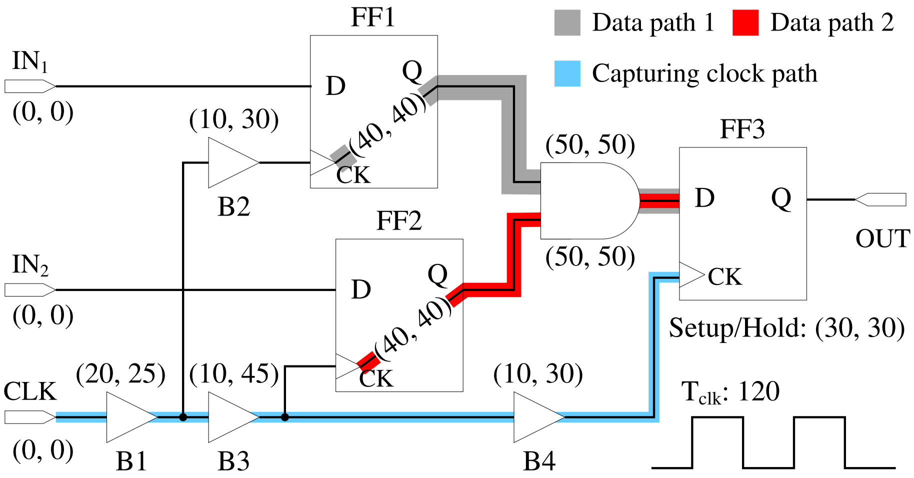


Figure 1.3: An example of sequential circuit network.

Consider a sample circuit in Figure 1.3, where two data paths feed a common FF. Numbers enclosed within parentheses denote the earliest and latest delay of a circuit node. Assuming all wire delays and arrival times of primary inputs are zero, we perform the setup test on FF3. The latest required arrival time of FF3 is obtained by subtracting the values of clock period plus the earliest arrival time at the clock pin of FF3 from the value of setup constraint, which is equal to $(120 + (20 + 10 + 10)) - 30 = 130$. The respective latest arrival times of data path 1 and data path 2 at the data pin of FF3 are $25 + 30 + 40 + 50 = 145$ and $25 + 45 + 40 + 50 = 160$. Using equation (1.2), the setup slacks of data path 1 and data path 2 are $130 - 145 = -15$ (failing) and $130 - 160 = -30$ (failing), respectively.

1.3 Common Path Pessimism Removal

The dual-mode split-timing analysis has greatly enabled timers to effectively account for any within-chip variation effects. However, the dual-mode analysis inherently embeds unnecessary pessimism, which results in an over-conservative design. Take the slack of data path 1 in Figure 1.3 for example. The pessimism arises with buffer B1 since it was accounted for both earliest and latest delays at the same time which is physically impossible. In general, the pessimism of two circuit nodes appears in the common path from the clock source to the closest point to which the two nodes converge through upstream traversal. Such a point is also referred to as the *clock reconverging node*. The amount of pessimism is equal to the cumulative differences between late and early delays along the common path. The true timing without pessimism can be obtained by adding the final slack to a credit which is defined as follows [4]:

$$credit_{u,v}^{hold} = at_{cp}^{late} - at_{cp}^{early} \tag{1.3}$$

$$credit_{u,v}^{setup} = at_{cp}^{late} - at_{cp}^{early} - (at_{r}^{late} - at_{r}^{early}) \tag{1.4}$$

$$slack_{post-CPPR}^{setup} = slack_{pre-CPPR}^{setup} + credit_{u,v}^{setup} \tag{1.5}$$

$$slack_{post-CPPR}^{hold} = slack_{pre-CPPR}^{hold} + credit_{u,v}^{hold} \tag{1.6}$$

Notice that r is the clock source and cp is the clock reconverging node of nodes u and v. Since the setup test compares the data point against the clock point in the subsequent clock cycle, the credit rules out the arrival time at the clock source [4]. The slack prior to common path pessimism removal (CPPR) is referred to as *pre-CPPR slack* and *post-CPPR slack* otherwise. For the same instance in Figure 1.3, the credits of data path 1 and data path 2 for the setup tests are respectively 5 and 40, which in turn tell their true slacks being $-15 + 5 = -10$ (failing) and $-30 + 40 = 10$ (passing). A key observation here is that the most critical pre-CPPR slack (data path 2) is not necessarily reflective of the true critical path (data path 1). Analyzing the single-most critical path during CPPR is obviously insufficient. In practice, reporting a number of ordered critical paths for a given test rather than merely the single-most critical one is relatively necessary and important.

1.4 Prior Works

Removing pessimism from the design during timing analysis is integral to meeting chip timing, area, and power targets. To this end, existing STA tools continue to invest heavily in research and development on this topic and explore new ideas and concepts to improve CPPR runtime and memory usage [10]. The predominant approach relies on identifying a set of critical paths without CPPR first. Then the CPPR credit of each of these paths are discovered through the traversal on the clock network, after which the true slack can be retrieved [6, 7]. Based on this framework, straightforward heuristics such as dominator grouping for clock reconverging nodes [3], hierarchical timing analysis [5], branch-and-bound pruning [11, 12], and CPPR credit caching [13] are proposed to either shrink the solution space or reduce the computational complexity. However, these works suffer from a common drawback of exhaustive search space. In spite of fine-tuned heuristics, the resulting performance is always case-by-case and has no guaranteed characteristics of polynomial space and time complexity.

1.5 Problem Formulation

The circuit network is input as a directed-acyclic graph (DAG) $G = \{V, E\}$. V is the node set with n nodes which specify pins of circuit elements (e.g., primary IO, logic gates, FFs, etc.). E is the edge set with m edges which specify pin-to-pin connections. Each primary input, i.e., the node with zero indegree, is assigned by an earliest arrival time and a latest arrival time. Each edge e or $e_{u \to v}$ is directed from its tail node u to head node v and is associated with a dual tuple of earliest delay $delay_e^{early}$ and latest delay $delay_e^{late}$. A path is an ordered sequence of nodes $\langle v_1, v_2, \cdots, v_n \rangle$ or edges $\langle e_1, e_2, \cdots, e_n \rangle$ and the path delay is the sum of delays through all edges. We are in particular emphasizing on the data path, which is defined as a path from the clock source pin of an FF to the data pin of another FF. The arrival time of a data path is the sum of its path delay and arrival time from where this data path originates. The clock tree is a subgraph of G which distributes the clock signal with clock period T_{clk} from the tree root r to all the sequential elements that need it. A test is defined with respect to an FF as either a hold check or setup check to verify the timing relationship between the clock pin and the data pin of the FF, so that the hold requirement T_{hold} or setup requirement T_{setup} is met. We refer to the testing FF as *destination FF* and those FFs having data paths feeding the destination FF as *source FFs.* Using the above knowledge, the CPPR problem is formulated as follows:

Objective: *Given a circuit network G and a hold or setup test t as well as a positive integer k, the goal is to identify the top k critical paths (i.e., data paths that are failing for the test) from source FFs to the destination FF in ascending order of post-CPPR slack.*

1.6 Algorithm

The overall algorithm of UI-Timer 1.0 is presented in Algorithm 1. It consists of of two stages: *lookup table preprocessing* and *pessimism-free path search.* The goal of the first stage is to tabulate the common path information for quick lookup of credit, while the goal in the second stage is to identify the top-k critical paths in a pessimism-free graph derived from a given test. We shall detail in this section each stage in bottom-up fashion.

```
Algorithm 1: UI-Timer_1.0($t$, $k$)
    Input: test $t$, path count $k$
    Output: solution set $\Psi$ of the top-$k$ critical paths
BuildCreditLookupTable();
$G_p \leftarrow$ pessimism-free graph for the test $t$;
$\Psi \leftarrow$ GetCriticalPath($G_p.source$, $G_p.destination$, $k$);
return $\Psi$;
```

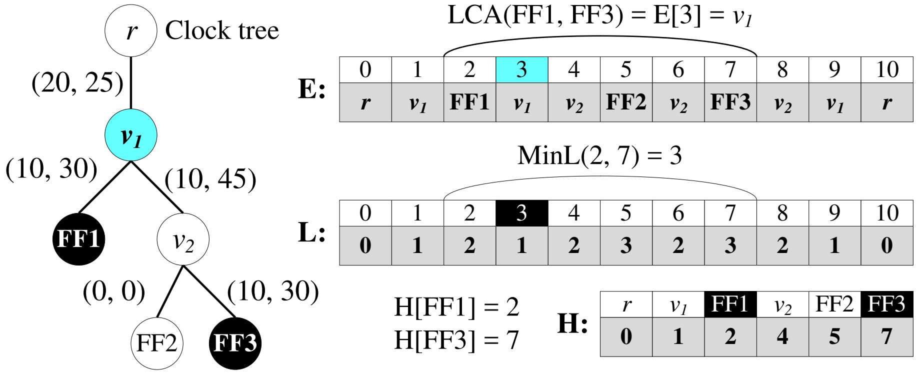
1.6.1 Lookup Table Preprocessing

In graph theory, the clock reconverging node of two nodes in the clock tree is equivalent to the lowest common ancestor (LCA) of the two nodes. The arrival time information of each node in the clock tree can be precomputed and therefore the credit of two nodes can be obtained immediately once their LCA is known. Many state-of-the-art LCA algorithms have been invented over the last few decades. The table-lookup algorithm by [14] is employed as our LCA engine due to its simplicity and efficiency. For a given clock tree, we build three tables as follows:

- The Euler table E records the identifiers of nodes in the Euler tour of the clock tree; $E[i]$ is the identifier of i^{th} visited node.
- The level table L records the levels of nodes visited in the Euler tour; $L[i]$ is the level of node $E[i]$.
- The occurrence table $H[v]$ records the index of the first occurrence of node v in array E.

As a result, the LCA of a node pair (u, v) is the node situated on the smallest level between the first occurrence of u the and first occurrence of v. We have the following: *Denoting the index of the node with the smallest level between the index a and b in the level table L as MinL(a, b), the LCA of a given node pair (u, v) is E[MinL($H[u]$, $H[v]$)].*

Take the LCA of FF1 and FF3 in Figure 1.4, for example. The occurrence indices of FF1 and FF3 in Euler tour are 2 and 7, respectively. Referring to the indices between 2 and 7 in the level table, the node with the lowest level is situated in the third position of the Euler table. Hence, the LCA of FF1 and FF3 is v_1. It is obvious the operations taken on the occurrence table and Euler table can be done in constant time. Finding the position of an



E:

0	1	2	3	4	5	6	7	8	9	10
r	v_1	FF1	v_1	v_2	FF2	v_2	FF3	v_2	v_1	r

L:

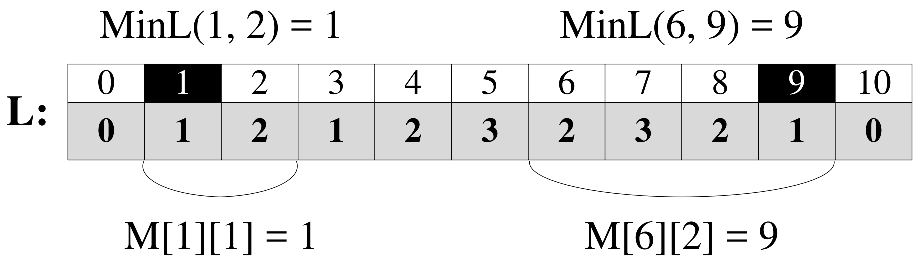
0	1	2	3	4	5	6	7	8	9	10
0	1	2	1	2	3	2	3	2	1	0

H:

r	v_1	FF1	v_2	FF2	FF3
0	1	2	4	5	7

Figure 1.4: Derived tabular fields from the clock tree in Figure 1.3.

element with the minimum value between two specified indices in the level table (i.e., the value returned by function $MinL(a,\, b)$ for a given index pair a and b) is the major task. We adopt the sparse-table solution whereby a two-dimensional (2D) table $M[i][j]$ is used to store the index of the minimum value in the level table starting at i having length 2^j [14]. This concept is visualized in Figure 1.5.



L:

0	1	2	3	4	5	6	7	8	9	10
0	1	2	1	2	3	2	3	2	1	0

Figure 1.5: Range minimum query to the level table from Figure 1.4.

Figure 1.5 indicates that the optimal substructure of $M[i][j]$ is the minimum value between the first and second halves of the interval with 2^{j-1} length each. Hence, the table M can be fulfilled using dynamic programming with the following recurrence:

$$M[i][j] = \begin{cases} i, \text{ base case } j = 0 \\ M[i][j-1], \text{ if } L[M[i][j-1]] \leq L[M[i+2^{j-1}][j-1]] \\ M[i+2^{j-1}][j-1], \text{ otherwise} \end{cases}$$

Provided the table M has been processed, the value of $MinL(a,\, b)$ can be computed by selecting two blocks that entirely cover the interval between a

and b and returning the minimum between them. Let c be $\lfloor log(b - a + 1) \rfloor$ and assume $b > a$, the following formula is used for computing the value of $MinL(a, b)$:

$$MinL(a,b) = \begin{cases} M[a][c], \text{ if } L[M[a][c]] \leq L[M[b-2^c+1][c]] \\ M[b-2^c+1][c], \text{ otherwise} \end{cases}$$

The procedure of building tables E, L, H, and M is presented in Algorithm 2. Tables E, L, and H can be built using a depth-first search starting at the root of the clock tree (line 1), while table M is fulfilled via bottom-up dynamic programming (line 2:16). Using these tables as infrastructure, the credit of two given nodes in the clock tree can be retrieved in constant time by Algorithm 3. The LCA of the two given nodes is found first (line 1:12). Then for the hold test, the credit is returned as the difference between the latest arrival time and the earliest arrival time at the LCA (line 14:15). For the setup test which performs the timing check in the subsequent clock cycle, the credit excludes the arrival time at the clock source (line 16:18).

Algorithm 2: BuildCreditLookupTable(G)

```
   Input: circuit network $G$
1  Build tables $E, L, H$ via Euler tour starting at the root $r$ of clock
    tree;
2  $size_1 \leftarrow L.size$;
3  $size_2 \leftarrow \lfloor log(L.size) \rfloor$;
4  Create a 2D table $M$ with size $size_1 \times$ $(size_2 + 1)$;
5  for $i \leftarrow 0$ to $size_1 - 1$ do
6      $M[i][0] \leftarrow i$;
7  end
8  for $j \leftarrow 1$ to $size_2 - 1$ do
9      for $i \leftarrow 0$ to $size_1 - 2^j$ do
10         if $L[M[i][j-1]] < L[M[i+2^{j-1}][j-1]]$ then
11             $M[i][j] \leftarrow M[i][j-1]$;
12         else
13             $M[i][j] \leftarrow M[i+2^{j-1}][j-1]$;
14         end
15     end
16 end
```

Algorithm 3: GetCredit(u, v)

```
Input: nodes $u$ and $v$
if $u$ or $v$ is not a node of the clock tree then
    return 0;
end
if $H[u] > H[v]$ then
    swap(u, v)
end
$c \leftarrow \lfloor log(H[u] - H[v] + 1) \rfloor$ ;
if $L[M[H[u]][c]] < L[M[H[v] - 2^c + 1][c]]$ then
    $lca \leftarrow E[M[H[u]][c]]$;
else
    $lca \leftarrow E[M[H[v] - 2^c + 1][c]]$;
end
if hold test then
    return $at_{lca}^{late} - at_{lca}^{early}$;
else
    $r \leftarrow$ root of the clock tree;
    return $at_{lca}^{late} - at_{lca}^{early} - (at_{r}^{late} - at_{r}^{early})$;
end
```

Theorem 1: *UI-Timer 1.0 builds lookup tables E, L, H, and M in O(nlogn) space and O(nlogn + m) time. Using these lookup tables, the credit of two given nodes in the clock tree can be retrieved in O(1) time.*

1.6.2 Formulation of Pessimism-Free Graph

In the course of a hold or a setup check, the required arrival time of the destination FF and the amount of pessimism between each source FF and the destination FF remain fixed regardless of which data path is being considered. Precisely speaking, the way data paths passing through plays the most vital role in determining the final slack values. In order to facilitate the path search without interleaving between slack computation and pessimism retrieval, we construct a pessimism-free graph $G_p = \{V_p, E_p\}$ for a given test t as follows:

Rule #1: We designate the data pin d of the destination FF the destination node and artificially create a source node s and connect it to the clock pin i of each source FF. Denoting the set of artificial edges as E_s, we have

$V_p = V \bigcup \{s\}$ and $E_p = E \bigcup E_s$.

Rule #2: We associate (1) *offset weight* with each artificial edge and (2) *delay weight* with each ordinary circuit connection as follows:

- $\forall e_{s \to i} \in E_s$, $w^{hold}_{e_{s \to i}} = credit^{hold}_{i,d} - rat^{early}_t + at^{early}_i$.
- $\forall e_{s \to i} \in E_s$, $w^{setup}_{e_{s \to i}} = credit^{setup}_{i,d} + rat^{late}_t - at^{late}_i$.
- $\forall e \in E$, $w^{hold}_e = delay^{early}_e$.
- $\forall e \in E$, $w^{setup}_e = -delay^{late}_e$.

An example of pessimism-free graph is shown in Figure 1.6. The intuition is to separate out the constant portion of the post-CPPR slack by an artificial edge such that the search procedure can focus on the rest portion which is totally depending on the way data paths passing through. It is clear that the cost of any source-destination path (i.e., sum of all edge weights) in the pessimism-free graph is equivalent to post-CPPR slack of the corresponding data path which is obtained by removing the artificial edge. This crucial fact is highlighted in the following theorem:

Theorem 2: *The cost of each source-destination path in the pessimism-free graph G_p is equal to the post-CPPR slack of the corresponding data path.*

Proof The cost of a source-destination path can be written as the delay of the corresponding data path p from the source FF i to the destination FF d plus the offset weight associated with the edge $e_{s \to i}$. The path cost for the hold test is $credit^{hold}_{i,d} - rat^{early}_t + at^{early}_i + \sum_{e \in p} delay^{early}_e$ and $credit^{setup}_{i,d} + rat^{late}_t - at^{late}_i - \sum_{e \in p} delay^{late}_p$ for the setup test. It is clear that by definition the cost is just the post-CPPR slack of a given path in either the hold test or the setup test.

The problem of identifying the top-k critical paths for a given test is similar to the path ranking problem applied to the pessimism-free graph. A number of state-of-the-art algorithms for path ranking have been proposed over the past few years [15, 16, 17, 18, 19]. The best time complexity acquired to date is $O(m + nlogn + k)$ from the well-know Eppstein's algorithm [16]. However, it relies on sophisticated implementation of a heap tree which results in little practical interests. Moreover, most existing approaches are developed for

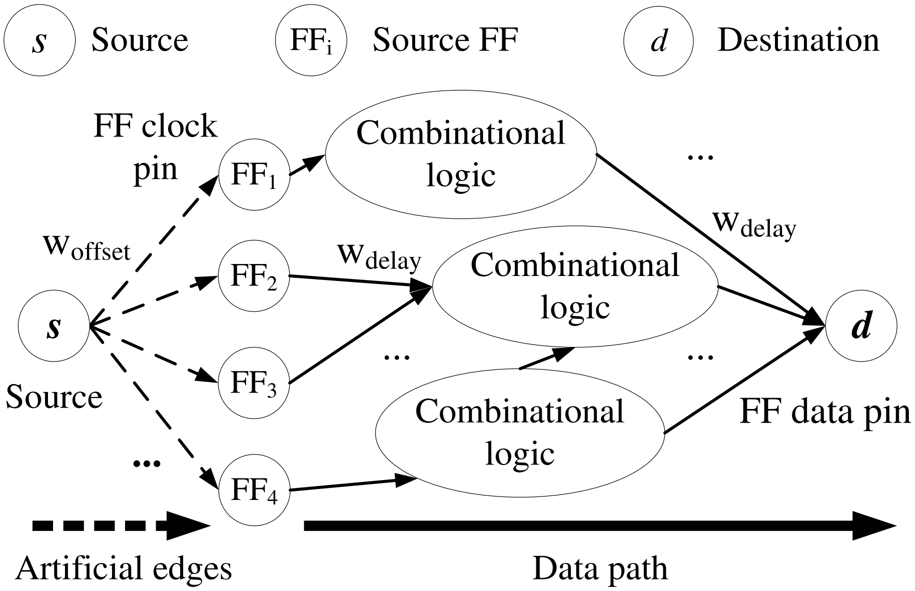


Figure 1.6: Derivation of pessimism-free graph from a given test.

general graphs and lack a compact and efficient specialization to certain graphs such as the directed-acyclic circuit network. We shall discuss in the following sections the key contribution of UI-Timer 1.0 in resolving these deficiencies.

1.6.3 Implicit Representation of Data Path

Although explicit path representation is the major pursuit of existing approaches, the inherent restriction makes it difficult to devise efficient algorithms with satisfactory space and time complexities [6, 7]. UI-Timer 1.0 performs implicit path representation instead, yielding significant improvements on memory usage and runtime performance. While the spirit is similar to [16], our algorithm differs in exploring a more compact and efficient way to the implicit path search and explicit path recovery. We introduce the following definitions:

Definition 1 – Suffix Tree: Given a pessimism-free graph, the suffix tree refers to the successor order obtained from the shortest path tree T_d rooted at the destination node.

Definition 2 – Prefix Tree: The prefix tree is a tree order of non-suffix-tree edges such that each node *implicitly* represents a path with prefix from its parent path deviated on the corresponding edge and suffix followed from the suffix tree. The root which is artificially associated with a null edge refers to the shortest path in T_d. Table 1.1 lists the data field to which we apply for each node.

Table 1.1: Data Field of a Prefix Tree Node

Member	Definition
p	pointer to the parent node
e	deviation edge
w	cumulative deviation cost
c	credit for pessimism removal
Constructor	PrefixNode(p, e, w, c)

An example is illustrated in Figure 1.7. The suffix tree is depicted with bold edges and numbers on nodes denote the shortest distance to the destination node. Dashed edges denote artificial connections from the source node. The shortest path is $\langle e_3, e_8, e_{12}, e_{15}\rangle$ which is implicitly represented by the root of prefix tree. The prefix tree node marked by "e_{11}" implicitly represents the path with prefix $\langle e_3, e_8\rangle$ from its parent path deviated on "e_{11}" and suffix $\langle e_{14}\rangle$ following from the suffix tree. As a result, explicit path recovery can be realized in a recursive manner as presented in Algorithm 4.

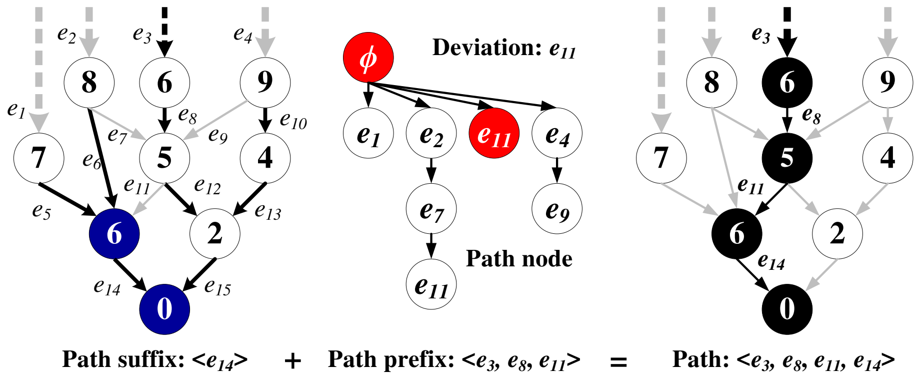


Figure 1.7: Implicit path representation using the suffix tree and the prefix tree.

In order to retrieve the path cost, we keep track of the deviation cost of each edge e, which is defined as follows [16]:

$$dvi[e] = dis[head[e]] - dis[tail[e]] + weight[e] \tag{1.7}$$

Algorithm 4: RecoverDataPath(*pfx*, *end*)

```
Input: prefix-tree node pointer pfx, node end
beg ← head[pfx.e];
if pfx.p ≠ NIL then
    RecoverDataPath(pfx.p, tail[pfx.e]);
end
while beg ≠ end do
    Record the path trace through pin "beg";
    beg ← successor[beg]
end
Record the path trace through pin "end";
```

Algorithm 5: Slack(*pfx*, *s*, *r*)

```
Input: prefix-tree node pointer pfx, source node s, CPPR flag r
Output: post-CPPR slack for true flag r or pre-CPPR slack
        otherwise
if $r$ = true then
    return $pfx.w + dis[s]$;
end
return $pfx.w + dis[s]$ - $pfx.c$;
```

Notice that $dis[v]$ denotes the shortest distance from node v to the destination node. Intuitively, deviation cost is a non-negative quantity that measures the distance loss by being deviated from e instead of taking the ordinary shortest path to destination. Therefore for each node in the prefix tree, the corresponding path cost (i.e., post-CPPR slack) is equal to the summation of its cumulative deviation cost and the cost of shortest path in T_d. Algorithm 5 realizes this process. We conclude the conceptual construction so far by the following two important lemmas.

Lemma 1: *UI-Timer 1.0 deals with the implicit representation of each data path in O(1) space and time complexities.*

Lemma 2: *The cumulative deviation cost of each node in the prefix tree is greater than or equal to that of its parent node.*

Lemmas 1 and 2 are two obvious byproducts of our prefix tree definition. UI-Timer 1.0 stores each data path in constant space and records or queries important information such as credit and slack in constant time. We shall demonstrate in the next section their strengths to help prune the search space.

1.6.4 Generation of Top-k Critical Paths

We begin by presenting a key subroutine of our path generating procedure – *Spur*, which is described in Algorithm 6. In a rough view, *Spur* describes the way UI-Timer 1.0 expands its search space for discovering critical paths. After a path p_i is selected as the i-th critical path, each node along the path p_i is viewed as a deviation node to spur a new set of path candidates (line 2:14). Any duplicate path should be ruled out from the candidate set (line 1 and line 5:7) and each newly spurred path is parented to the path p_i in the prefix tree (line 8). Having a path candidate with non-negative post-CPPR slack, the following search space can be pruned and is exempted from the queuing operation (line 9:11). This simple yet effective prune strategy is a natural result of lemma 3 due to the monotonic growth of path cost along with our search expansion.

Algorithm 6: Spur(*pfx*, *s*, *d*, *Q*)

```
   Input: prefix-tree node pointer pfx, source node s, destination node
          d, priority queue Q
1  $u \leftarrow head[pfx.e]$;
2  while $u \neq d$ do
3      for $e \in fanout(u)$ do
4          $v \leftarrow head[e]$;
5          if $v = successor[u]$ or $v$ is unreachable then
6              continue;
7          end
8          $pfx\_new \leftarrow$ new PrefixNode($pfx$, $e$, $pfx.w + dvi[e]$, $pfx.c$);
9          if Slack($pfx\_new$, $s$, true) $< 0$ then
10             $Q$.enque($pfx\_new$);
11         end
12     end
13     u $\leftarrow successor$[u];
14 end
```

Lemma 3: *The procedure Spur is compact, meaning every path candidate is generated uniquely.*

Proof Suppose there is at least a pair of duplicate path candidates p_1 and p_2, which are implicitly represented by ξ_1 and ξ_2 the sets of deviation edges. Since p_1 and p_2 are identical, ξ_1 and ξ_2 must be identical as well. If both ξ_1 and ξ_2 contain only one edge, the respective prefix tree nodes must be parented to the same node, which is invalid due to the filtering statement in line 5:7. If both ξ_1 and ξ_2 contain multiple edges, there exists at least two distinct permutations in the prefix tree that represent the same path. However, this will results in a cyclic connection of edges which violates the graph property of the circuit network. Therefore by contradiction the procedure *Spur* is compact.

Lemma 4: *The procedure Spur takes O(n + mlogk) time complexity.*

Proof The entire procedure takes up to n phases on scanning a given path and spurs at most m new path candidates. We maintain only the top-k critical candidates ever seen such that the maximum number of items in the priority queue at any time will not exceed k. This can be achieved in *O(mlogk)* time using a min-max priority queue [20]. Therefore the total complexity is *O(n + mlogk)*.

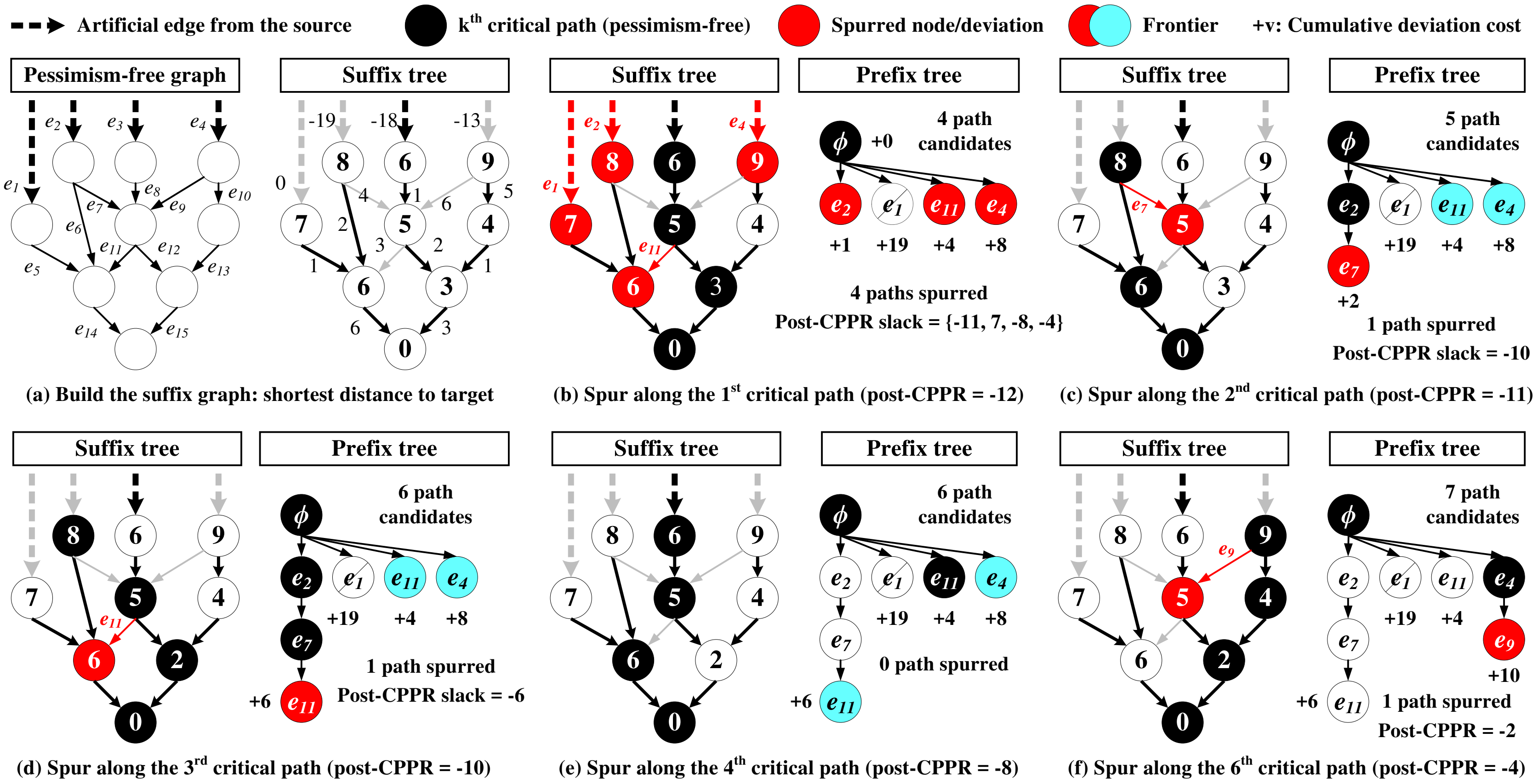
Using Algorithms 4–6 as primitive, the top-k critical paths can be identified using Algorithm 7. Prior to the search, we construct the suffix tree by finding the shortest path tree rooted at the destination node d in the pessimism-free graph (line 1). Then each of the most critical paths from source FFs to the destination FF is viewed as an initial path candidate (line 5:11). The major search loop (line 12:20) iteratively looks for a path with lowest cumulative deviation cost from the path candidate set and performs spurring operation on it. Iteration ends when we have extracted k paths (line 16:18) or no more steps can be proceeded. Finally, we draw the following two theorems.

Theorem 3: *UI-Timer 1.0 is complete, meaning that it can exactly identify the top-k critical paths for each hold test or setup test without common path pessimism.*

Proof Proving the completeness of UI-Timer 1.0 is equivalent to showing that the major search framework of UI-Timer 1.0 is exactly identical to a typical graph search problem [19]. The search space or search tree of UI-Timer

Algorithm 7: GetCriticalPath(s, d, k)

```
Input: source node $s$, destination node $d$, path count $k$
Output: solution set $\Psi$ of the top-$k$ critical paths
Build the suffix tree by finding the shortest path tree rooted at $d$;
Initialize a priority queue $Q$ keyed on cumulative deviation cost;
$\Psi \leftarrow \phi$ ;
$num\_path \leftarrow 0$;
for $e \in fanout(s)$ do
    $credit \leftarrow$ GetCredit($head[e]$, $d$);
    $pfx \leftarrow$ new PrefixNode(NIL, $e$, $dvi[e]$, $credit$);
    if $Slack(pfx, s,$ true$) < 0$ then
        $Q$.enque($pfx$);
    end
end
while $Q$ is not empty do
    $pfx\_new \leftarrow Q$.deque();
    $num\_path \leftarrow num\_path + 1$;
    $\Psi \leftarrow \Psi \bigcup$ RecoverDataPath($pfx$, $d$);
    if $num\_path \geq k$ then
        break;
    end
    Spur($pfx$, $s$, $d$, $Q$);
end
return $\Psi$;
```



(a) Build the suffix graph: shortest distance to target

(b) Spur along the 1st critical path (post-CPPR = -12)

(c) Spur along the 2nd critical path (post-CPPR = -11)

(d) Spur along the 3rd critical path (post-CPPR = -10)

(e) Spur along the 4th critical path (post-CPPR = -8)

(f) Spur along the 6th critical path (post-CPPR = -4)

Figure 1.8: Exemplification of UI-Timer 1.0. (a) UI-Timer 1.0 builds a suffix tree in the initial iteration by finding the shortest path tree rooted at the target node. (b) During the first search iteration, four paths are spurred from the most critical path $\langle e_3, e_8, e_{12}, e_{15}\rangle$. (c) During the second search iteration, one path is spurred from the second critical path $\langle e_2, e_6, e_{14}\rangle$. (d) During the third search iteration, one path is spurred from the third critical path $\langle e_2, e_7, e_{12}, e_{15}\rangle$. (e) No path is generated from the fourth and fifth search iterations. (f) During the sixth search iteration, one path is spurred from the sixth critical path $\langle e_4, e_{10}, e_{13}, e_{15}\rangle$.

1.0 grows equivalently with the prefix tree, in which each state represents a path implicitly. *Spur* is responsible for neighboring expansion, iteratively including a set of new deviation edges as tree leaves or search frontiers. Since by definition all paths can be viewed as being deviated from the shortest path, the initial state is equivalent to the root of the prefix tree. Using a priority queue, the items or paths extracted are in the order of criticality.

Theorem 4: *UI-Timer 1.0 solves each hold test or setup test in space complexity O(nlogn + m + k) and time complexity O(nlogn + kn + kmlogk).*

Proof The space complexity of UI-Timer 1.0 involves *O(n + m)* for storing the circuit graph, *O(nlogn)* for lookup table, and *O(n)* for the suffix tree as well as *O(k)* for the prefix tree. As a result, the total space requirement is *O(nlogn + n + k)*. On the other hand, it takes up to k iterations on calling the procedure *Spur* in order to discover the top-k critical paths. Recalling that the lookup table is built in time *O(nlogn)* and the suffix tree can be constructed in time *O(n + m)* using topological relaxation, the time complexity of UI-Timer 1.0 is thus *O(nlogn + kn + kmlogk)*.

An exemplification is given in Figure 1.8. (a) illustrates a suffix tree derived by computing the shortest path tree rooted at the destination node from a given pessimism-free graph. (b) shows a total of four paths are spurred from the current-most critical path $p_1 = \langle e_3, e_8, e_{12}, e_{15} \rangle$ in the first search iteration. For instance, the path with deviation edge e_{11} has cumulative cost equal to $0 + (6 - 5 + 3) = 4$. The corresponding explicit path recovery is $\langle e_3, e_8, e_{11}, e_{14} \rangle$ as a result of combining the prefix of p_1 ending at the tail of e_{11} and the suffix from the suffix tree beginning at the head of e_{11}. On the other hand, the path with deviation edge e_1 has deviation cost equal to $0 + (7 - (-12) + 0) = 19$ which in turns tells the value of its post-CPPR slack being $-12 + 19 = 7$. Since the post-CPPR slack has been positive already, by lemma 3 the following search space can be pruned (node marked with a slash "/"). Accordingly in the end of this iteration, only three of the four spurred paths are explored as search frontiers from the parent path p_1. (c)–(f) repeat the same procedure except no more paths are spurred from the fourth and fifth search iterations.

1.7 Application to Multiple Tests

The architecture of UI-Timer 1.0 is developed on the basis of one test at one time. That is, each test is regarded as an independent input and has no dependence on each other. For applications where multiple tests are designated, a readily available parallel framework can be carried out by forking multiple threads with each operating on a subset of tests. With the shared lookup table and the circuit graph, we impose the least memory requirement by maintaining only private information about the suffix tree and the prefix tree for each thread. A number of tests with up to the maximum number of threads supported by the machine can be simultaneously processed. One multi-threaded application is presented in Algorithm 8, in which we sweep the test and report the top-k critical paths for each test.

```
Algorithm 8: SweepReport($\widehat{t}$, $k$)
Input: test vector $\widehat{t}$, path count $k$
Output: solution vector $\widehat{\Psi}$ of the top-$k$ critical paths for each test
BuildCreditLookupTable();
#Parallel for index i in range($\widehat{t}$) do
    $G_p^i \leftarrow$ pessimism-free graph for the test $\widehat{t}[i]$;
    $\widehat{\Psi}[i] \leftarrow$ GetCriticalPath($G_p^i.source$, $G_p^i.destination$, $k$);
end
return $\widehat{\Psi}$;
```

As opposed to the sweep report in Algorithm 8, the block report is another common application where probing the top-k critical paths across all timing tests is the main goal. We refer the criticality of a test to the slack value of the top most critical path extracted from this test. It is intuitive by set property that the top-k critical paths must exist in the path set generated from the top-k critical tests. Therefore, we first develop Algorithm 9 to peel the top-k critical tests out of a given test set. Algorithm 9 sweeps the test set and finds the most critical path for each test (line 1:4). The post-CPPR slack value of each path is used as the criticality of the corresponding test (line 5). A sorting procedure is then followed so as to peel out the top-k critical tests (line 7:9).

Using Algorithm 9, the function of the block report for the globally top-k critical paths is constructed in Algorithm 10. We first apply Algorithm 9 to

```
Algorithm 9: GetCriticalTest($\widehat{t}$, $k$)
   Input: test vector $\widehat{t}$, test count $k$
   Output: the set $\widehat{\Omega}$ of the top-$k$ critical tests
 1 BuildCreditLookupTable();
 2 #Parallel for $index\ i\ in\ range(\widehat{t})$ do
 3     $G_p^i \leftarrow$ pessimism-free graph for the test $\widehat{t}[i]$;
 4     $p \leftarrow$ GetCriticalPath($G_p^i.source$, $G_p^i.destination$, 1);
 5     $t.criticality \leftarrow p.slack$;
 6 end
 7 sort $\widehat{t}$ according to criticality;
 8 $\widehat{\Omega} \leftarrow$ top-$k$ tests in $\widehat{t}$;
 9 return $\widehat{\Omega}$;
```

peel out the top-k critical tests (line 1). Since it has been shown that the globally top-k critical paths must be investigated from these tests, we iteratively extract the top-k critical paths from each of the top-k critical tests (line 3:10). An efficient min-max priority queue [20] is employed to dynamically maintain the solution paths (line 2) and prune unnecessary search (line 4:6).

```
Algorithm 10: BlockReport($\widehat{t}$, $k$)
   Input: test vector $\widehat{t}$, path count $k$
   Output: the set $\widehat{\Psi}$ of the globally top-$k$ critical paths across $\widehat{t}$
 1 $\widehat{\Omega} \leftarrow$ GetCriticalTest($\widehat{t}$, $k$);
 2 $Q \leftarrow$ priority queue keyed on slack values;
 3 for $t \in \widehat{\Omega}$ do
 4     if $Q.size = k$ and $t.criticality \geq Q.top\_max$ then
 5         break;
 6     end
 7     $G_p^t \leftarrow$ pessimism-free graph for the test $t$;
 8     $Q \leftarrow Q \cup$ GetCriticalPath($G_p^t.source$, $G_p^t.destination$, $k$);
 9     $Q.maintain\_top\_k\_min(k)$;
10 end
11 $\widehat{\Psi} \leftarrow$ paths from the priority queue $Q$;
12 return $\widehat{\Psi}$;
```

Theorem 5: *The function SweepReport in Algorithm 8 takes O(nlogn +*

$|\widehat{t}|(kn + kmlogk) / C)$ time complexity, where $\widehat{t}$ is the input test vector and C is the number of available cores or threads.

Proof Algorithm 8 exerts the core procedure of UI-Timer 1.0 on a given test vector $\widehat{t}$. A sequential version hence takes $O(nlogn + |\widehat{t}|(kn + kmlogk))$ time complexity. Notice that the lookup tables for CPPR credit only needs one-time building, which takes $O(nlogn)$ time complexity. Running Algorithm 8 in a machine with C cores or C threads supports a parallel reduction by up to a factor of C. Therefore, the runtime complexity of sweep report is $O(nlogn + |\widehat{t}|(kn + kmlogk) / C)$.

Theorem 6: *The function GetCriticalTest in Algorithm 9 takes $O(nlogn + (n + m) / C + |\widehat{t}|log|\widehat{t}| + k)$ time complexity, where $\widehat{t}$ is the input test vector and C is the number of available cores or threads.*

Proof The first section (before sorting) of Algorithm 9 is nearly the same as Algorithm 8, except that only the single most critical paths is generated. Therefore, the time complexity is $O(nlogn + |\widehat{t}|(n + m) / C)$. Afterwards, sorting the test vector $\widehat{t}$ takes $O(|\widehat{t}|log|\widehat{t}|)$ time complexity and outputting the top-k critical tests takes linear time complexity $O(k)$. Hence, the entire runtime complexity of Algorithm 9 is $O(nlogn + (n + m) / C + |\widehat{t}|log|\widehat{t}| + k)$.

Theorem 7: *The function BlockReport in Algorithm 10 takes $O(nlogn + (n + m) / C + |\widehat{t}|log|\widehat{t}| + k^2n + k^2mlogk)$ time complexity, where $\widehat{t}$ is the input test vector and C is the number of available cores or threads.*

Proof Algorithm 10 first calls Algorithm 9 to obtain the top-k critical tests from a given test vector $\widehat{t}$, which takes $O(nlogn + (n + m) / C + |\widehat{t}|log|\widehat{t}| + k)$ time complexity. Generating the globally top-k critical paths involves k iterations calling Algorithm 7. Besides, each iteration requires k logarithmic operations in order to maintain the top-k critical paths in the priority queue. The time complexity of each iteration is thus $O(kn + kmlogm + klogk)$. As a result, the total time complexity of the block report is $O(nlogn + (n + m) / C + |\widehat{t}|log|\widehat{t}| + k^2n + k^2mlogk)$.

1.8 Implementation and Technical Details

In this section, we highlight two implementation techniques that are practical for the improvement of runtime performance, despite not reducing the theoretical bound. It is observed from the program profiler that the majority of the runtime is spent on the construction of suffix tree, which is equivalent to finding the shortest path tree in the pessimism-free graph. The shortest path routines such as storage initialization, distance relaxation, and fanin/fanout scanning typically exhibit a wild and deep swing in the search space and consume a huge amount of CPU instructions. The problem becomes even critical when multiple tests are taken into account. To remedy this problem, two verified trials are worth delivering.

1.8.1 Memory Pool for Storage Initialization

Constructing the suffix tree is equivalent to discovering the shortest path tree rooted at the target node of the pessimism-free graph. A generic framework of any shortest path algorithms requires two data arrays, *distance* and *successor*, for storing the distance labels and shortest path tree connection, respectively [21]. Before the relaxation on distance labels takes effect, programmer should clear the two arrays by assigning an infinite value to every distance entry and a nil value to every successor entry. Nonetheless, real applications come with multiple tests. This linear procedure will be repeated for each test and the accumulative runtime becomes non-negligible. Furthermore, in most cases each test involves only a small portion of the entire circuit graph in labeling process. It is desirable to clear those entries ever participating in the previous search. To this end, we pre-allocate a memory pool for *distance* and *successor* arrays and clear their memory values in the very beginning. We also keep track of those entries whose values were ever modified in the course of shortest path routines and clear these entries by the end of function return. As a consequence, the computational effort on storage initialization can be minimized.

1.8.2 Redundant Search Space Pruning

Reducing the size of suffix tree is another effective way to decrease the runtime, and it can be beneficial for the later search on prefix paths. Since we consider only violating points, any suffix paths discovered so far with positive value can be discarded so as to prune the subsequent search space. In the course of shortest path search, the worst timing quantities at a given pin (which can be precomputed) provide a lower bound and a upper bound on the minimum hold and maximum setup path slack that are reachable from this pin. An A*-like pruning strategy can thus be employed, as presented in Algorithm 11. Notice that without loss of generality one can replace the cutoff value with any user-specified slack threshold and this has no impact on the overall correctness subject to a proper implementation of shortest path algorithms.

```
Algorithm 11: is_prunable($m$, $p$, $dis$)
  Input: test type $m$, a pin $p$, a distance array $dis$
  Output: true if $p$ is prunable from the suffix tree or false otherwise
if $m =$ HOLD then
    if $dis[p] + at_p^{early} \geq cutoff$ then
        return true;
    end
end
if $dis[p] - at_p^{late} \geq cutoff$ then
    return true;
end
return false;
```

Lemma 5: *The pruning strategy in Algorithm 11 is correct, meaning that the derived suffix tree contains no path suffix which has a slack value larger than the given cutoff value.*

We have proved that the cost of any source-destination path in the pessimism-free graph is identical to the slack value of the corresponding data path. In hold time test, the distance value of a pin p, denoted as $dis[p]$, represents the potential slack value discovered so far from the destination. The earliest arrival time at this pin, denoted as at_p^{early}, is the minimum delay that will be added for any complete data paths suffixed at the pin p. That is, the slack values of such paths are lower-bounded by $dis[p] + at_p^{early}$ and any search points exceeding the cutoff values can be pruned. The proof for the setup time test can be drawn in a similar way.

1.9 Experimental Results

UI-Timer 1.0 is implemented in C++ language on a 2.67 GHz 64-bit Linux machine with 8 GB memory. The application programming interface (API) provided by OpenMP 3.1 is used for our multi-thread parallelization [22]. Our machine can execute a maximum of four threads concurrently. Experiments are undertaken on a set of circuit benchmarks released from the TAU 2014 CAD contests [1]. The benchmarks are modified from well-known industrial circuits (e.g., s27, s510, systemcdes, wb_dma, pci_bridge32, vga_lcd, etc.) that have been released to the public domain for research purpose. Statistics of these circuits are summarized in Table 2.1. All benchmarks are associated with multiple tests. The three largest circuits, Combo5, Combo6, and Combo7, have million-scale graph data. For example, the circuit Combo6 has 3577926 pins and 3843033 edges.

1.9.1 Effectiveness of CPPR

Figure 1.9 depicts the impact of CPPR on hold and setup test slacks for circuits des_perf and vga_lcd. The horizontal and vertical axes in the plots denote the pre-CPPR slack and the post-CPPR slacks, respectively. Each plot is attached a reference line with slope 1.0 indicating the identical slacks. It is observed that each post-CPPR slack is at least the pre-CPPR slack value and most post-CPPR slack values are improved. The plots indicate the effectiveness of CPPR during design closure from designers' perspective. The synthesis and optimization tools can focus their efforts on true timing-critical paths and optimize these paths only by the amount necessary to meet the target clock frequency of the chip.

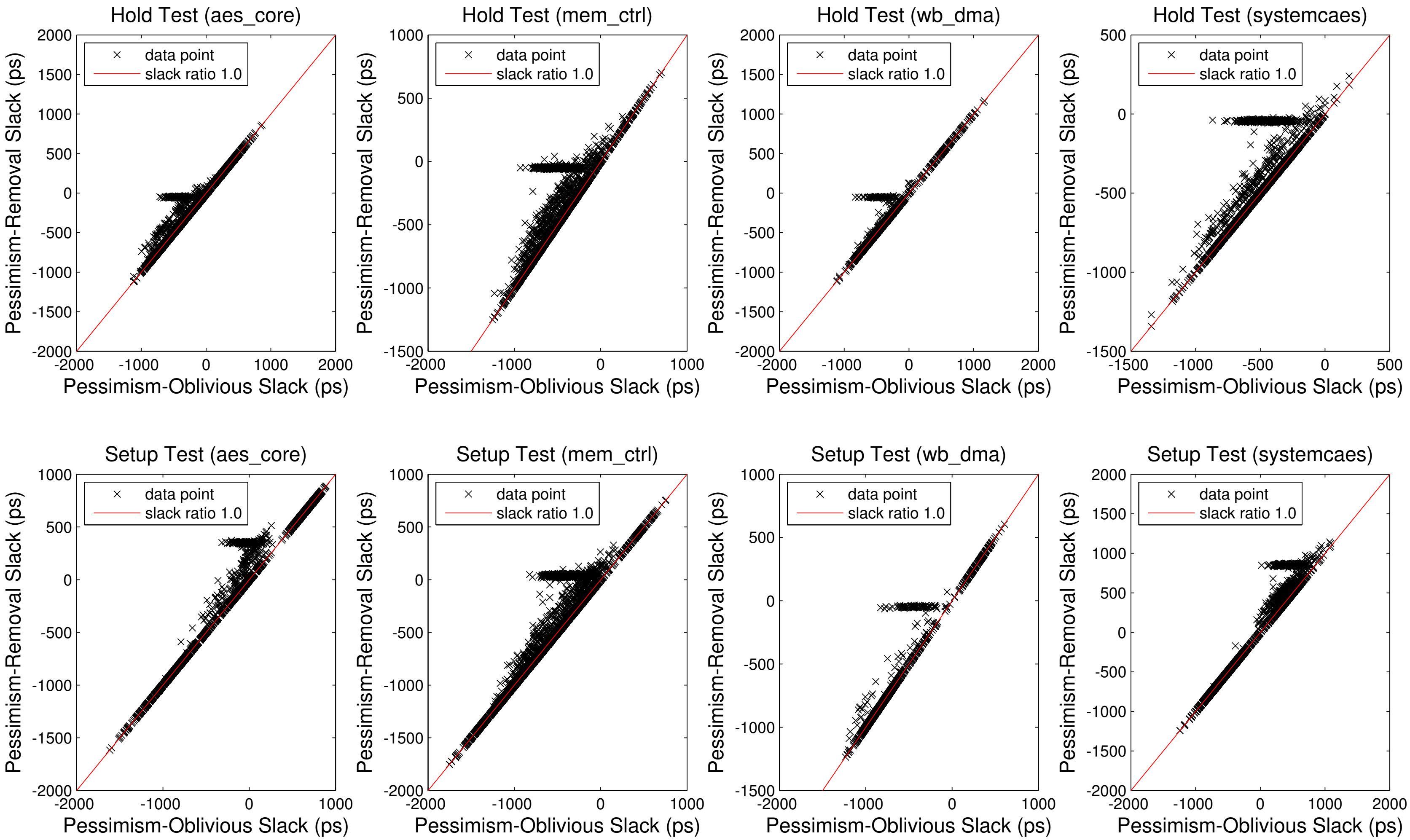


Figure 1.9: Impact of CPPR on hold and setup time slacks for circuits aes_core, mem_ctrl, wb_dma, and systemcaes. Data points are sampled based on the worst pre-CPPR slack value of each test.

Table 1.2: Comparison Between UI-Timer 1.0 and the Top-3 Winners, Timer-1st, Timer-2nd, and Timer-3rd from the TAU 2014 CAD Contest [4]

Circuit	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert C\rvert$	# Tests	# Paths	Timer-2nd			Timer-3rd		Timer-1st		UI-Timer 1.0	
						AER	MER	CPU	AER	CPU	AER	CPU	AER	CPU
s27	109	112	6	6	9	9.97	50.00	0.20	0	0.40	0	0.20	0	0.01
s344	574	658	16	11	11	0	0	0.22	0	0.53	0	0.22	0	0.02
s349	598	682	16	11	11	0	0	0.25	0	0.53	0	0.22	0	0.02
s386	570	701	7	9	7	0	0	0.20	0	0.49	0	0.20	0	0.02
s400	708	813	22	5	6	0	0	0.23	0	0.56	0	0.21	0	0.02
s510	891	1091	7	21	7	0	0	0.18	0	0.40	0	0.18	0	0.01
s526	933	1097	22	5	6	0	0	0.25	0	0.56	0	0.22	0	0.02
s1196	1928	2400	19	16	14	0	0	0.25	0	0.59	0	0.22	0	0.01
s1494	2334	2961	7	10	19	0	0	0.25	0	0.58	0	0.21	0	0.02
systemcdes	10826	13327	1967	380	41436	6.79	32.89	2.27	0	3.62	0	0.14	0	0.09
wb_dma	14647	17428	5218	1374	158	7.46	39.30	0.23	0	0.90	0	0.28	0	0.19
tv80	18080	23710	3608	838	19227963	8.20	43.49	32.38	0	23.13	0	0.23	0	0.23
systemcaes	23909	29673	6643	2500	13069928	6.53	29.92	33.23	0	22.44	0	0.62	0	0.37
mem_ctrl	36493	45090	10638	3754	62938	5.41	24.73	0.65	0	3.71	0	0.83	0	0.52
ac97_ctrl	49276	55712	22223	9370	148	-	-	-	0	2.95	0	1.31	0	0.69
usb_funct	53745	66183	17665	4392	129854	6.43	37.87	0.94	0	5.64	0	1.41	0	0.78
pci_bridge32	70051	78282	33474	16450	17296	5.04	25.49	2.27	0	14.49	0	4.71	0	2.91
aes_core	68327	86758	5289	2528	21064	6.72	31.70	0.68	0	4.46	0	0.96	0	0.62
des_perf	330538	404257	88751	19764	1682	4.60	11.89	3.37	0	18.37	0	19.24	0	6.25
vga_lcd	449651	525615	172065	50182	5281	7.94	43.21	16.78	0	119.24	0	159.15	0	30.19
Combo2	260636	284091	171529	29574	62938	4.70	24.07	9.19	0	49.00	0	56.12	0	13.67
Combo3	181831	284091	73784	8294	129854	6.71	35.14	3.39	0	20.30	0	11.35	0	4.53
Combo4	778638	866099	469516	53520	19227963	7.93	42.13	205.69	0	557.81	0	333.04	0	78.10
Combo5	2051804	2228611	1456195	79050	19227963	-	-	-	N/A	> 3 hrs	0	1225.50	0	226.47
Combo6	3577926	3843033	2659426	128266	19227963	-	-	-	N/A	> 3 hrs	0	3544.04	0	544.36
Combo7	2817561	3011233	2136913	109568	19227963	-	-	-	N/A	> 3 hrs	0	2485.81	0	464.68

$\lvert V\rvert$: size of node set. $\lvert E\rvert$: size of edge set. $\lvert C\rvert$: size of clock tree. # Tests: # of setup tests and hold tests. # Paths: max # of data paths per test.
AER/MER: avg/max error rate of mismatched paths (%). CPU: avg program runtime (seconds). -: unexpected program fault.

1.9.2 Comparison with TAU 2014 CAD Contest Winners

We first compare UI-Timer 1.0 with the final entries in the TAU 2015 CAD contest. Adhering to contest rules, we ran the timer for each circuit benchmark with different path counts k from 1 to 20 across all setup and hold tests and collected averaged quantities on runtime and accuracy for comparison. The accuracy is measured by the percentage of mismatched paths to a golden reference generated by an industrial timer [4, 1]. Table 2.1 lists the overall performance of UI-Timer 1.0 in comparison to the top-3 timers, "Timer-1st", "Timer-2nd", and "Timer-3rd", for short, from the TAU 2014 CAD contest [4]. For fair comparison, all timers are run in the same environment with four threads.

We begin by comparing UI-Timer 1.0 with Timer-2nd. The strength of UI-Timer 1.0 is clearly demonstrated in the accuracy value. Our timer achieves exact accuracy yet Timer-2nd suffers from many path mismatches. The highest error rate is observed in the smallest design s27. Unfortunately, we are unable to report experimental data of ac97_ctrl, Combo5, Combo6, and Combo7, because Timer-2nd encounters execution faults. It is expected that Timer-2nd is faster in some cases as they sacrifice the accuracy for speed. However, the performance margin of Timer-2nd can be up to ×141.78 worse than UI-Timer 1.0 in circuit tv80 (i.e., 32.38 vs 0.23) while the counterpart of UI-Timer 1.0 is more competitive by at most ×1.85 slower in des_perf (i.e., 3.37 vs 6.25). As a result, the solution quality of UI-Timer 1.0 is more stable and reliable, especially for high-frequency designs where accuracy is the top priority of timing-specific optimizations.

Next we compare UI-Timer 1.0 with Timer-3rd and Timer-1st. In general, full accuracy scores are observed for all timers, while UI-Timer 1.0 reaches the goal far faster than the others. It can be seen that Timer-3rd suffers from significant runtime overhead across nearly all benchmarks and fails to accomplish the three largest designs, Combo5, Combo6, and Combo7, within 3 hours. Compared to Timer-1st, the first-place winner in the TAU 2014 CAD Contest, our Timer achieves fairly remarkable speedup across all benchmarks. For example, our timer reaches the goal by ×22.0, ×5.3, and ×6.5 faster than Timer-1st in circuits s1196, vga_lcd, and Combo6, respectively. Similar trend can be found in other cases as well. The speedup curve becomes more pronounced for large circuits. In terms of memory profiling, we did not see too much difference between UI-Timer 1.0 and other entires. All computations are able to fit into the main memory with less than 1GB.

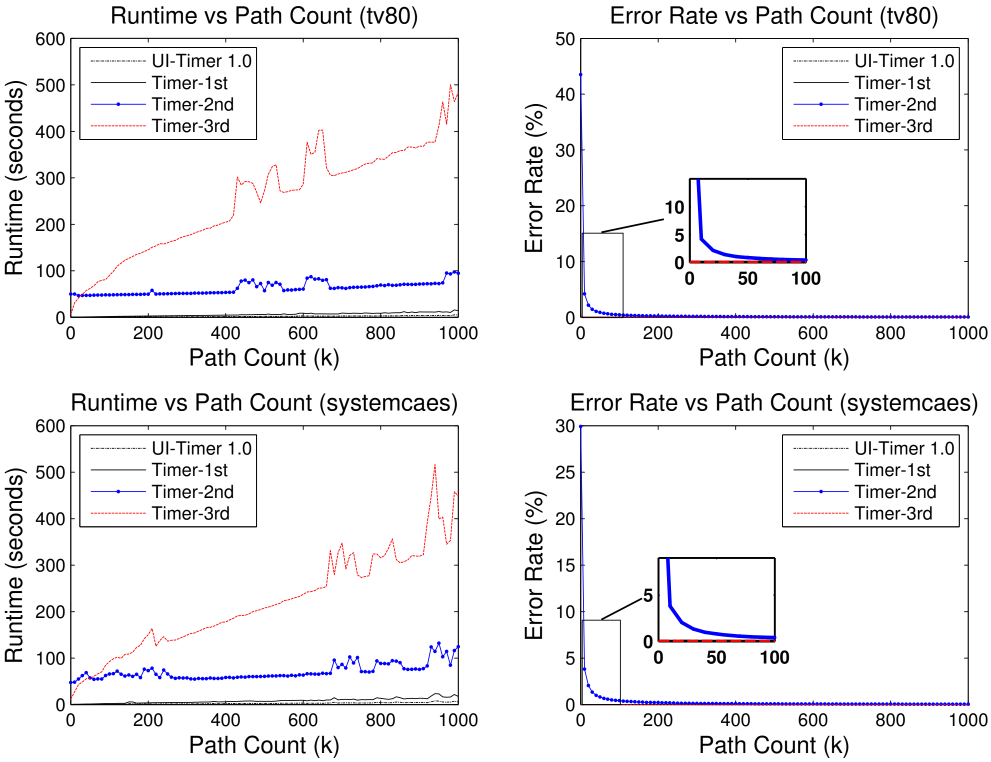


Figure 1.10: Performance comparison of UI-Timer 1.0, Timer-1st, Timer-2nd, and Timer-3rd for circuits tv80 and systemcaes.

We investigate the scalability of UI-Timer 1.0 by varying the input parameter, the path count k, from 1 to 1000. The performance comparing UI-Timer 1.0 with the top-3 entires, Timer-1st, Timer-2nd, and Timer-3rd on two example circuits, tv80 and systemcaes, is characterized in Figure 1.10. We see all runs are accomplished instantaneously by UI-Timer 1.0 and the runtime gap to the other timers becomes clear as path count grows. Take the point of 980 paths for example. UI-Timer 1.0 consumes only 3.41 seconds while the runtime values for Timer-1st, Timer-2nd, and Timer-3rd are 10.38 seconds, 93.25 seconds, and 500.26 seconds, respectively. With regard to accuracy, our timer is always exact and confers a fundamental difference to Timer-2nd which sacrifices accuracy for speedup.

Finally we give a scatter plot showing the runtime growth of UI-Timer 1.0 versus the design size in Figure 1.11. The measurement is taken over the open core series (systemcdes, wb_dma, etc.) and the combo series (Combo2, Combo3, etc.). We approximate the design size using discrete quantity on

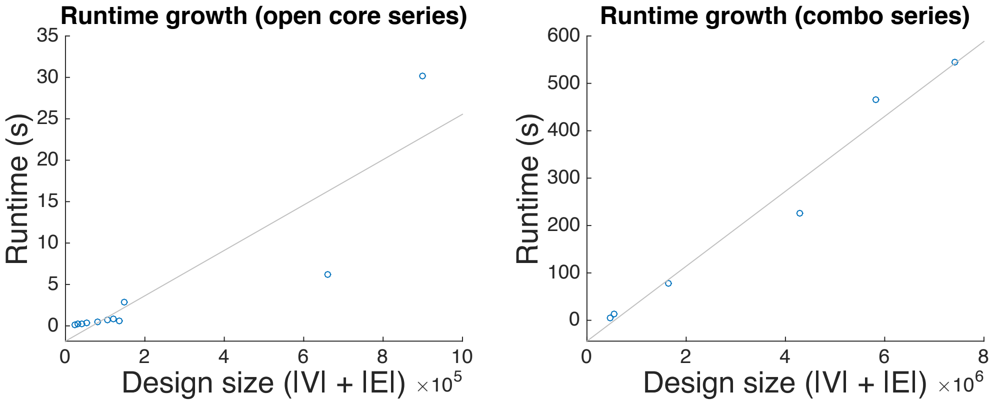


Figure 1.11: Scatter plot on runtime growth and design size for UI-Timer 1.0.

the total number of nodes and edges in the circuit graph. It is convinced by the least square reference line that the runtime of UI-Timer 1.0 grows linearly with respect to the increase of design size. One can indirectly infer the amount of runtime needed for larger designs.

1.9.3 Comparison with the State-of-the-Art Timer

We have seen the superior performance of UI-Timer 1.0 in comparison to the top-ranked timers in the TAU 2014 timing analysis contest. Ever since the contest was concluded, a few following works demonstrating promising results have been published in recent years [13, 11, 12]. We are particularly interested in the comparison with the timer, "iTimerC" [12], as it presented significant improvement to the contest winners. We observed both timers, iTimerC and UI-Timer 1.0, performed very well and achieved close results based on the TAU 2014 contest environment. In order to discover the performance margin, we enhance the difficulty and the scale of this experiment on the six largest benchmarks, Combo2–Combo7. Each timer is requested to peel out the top-50 critical tests and report the top-2000 critical paths for each of the tests. In other words, evaluation is undertaken under an extreme condition in which reporting a high number of critical paths over a subset of critical tests is the goal.

The performance comparison between UI-Timer 1.0 and iTimerC [12] is presented in Table 1.3. It can be seen that UI-Timer 1.0 achieves highly scalable and reliable performance when the design size and query difficulty scale

Table 1.3: Comparison Between UI-Timer 1.0 and iTimerC [12]

Circuit	Type	iTimerC [12]		UI-Timer 1.0	
		AER	CPU	AER	CPU
Combo2	hold	0	4.20	0	2.77
Combo2	setup	0	12.94	0	11.35
Combo3	hold	0	3.98	0	1.39
Combo3	setup	0	10.08	0	8.16
Combo4	hold	0	14.09	0	14.38
Combo4	setup	0	73.91	0	24.21
Combo5	hold	0	1334.24	0	47.20
Combo5	setup	unknown	> 1 hr	0	59.01
Combo6	hold	unknown	> 1 hr	0	130.60
Combo6	setup	unknown	> 1 hr	0	127.59
Combo7	hold	unknown	> 1 hr	0	88.91
Combo7	setup	unknown	> 1 hr	0	110.90

AER: avg error rate of mismatched paths (%). CPU: runtime (s).

up. The higher runtime in setup test is expected because most critical paths come from the violation of setup constraint. Our runtime is superior in almost all testcases. We have observed significant runtime speedup to iTimerC by more than an order of magnitude for million-scale graphs, Combo5, Combo6, and Combo7. Considering the hold tests in Combo5, UI-Timer 1.0 requires only 47.20 seconds which is ×28.27 faster than that by iTimerC. For the rest of million-scale graphs, our timer is able to analyze the timing by less than 3 minutes, whereas iTimerC cannot finish the program within 1 hour. These results have justified the practical viability of our timer.

1.9.4 Search Space Pruning through Slack Cutoff

Due to the high complexity of CPPR, modern industrial timers, in practice, apply various cutoff slack strategies to prune the search space. For example, the number of CPPR branching points can be controlled by some tolerance or threshold values so as to reduce the runtime and memory. As aforementioned, one important feature of UI-Timer 1.0 is the ease to control the slack margin, which has the potential to affect the number of paths generated during CPPR. By default, UI-Timer 1.0 reports negative slack and such cutoff value can be easily tuned since every path is (1) implicitly represented in constant time and space, and (2) generated in increasing order of post-CPPR slack values.

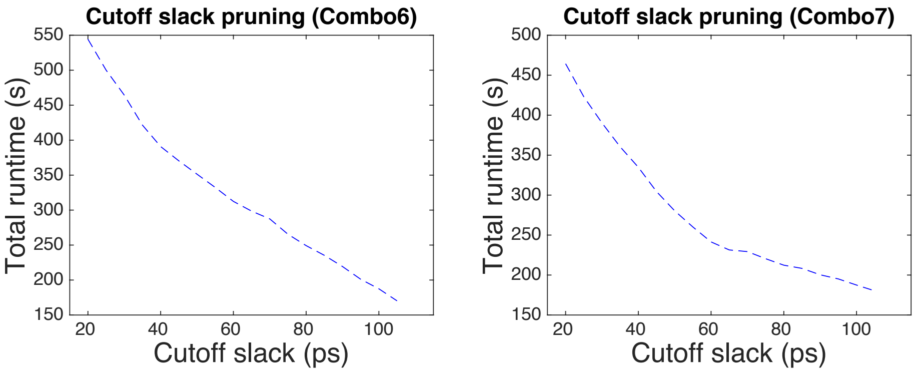


Figure 1.12: Runtime reduction curve under different slack cutoff values.

The runtime reduction under different cutoff slack values is plotted in Figure 1.12. We run experiments with five cutoff slack values, 20 ps, 40 ps, 60 ps, 80 ps, and 100 ps on the two largest benchmarks, Combo6 and Combo7. It is expected that the runtime decreases as the cutoff slack values increase. The higher the cutoff slack value is, the less the search space is spanned by path ranking. In spite of higher pessimism (less CPPR credit), the curve can be an useful indicator in striking a balance between program runtime and pessimism margin.

1.9.5 Extension to Distributed Computing

We have performed an extra evaluation on a distributed system running the three largest cases, Combo5, Combo6, and Combo7, in order to further demonstrate the scalability of our program. UI-Timer 1.0 is advantageous in handling every timing test independently. In distributed environment, multiple tests can be evenly partitioned into groups with respect to the number of cores. Each group is then assigned to one computing node and is analyzed by the timer independently. The application programming interface (API) provided by OpenMPI 1.6.5 is used as our message passing interface for distributed computing [23]. The evaluation is taken on a computer cluster having over 500 compute nodes with each configured with 16 Intel E5-2670 2.60GHz cores and 128GB RAM. The network infrastructure is 384-port Mellanox MSX6518-NR FDR InfiniBand for high speed cluster interconnect [24].

We begin by demonstrating the runtime performance versus the number of

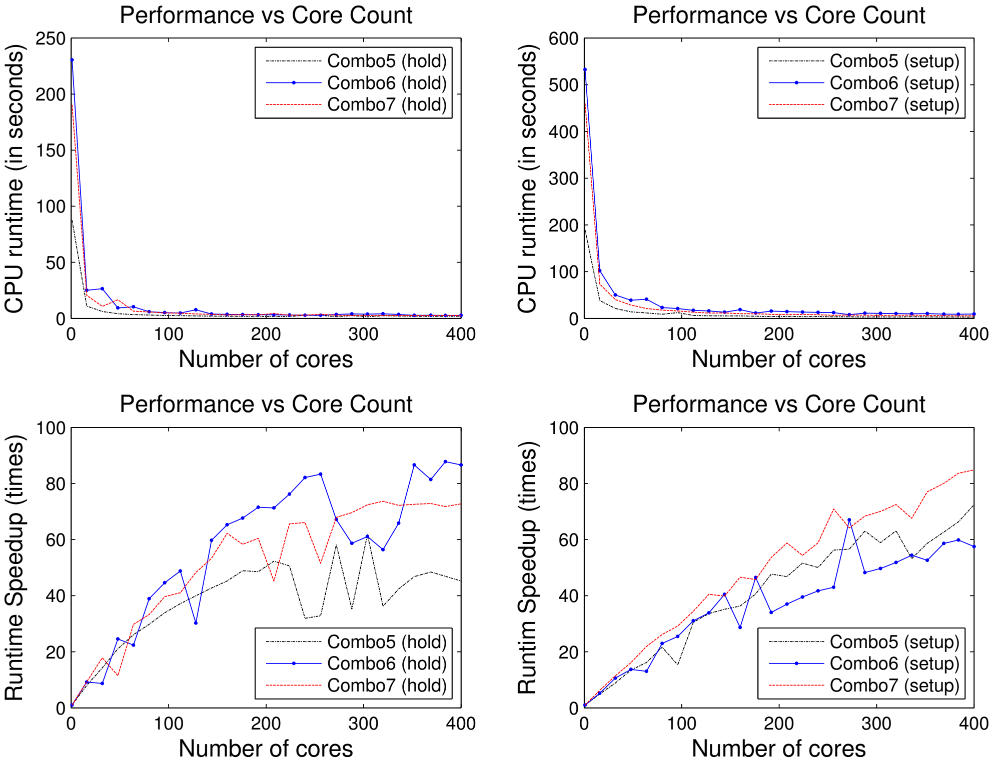


Figure 1.13: Runtime and speedup curves of hold tests and setup tests for benchmarks Combo5, Combo6, and Combo7 on a distributed system.

cores that is invoked for running our program. The core count is varied from 1 to 400 and the runtime is measured by a synchronized moment at which all process cores complete their jobs (i.e., reading the file, passing message, and handling all algorithmic procedures). The performance is interpreted in terms of the runtime and its relative speedup to a baseline which was run in single-core execution. Figure 1.13 shows the performance plot of this evaluation. It can be clearly seen that the runtime is reduced drastically as the number of cores increases. For example, the setup tests of Combo6 are accomplished by less than 1 minute with 16 cores, obtaining ×5.23 speedup to the single-core execution (266.29 vs 50.95). Similar speedup curve is also present in other testcases. In a single minute, hold tests and setup tests of all testcases are solvable using only 16 cores.

1.10 Conclusion

In this chapter we have presented UI-Timer 1.0, an exact and ultra-fast algorithm for handling the CPPR problem during static timing analysis. Unlike existing approaches which frequently use exhaustive path search with case-by-case heuristics, our timer maps the CPPR problem to a graph-theoretic formulation and applies an efficient search routine using a highly compact and efficient data structure to obtain an exact solution. We have highlighted important features of UI-Timer 1.0 such as simplicity, coding ease, and most importantly the theoretically-proven completeness and optimality. Comparatively, experimental results have demonstrated the superior performance of UI-Timer 1.0 in terms of accuracy and runtime over existing timers.

We shall discuss how we extend UI-Timer 1.0 in Chapter 2 to deal with incremental timing analysis with CPPR [25]. Various stages of the design flow such as logic synthesis, placement, routing, physical synthesis, and optimization facilitate a need for incremental timing analysis. The performance of incremental timing with CPPR plays a key role in the success of timing optimizations. Due to the path-specific property of CPPR, CPPR-aware incremental timing has emerged as one of the major challenges in existing timing analysis tools [9]. A high-quality CPPR-aware incremental timer is definitely important to speed up the timing closure.

CHAPTER 2

OPENTIMER: A HIGH-PERFORMANCE TIMING ANALYSIS TOOL

2.1 Introduction

The lack of accurate and fast algorithms for high-performance timing analysis tool with incremental capability has been recently pointed out as a major weakness of existing timing optimization flows [9]. In the deep submicron era, timing-driven operations are imperative for the success of optimization flows. Optimization transforms change the design and therefore have the potential to significantly affect timing information. The timer must reflect such changes and update timing information incrementally and accurately in order to ensure slack integrity as well as reasonable turnaround time and performance [2]. However, such a process requires extremely high complexity especially when path-based analysis is configured [1, 8, 12]. A high-quality incremental timer capable of path-based analysis is definitely advantageous in speeding up the timing closure.

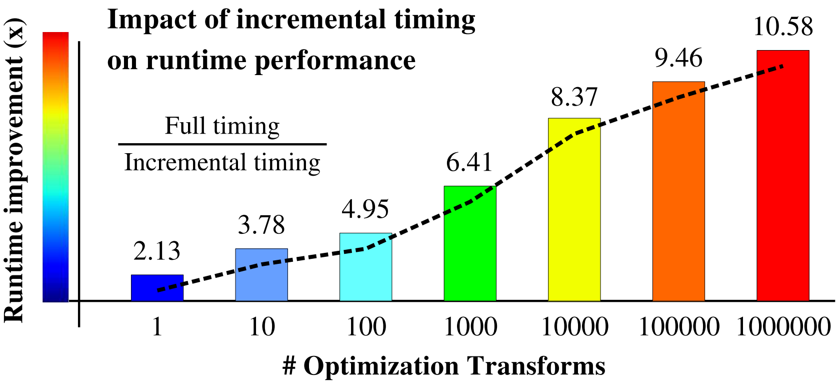


Figure 2.1: Performance improvement of incremental timing to full timing on one benchmark from [9].

The significance of incremental timing is demonstrated in Figure 2.1. It is observed that the runtime improvement keeps growing as the number of optimization transforms increases. One obvious reason is that once the critical paths in a design have been reported, the optimization tool would optimize the logic (e.g., gate sizing, buffer insertion) so as to overcome the timing violations. This subtle change can affect up to the majority of a circuit, whereas in reality, depending on the trace of critical paths, the timing update may only involve a small portion of the circuit. Since an optimization tool can perform millions of logic transformations, it is important that the timing profile is kept up-to-date in an incremental fashion. Otherwise, optimization tools cannot support fast turnaround for timing-specific improvement, which dramatically degrades the productivity.

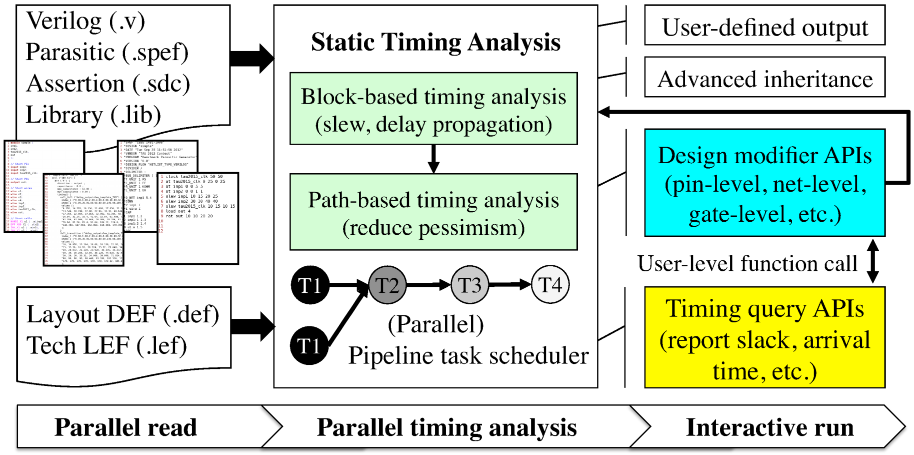

Verilog (.v)
Parasitic (.spef)
Assertion (.sdc)
Library (.lib)
Layout DEF (.def)
Tech LEF (.lef)
Static Timing Analysis
Block-based timing analysis (slew, delay propagation)
Path-based timing analysis (reduce pessimism)
T1
T2
T3
T4
T1
(Parallel) Pipeline task scheduler
User-defined output
Advanced inheritance
Design modifier APIs (pin-level, net-level, gate-level, etc.)
User-level function call
Timing query APIs (report slack, arrival time, etc.)
Parallel read
Parallel timing analysis
Interactive run

Figure 2.2: The software architecture of OpenTimer.

Besides being incremental, one important feature of a practical timer is the capability of common path pessimism removal (CPPR). CPPR is a path-specific timing update that intends to remove redundant pessimism incurred by common segments between data paths and clock paths. Unwanted pessimism might force designers and optimization tools to waste an unnecessary yet significant amount of effort on fixing paths that meet the intended clock frequency. This problem becomes even more critical when design comes to the deep submicron era where data paths are shorter, clocks are faster, and clock networks are longer to accommodate larger and complex chips. However, the real problem is the amount of pessimism that needs to be removed is path-specific. Computational complexity and space requirements for CPPR typically grows exponentially as the design size increases, not to mention the challenge in conjunction with incremental timing analysis. Consequently, in this chapter we introduce OpenTimer, an open-source high-performance timing analysis tool. An overview of OpenTimer is shown in Figure 2.2. We highlight three key features of OpenTimer as follows:

- **Parallel framework.** OpenTimer applies a pipeline task scheduler as the central engine. Critical tasks such as timing propagation and endpoint slack calculation are scheduled into the pipeline to overlap their runtimes.
- **Incremental capability.** OpenTimer precisely and minimally captures the features that are key to incremental timing. With lazy evaluation, we are able to keep computation minimum.
- **Path-based analysis.** OpenTimer represents the path implicitly using efficient and compact data structure, yielding a significant saving in both search space and search time for CPPR.

The effectiveness and efficiency of our timer have been evaluated on a set of industry benchmarks released from the TAU 2015 CAD contest. Compared to the top performers in the TAU 2015 CAD contest, OpenTimer confers a high degree of differential in nearly all aspects. The source code of OpenTimer has been released to the public domain for promoting further research.

2.2 Incremental Timing Analysis and CPPR

Various stages of the design flow such as logic synthesis, placement, routing, physical synthesis, and optimization facilitate a need for incremental timing analysis [9]. During these stages, local operations such as gate sizing, buffer insertion, or net rerouting can modify small fractions of the design and significantly change both local and global timing landscapes. As the example shown in Figure 2.3, a change on gate B3 has the potential to affect up to the majority of the circuit (downstream timing). Nevertheless, depending on the trace of critical paths, only a small portion of the timing would need to be updated. For instance, if such a change does not affect the arrival time at I1:o, then every downstream timing after I1:o is unaffected.

In addition to incremental processing, the capability of CPPR is another important component for modern timing analysis tools. Optimization trans-

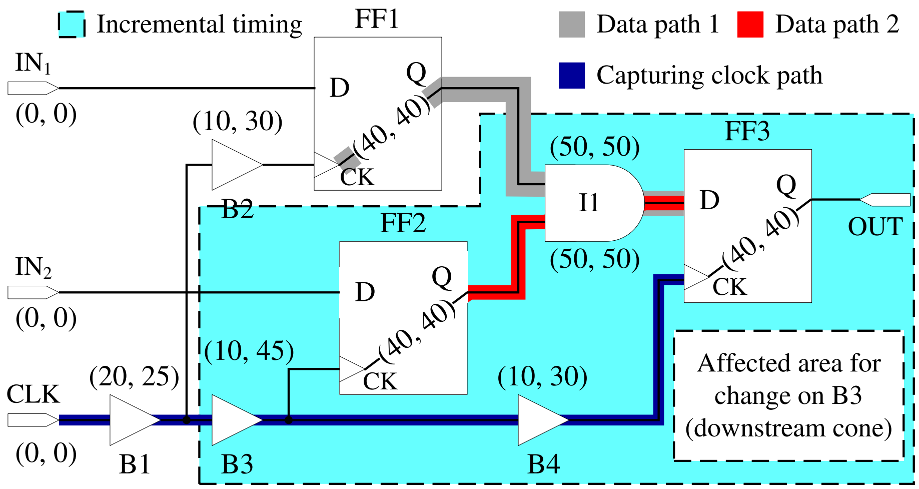


Figure 2.3: An example of sequential circuit network.

forms on the data network have no impact on CPPR credit (or CPPR adjustment) for any given launch-capture flip-flop (FF) pairs. Because the clock paths are not changed, any cached value for CPPR credit can be reused. However, in reality many optimization transforms are applied to the clock network, such as resizing a buffer or adding or deleting buffers on the clock tree in order to meet slack or skew targets. These changes can potentially affect a large number of data paths and slacks, and these data points must be recomputed with updated CPPR credits. Further, in some cases, changes on the clock network may not even impact CPPR for any data paths at all. As the example shown in Figure 2.3, the change on B3 can impact the CPPR credit for the launch-capture FF pair FF2 and FF3, while a change on B4 does not affect the CPPR credit for any FF pair. Therefore, the challenge of incremental CPPR is correctly identifying what data points are affected by which changes in an incremental manner.

2.3 Tool Configuration

OpenTimer follows the industry format to analyze the timing of your designs. The industry-standard format for timing analysis requests the following input files.

- **Two liberty (.lib) files** that defines the *early* and *late* characteristics of available cells in a given design, including pin capacitance, delay and slew look-up tables (LUTs), and setup/hold timing guards for sequential elements.

- **A verilog (.v) file** that defines the net list and circuit topology in the gate level for a given design, including primary input/output ports and connections among gates.
- **A parasitics (.spef) file** that defines the design parasitics of a set of nets as a resistive-capacitive (RC) network, including the capacitance of internal nodes and wire resistance between internal nodes.
- **A Synopsys design constraint (.sdc) file** that defines the design operating conditions, including the clock port, clock period, initial timing on primary input ports, and load capacitance of primary output ports.

Given these input files, we develop a CPPR-aware incremental timer that supports incremental timing updates subject to a set of design modifiers and reports the timing with CPPR of any queried data path or timing point. The slack prior to and after CPPR is referred to as *pre-CPPR slack* and *post-CPPR slack*, respectively. Particularly, the timer adheres to the following operations:

- **insert_gate**: adds an unconnected gate.
- **repower_gate**: changes the size of a gate.
- **remove_gate**: removes a disconnected gate.
- **insert_net**: creates an empty net.
- **remove_net**: removes a net from the design.
- **read_spef**: asserts parasitics on existing nets.
- **disconnect_pin**: disconnects a pin from its net.
- **connect_pin**: connects a pin to a net.
- **report_at**: reports the arrival time at a pin for any rise/fall transition and early/late split.
- **report_rat**: reports the required arrival time at a pin for any rise/fall transition and early/late split.
- **report_slack**: reports the worst post-CPPR slack at a pin for any rise/fall transition and early/late split.

- **report_worst_paths**: reports the worst post-CPPR path either in the design or through a specified pin.

The first eight operations describe the gate-level, net-level, and pin-level modifications on the design topology. The last four operations probe the design to report timing information. In order to collaborate with optimization tools, the timer should process these operations in an interactive or online manner. That is, advanced input disclosure or offline preprocessing is prohibited.

2.4 Algorithm

The overall framework of OpenTimer is presented in Algorithm 12. It first initializes the circuit based on input liberty, verilog, parasitic, and Synopsys design constraint files. Then it enters the interactive while loop, reading the operation commands and processing each command accordingly.

Algorithm 12: OpenTimer(.lib, .v, .spef, .sdc)

```
Input: .lib, .v, .spef, .sdc files
initialize the circuit from input .lib, .v, .spef, and .sdc files;
while $op \leftarrow GetOperationCommand$ do
    process the operation command $op$ accordingly;
end
```

2.4.1 State of the Art: UI-Timer

OpenTimer is built upon the state-of-the-art timer, UI-Timer (the winner of the TAU 2014 CAD contest), which targets on one-time full timing update with CPPR [8]. We have presented UI-Timer in Chapter 1. OpenTimer inherits the merits of UI-Timer, in particular its efficient data structures for pessimism retrieval and path search, and enhances it to be capable of incremental processing. For pessimism retrieval, we have implemented the LUT-based method by UI-Timer. Several LUTs are first built through the clock tree. Based on these LUTs, the amount of pessimism can be quickly retrieved by referring to the lowest-common ancestor (LCA) between tree nodes.

The second idea we borrowed from UI-Timer is the implicit representation of path. UI-Timer proposed two complementary data structures, namely suffix tree and prefix tree, to represent the search space of the path ranking. The suffix tree represents the shortest path tree rooted at a referenced node. The prefix tree is a tree order of non-suffix-tree edges such that each tree node represents the path being deviated on the corresponding edge from its ordinary trace in the suffix tree. Each path can be implicitly stored by the two data structures and the memory usage and the search time can be significantly reduced to constant time per path during the search. In the following sections, we shall focus on the major contributions of OpenTimer, while algorithmic details of pessimism retrieval and path ranking can be referred to [8].

2.4.2 Topological Ordering and Incremental Levelization

In timing analysis, the circuit is interpreted as a set of pin-to-pin connections or a directed acyclic graph (DAG) $G = \{P, E\}$, where P is the pin set and E is the edge set. Because of this special property, every pin p in the circuit graph can be levelized by a level index "$level[p]$" such that the topological order among different pins is maintained. The timing can thus be propagated level by level without destroying the circuit topology. In fact, we observe three major advantages of the topological levelization:

- Incremental timing can be achieved via the insertion of frontier pins from which the timing propagation originates.
- Using the level indices, timing can be propagated in a pipeline fashion as dependencies can be scheduled into different levels.
- Multi-threading is highly scalable since the timing in a given level can be propagated simultaneously.

As a result, we construct a bucket list as the core data structure for timing propagation. Each bucket is associated with a level index l and has a list storing those pins with level indices equal to l. The bucket list also records the minimum and maximum level indices of non-empty pin lists. Starting from the pin list in the lowest level, the function of incremental levelization is presented in Algorithm 13. In a rough view, Algorithm 13 iteratively levelizes a pin from the lowest level to the highest level (line 2:16). Once the pin is levelized, all its fanout pins are inserted to the bucket list (line 8:13).

Algorithm 13: IncrementalLevelization(B)

```
   Input: bucket list $B$
   Output: level indices of pins
 1 $l \leftarrow B.min\_nonempty\_level$;
 2 while $l \leq B.max\_nonempty\_level$ do
 3     for $p \in B.pinlist(l)$ do
 4         for $p^- \in p.fanin\_pins$ do
 5             $level[p] \leftarrow \max(level[p^-] + 1,\ level[p])$;
 6         end
 7         $B.insert(p)$;
 8         for $p^+ \in p.fanout\_pins$ do
 9             if $level[p] + 1 > level[p^+]$ then
10                 $level[p^+] = level[p] + 1$;
11             end
12             $B.insert(p^+)$;
13         end
14     end
15     $l \leftarrow l + 1$;
16 end
```

Lemma 6: *Denoting the downstream pin set of a pin as D_p^+, for every pin p in the bucket list B, we have $\{p' \in B \mid p' \in D_p^+\}$ after Algorithm 13.*

2.4.3 Forward Timing Propagation

Using the levelized bucket list, we develop the procedure of forward timing propagation. The forward timing propagation performs six tasks, *RC propagation*, *slew propagation*, *delay propagation*, *arrival time propagation*, *jump point propagation*, and *CPPR credit propagation*, for every pin in the bucket list level by level. RC propagation updates the RC parameters that are required for slew and delay propagations through a net. Slew propagation propagates the slew from an input cell pin to the output cell pin through a cell or an output cell pin to multiple input cell pins through a net. Delay propagation computes the edge delay through cells and nets. Similar to slew propagation, arrival time propagation propagates the arrival time through delay values on cell edges or net edges. In jump point propagation, we contract the graph in order to reduce the search space. CPPR credit propagation computes the amount of pessimism to be removed for a timing test.

RC Propagation

We adopt the parasitic protocol by [9], where the output slew and delay through the RC network of a net are approximated by the symmetric of the value of the first and second moments of the impulse response. The approximation can be parameterized in a way such that the output slew and delay are functions of these RC parameters. Therefore, the goal of the RC propagation is to compute these RC parameters for any RC network. While the details are referred to [9], Algorithm 14 presents the procedure of RC propagation on the RC networks in a given level.

Algorithm 14: PropagateRC(l)

```
Input: level index $l$
$B \leftarrow$ bucket list of the timer;
for $p \in B.pinlist(l)$ do
    if $p.is\_rc\_network\_root$ = true then
        $n \leftarrow p.net$;
        if $n.is\_rc\_up\_to\_date$ = false then
            update RC parameters for net $n$;
        end
    end
end
```

Slew and Delay Propagation

The propagations of slew and delay are carried out by Algorithm 15 and Algorithm 16, respectively. For each pin from a given level, the slew and delay to this pin are propagated from its fanin through either the RC network using pre-computed RC parameters (line 7:8 in Algorithm 15 and line 4:5 in Algorithm 16) or the cell timing arc where the values are obtained via extrapolation or interpolation on the corresponding slew and delay LUTs (line 10:12 in Algorithm 15 and line 7:9 in Algorithm 16).

```
Algorithm 15: PropagateSlew($l$)
   Input: level index $l$
1  $B \leftarrow$ bucket list of the timer;
2  for $p \in B.pinlist(l)$ do
3      if $p.num\_fanins$ = NULL then
4          assign slew to $p$ from the primary input;
5      else
6          for $e \in p.fanin\_edges$ do
7              if $e.is\_net\_edge$ = true then
8                  propagate slew to $p$ through rc-timing on $e.net$;
9              else
10                 if $e.is\_constraint\_edge$ = false then
11                     propagate slew to $p$ through LUT on $e$;
12                 end
13             end
14         end
15     end
16 end
```

Arrival Time Propagation

The propagation of arrival time is trivial once the delay value on each edge is ready. It has been shown that finding the earliest and latest arrival time in the circuit graph is equivalent to finding the shortest and longest paths in a DAG, which can be fulfilled using levelized propagation [2]. Algorithm 17 presents such propagation at a given level.

Jump Point Propagation

Reducing the size of the timing graph is an effective way to speed up the path search. Because of intrinsic properties of cells, many paths are present in a tree form. To be more specific, for some pin pairs at certain transitions, the paths in between are uniquely defined. For instance, the AND gate in Figure 2.4 is unate-definite (i.e., either positive unate or negative unate), and hence any paths passing through are not diverged. Starting from pin

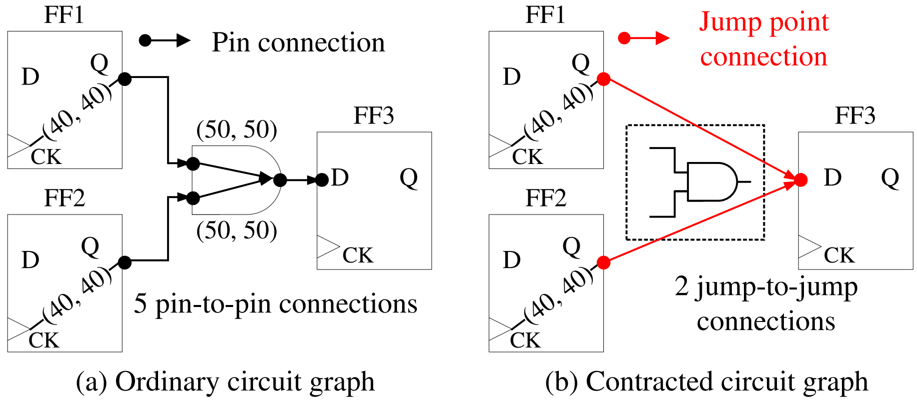
Algorithm 16: PropagateDelay(l)

```
Input: level index $l$
$B \leftarrow$ bucket list of the timer;
for $p \in B.pinlist(l)$ do
    for $e \in p.fanin\_edges$ do
        if $e.is\_net\_edge$ = true then
            update delay of $e$ through rc-timing on $e.net$;
        else
            if $e.is\_constraint\_edge$ = false then
                update delay of $e$ through LUT on $e$;
            end
        end
    end
end
```

Algorithm 17: PropagateArrivalTime(l)

```
Input: level index $l$
$B \leftarrow$ bucket list of the timer;
for $p \in B.pinlist(l)$ do
    if $p.num\_fanins = 0$ then
        assign arrival time to $p$ from the primary input;
    else
        for $e \in p.fanin\_edges$ do
            propagate arrival time to $p$ through delay on $e$;
        end
    end
end
```

FF3:D at any transition, there exists only one path back to pin FF1:Q or pin FF2:Q. Consequently, we can construct a shortcut that allows the path search to jump over the subcircuit from FF2:Q or FF1:Q to FF3:D. In this case, the pin FF3:D is named as "*jump head*" and pins FF2:Q and FF1:Q are named as "*jump tail*".



(a) Ordinary circuit graph

(b) Contracted circuit graph

Figure 2.4: Graph contraction using jump-point connections.

The examination of whether a pin is a jump head or a jump tail is presented in Algorithms 18–19. It can be analogized to a tree where the jump head is the root and the jump tail is the leave. As shown in Algorithm 18, a pin at any transition and timing split is referred to as a jump head only if its output signal is not branched. On the other hand, the jump tail is determined by whether its input signal is uniquely defined. Using Algorithms 18–19, the construction and propagation of jump points are given by Algorithms 20–21. In a rough view, Algorithm 20 induces the jump point connection through a recursive traversal to discover any tree-structured subcircuit. Algorithm 21 applies Algorithm 20 to each pin in a give level. Notice that jump point connections are only considered among data network.

```
Algorithm 18: is_jump_head($p$)
  Input: an existing pin $p$
  if $p.num\_fanouts = 0$ or $p.num\_fanouts > 1$ or $p.is\_in\_clock\_tree$
    then
      return true;
  end
  $e \leftarrow p.fanout\_edges$;
  return $e.timing\_sense$ = non_unate;
```

Algorithm 19: is_jump_tail(p)

```
Input: an existing pin $p$
if $p.num\_fanins = 0$ then
    return true;
end
for $e \in p.fanin\_edges$ do
    if $e.is\_constraint\_edge =$ false then
        $head \leftarrow$ is_jump_head($e.from\_pin$);
        if $head =$ true then
            return true;
        end
    end
end
return false;
```

Algorithm 20: induce_jump_point(p, p', d)

```
Input: two pins $p$ and $p'$ and a delay value $d$
$p.jump\_head \leftarrow p'$;
if $is\_jump\_tail(p) =$ true then
    if $p \neq p'$ then
        insert a jump connection from $p$ to $p'$ with delay $d$;
    end
    return;
end
for $e \in p.fanin\_edges$ do
    $p^- \leftarrow e.from\_pin$;
    if $e.is\_constraint\_edge =$ true or $p^-.is\_in\_clock\_tree =$ true then
        continue;
    end
    induce_jump_point($p^-$, $p'$, $d + e.delay$);
end
```

Algorithm 21: PropagateJumpPoint(l)

```
Input: level index $l$
$B \leftarrow$ bucket list of the timer;
for $p \in B.pinlist(l)$ do
    if $p.is\_in\_clock\_tree =$ true or $is\_jump\_head(p) =$ false then
        return;
    end
    induce_jump_point($p$, $p$, 0);
end
```

```
Algorithm 22: PropagateCPPRCredit($l$)
   Input: level index $l$
 1 $B \leftarrow$ bucket list of the timer;
 2 for $p \in B.pinlist(l)$ do
 3     $t \leftarrow p.timing\_test$;
 4     if $t$ = NULL or $t.is\_sequential\_test$ = false then
 5         continue;
 6     end
 7     # Fork_Thread_Task {
 8         $path \leftarrow$ GetCriticalPath($t$, 1) [8];
 9         $t.cppr\_credit \leftarrow path.cppr\_credit$;
10     };
11 end
```

CPPR Credit Propagation

For each data pin of an FF that is guarded by setup tests or hold timing tests, we need to discover the corresponding CPPR credit for slack adjustments [1]. The CPPR credit is defined as the numeric that is applied to skew the worst post-CPPR slack of a particular test [9]. As aforementioned, the state-of-the-art path tracing algorithm by UI-Timer [8] is our default engine for the investigation of CPPR credits for any timing tests. The algorithm of CPPR credit propagation is presented in Algorithm 22. In contrast to UI-Timer where the search graph is induced from the flattened circuit graph, we are able to reduce the search space with jump points which can lead to significant speedup. Because of the independence of timing tests, the path tracing can be performed in a parallel manner (line 7:10).

2.4.4 Backward Timing Propagation

In contrast to forward timing propagation, the backward timing propagation propagates the timing for every pin in the bucket list from the highest level to the lowest level by performing two major tasks, *fanin propagation* and *required arrival time propagation*. Fanin propagation inserts the fanin of each pin from the bucket list in order to construct the upstream cone. Required arrival time propagation propagates the timing constraint in a backward manner.

Fanin Propagation

In order to perform backward timing propagation, we need to construct the upstream cone of every pin in the bucket list. Considering the procedure in Algorithm 23 which inserts all fanin pins from a pin list in a given level, the upstream cone for backward timing propagation can be constructed by calling this procedure level by level.

Algorithm 23: PropagateFanin(l)

```
Input: level index $l$
$B \leftarrow$ bucket list of the timer;
for $p \in B.pinlist(l)$ do
    for $p^{-} \in p.fanin\_pins$ do
        $B.insert(p^{-})$;
    end
end
```

Required Arrival Time Propagation

The propagation of required arrival time in a given level is shown in Algorithm 24. Algorithm 24 exerts similar procedure as Algorithm 17 but in a reversed direction (line 8:10). For constrained pin, the required arrival time is assigned by the constraint value from the corresponding timing test (line 4:5) and is adjusted by the CPPR credit in case of sequential timing tests (line 6).

2.4.5 Design Modification

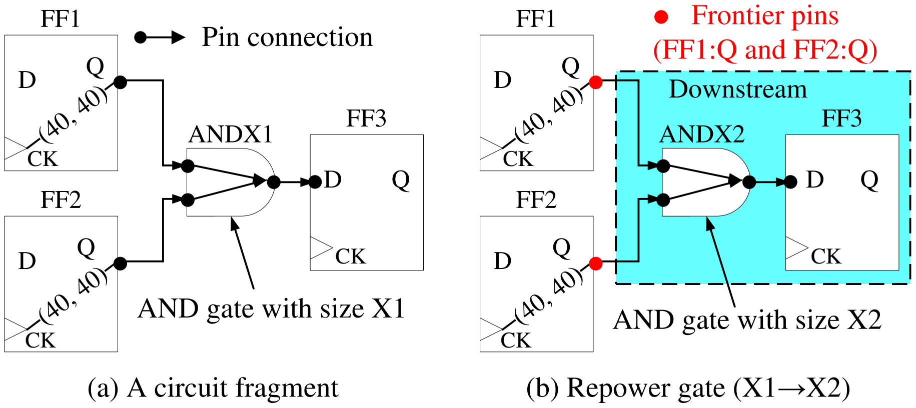
Based on the levelized bucket list, the objective of dealing with design modifiers is to identify the set of "*frontier pins*" from which the incremental timing update originates. Starting at the frontier pins, Algorithm 13 constructs a downstream cone of the affected area which will be used for incremental timing update. We consider the design modifiers at gate level, net level, and pin level.

Algorithm 24: PropagateRequiredArrivalTime(l)

```
Input: level index $l$
$B \leftarrow$ bucket list of the timer;
for $p \in B.pinlist(l)$ do
    if $p.num\_fanouts = 0$ then
        $t \leftarrow p.timing\_test$;
        assign required arrival time to $p$ from $t$;
        adjust required arrival time with CPPR credit from $t$;
    else
        for $e \in p.fanout\_edges$ do
            propagate required arrival time to $p$ through delay on $e$;
        end
    end
end
```

Gate-Level Modifications

The operations that modify the design at gate level are (1) insert_gate, (2) remove_gate, and (3) repower_gate. Recall that the operation insert_gate creates a new gate in the design and the operation remove_gate removes a disconnected gate from the design. It is obvious that the two operations introduce no frontier pins as the gate being inserted or removed is not connected to the current circuit. Therefore, for gate-level design modifiers we only deal with the operation repower_gate.



(a) A circuit fragment

(b) Repower gate (X1→X2)

Figure 2.5: A design modification by repowering the gate with another size (repower_gate).

An example of the operation repower_gate is shown in Figure 2.5. The AND gate in the data network is repowered from size X1 (cell ANDX1) to size X2 (cell ANDX2). Repowering a gate changes the cell timing and the pin capacitance. The affected area should be traced back by one level where the pins connecting the gate originate the incremental timing. In this example, the incremental timing propagation is captured by two frontier pins FF1:Q and FF2:Q. Using this fact, our solution to the operation repower_gate is presented in Algorithm 25. Algorithm 25 first replaces the cell that was attached to the gate with the new cell (line 1). Afterward the frontier pins, which are fanin pins of each input pin of the gate, are inserted into the bucket list (line 3:9) for incremental timing update.

Algorithm 25: repower_gate(g, c)

```
  Input: an existing gate $g$, a new cell $c$
1 remap the gate $g$ to the new cell $c$;
2 $B \leftarrow$ bucket list of the timer;
3 for $p \in g.input\_pins$ do
4     for $p^- \in p.fanin\_pins$ do
5         $B.insert(p^-)$;
6         $n \leftarrow p^-.net$;
7         $n.is\_rc\_up\_to\_date \leftarrow$ false;
8     end
9 end
```

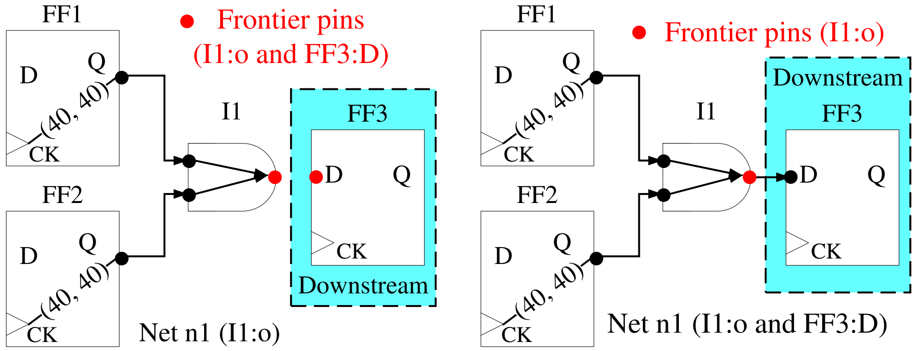
Net-Level Modifications

There are three operations that modify the design at net level: (1) insert_net, (2) remove_net, and (3) read_spef. Similar to gate-level modifications, the operation insert_net creates an empty (disconnected) net for the design and the operation remove_net deletes an empty net from the design. Due to the isolation, both operations have no impact on current timing profile. The net-level design modifier read_spef is the only operation that could affect the timing. Our solution to read_spef is presented in Algorithm 26. Algorithm 26 first parses the given .spef file into an object (line 1). Then it iterates each net that was parsed from the .spef file and asserts the new parasitics to it (line 3:4). Whenever the parasitics of a net change, the incremental timing update is captured by the root of the corresponding RC network (line 5:7).

```
Algorithm 26: read_spef(.spef)
Input: a .spef file
$O \leftarrow$ parse .spef file into an object;
$B \leftarrow$ bucket list of the timer;
for net $n \in O$ do
    update the parasitics of net $n$ through $O$;
    $n.is\_rc\_up\_to\_date \leftarrow$ false;
    $p_r \leftarrow n.rc\_network\_root\_pin$;
    $B.insert(p_r)$;
end
```

Pin-Level Modifications

The pin-level design modifiers are the most crucial operations since they directly alter the connectivity in the design. There are two operations that modify the design at pin level: (1) disconnect_pin and (2) connect_pin. The operation disconnect_pin disconnects the pin from the net it is connected to and the operation connect_pin connects the pin to a given net. Both operations alter the structure of the design and directly affect the timing. Consequently, we need to identify the frontier pins that capture the incremental timing update for such changes.



(a) Disconnect pin FF3:D from net n1 (b) Connect pin FF3:D to net n1

Figure 2.6: A design modification by disconnecting/connecting a pin from/to a net (disconnect_pin/connect_pin).

An example for operations disconnect_pin and connect_pin are given in Figure 2.6. It can be seen from (a) disconnecting the pin I1:o from its net cuts off the connection from I1:o to FF3:D. This change affects the timing at the pins I1:o and FF3:D as well as the downstream cone of the pin FF3:D.

Therefore, disconnecting a pin introduces two frontier pins that are the two end points at the connection to or from which the pin is connected. On the other hand, connecting a pin to a given net establishes a new connection. In (b), connecting the pin I1:o to the net n1 produces a new connection from the pin I1:o to the pin FF3:D. This change has impact on the timing profile in the downstream cone of pin I1:o. As a result, connecting a pin introduces one frontier pin which is the tail of this connection. Algorithms 27–28 present our solutions to pin-level operations. Notice that a pin is considered either a root of the RC network where we need to remove or insert all possible connections, including the jump point connection that covers such a change (line 4:8 in Algorithm 27 and line 2:7 in Algorithm 28), or the terminal of the RC network in which case we deal with the only one connection (line 10 in Algorithm 27 and line 9:11 in Algorithm 28).

Algorithm 27: disconnect_pin(p)

```
   Input: an existing pin $p$
1  $n \leftarrow p.net$ ;
2  $p_r \leftarrow n.rc\_network\_root\_pin$ ;
3  $B \leftarrow$ bucket list of the timer;
4  if $p = p_r$ then
5      for $p' \in n.pinlist - \{p_r\}$ do
6          $B.insert(p')$;
7          disconnect pin $p'$ from the net $n$;
8      end
9  else
10     $B.insert(p_r)$;
11 end
12 $B.insert(p)$;
13 disconnect all jump point connections to $p.jump\_head$;
14 disconnect the pin $p$ from the net $n$;
```

2.4.6 Incremental Timing Update

Based on Algorithms 13–28, we are able to deliver the key procedure for incremental timing update. In order to guarantee correct timing results, the task dependency among different timing propagations needs to be carefully addressed. For backward timing propagation in a given level, the procedures

Algorithm 28: connect_pin(p, n)

```
Input: an existing pin $p$ and an existing net $n$
$B \leftarrow$ bucket list of the timer;
if $p.is\_rc\_network\_root\_pin$ = true then
    for $p' \in n.pinlist$ do
        establish the connection from $p$ to $p'$;
        disconnect all jump point connections to $p'.jump\_head$;
    end
    $B.insert(p)$;
else
    $p_r \leftarrow n.rc\_network\_root\_pin$ ;
    establish the connection from $p_r$ to $p$;
    $B.insert(p_r)$;
end
disconnect all jump point connections to $p.jump\_head$;
connect the pin $p$ to the net $n$;
```

of fanin propagation and required arrival time propagation are apparently independent to each other. However, for forward timing propagation in a given level, the following dependency should be satisfied: (1) RC propagation (RCP) precedes the slew propagation (SLP) and delay propagation (DLP); (2) DLP precedes the arrival time propagation (ATP); (3) ATP precedes the jump point propagation (JMP); (4) JMP precedes the CPPR credit propagation (CRP). As the timing propagation is conducted level by level, the task dependency can be efficiently encapsulated by a parallel pipeline. Figure 2.7 illustrates this concept (subscript delineates the level index).

Algorithm 29 presents our solution to incremental timing update. It first calls Algorithm 13 to construct the downstream cone of all frontier pins in the bucket list (line 5). The timing propagation is then performed level by level in a parallel pipeline fashion (line 8:18 for forward timing propagation and line 19:25 for backward timing propagation). By the end of each pipeline stage, a barrier is imposed to synchronize all forked threads (line 16 and line 23). The bucket list is reset after the timing propagation is accomplished (line 26).

Algorithm 29: update_timing()

```
$B \leftarrow$ bucket list of the timer;
if $B.num\_pins = 0$ then
    return;
end
IncrementalLevelization($B$);
$l_{min} \leftarrow B.min\_nonempty\_level$;
$l_{max} \leftarrow B.max\_nonempty\_level$;
# Parallel_Region {
# Master_Thread_do for $l = l_{min}$ to $l_{max} + 4$ do
    # Fork_Thread_Task PropagateRC($l$);
    # Fork_Thread_Task PropagateSlew($l - 1$);
    # Fork_Thread_Task PropagateDelay($l - 1$);
    # Fork_Thread_Task PropagateArrivalTime($l - 2$);
    # Fork_Thread_Task PropagateJumpPoint($l - 3$);
    # Fork_Thread_Task PropagateCPPRCredit($l - 4$);
    # Synchronize_Thread_Tasks;
end
};
# Parallel Region {
# Master_Thread_do for $l = l_{max}$ to $B.min\_non\_empty\_level$ do
    # Fork_Thread_Task PropagateFanin($l$);
    # Fork_Thread_Task PropagateRequiredArrivalTime($l$);
    # Synchronize_Thread_Tasks;
end
};
remove all pins from the bucket list $B$;
```

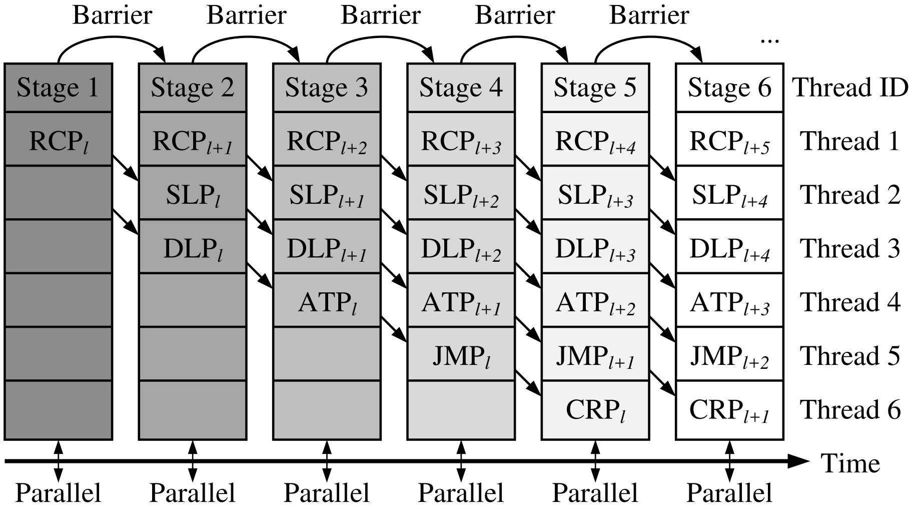


Figure 2.7: Parallel forward timing propagation using pipeline.

2.4.7 Timing Query

Using Algorithm 29 as the infrastructure, the value-based timing queries, for example, reporting the arrival time, can be implemented as Algorithm 30. Queries for required arrival time and slack can be mimicked in a similar manner. The path-based query is presented in Algorithm 31. Algorithm 31 takes two arguments, one pin p and a path count K, and reports the top K post-CPPR critical paths through p. If p is nil, the paths are searched across the entire circuit graph G (line 2). Otherwise, the search graph is limited to the region of downstream cone D_p^+ and upstream cone D_p^- of p such that every path discovered in the search graph passes through p (line 3:5). Then we apply the path ranking algorithm by [8] to peel out the top K critical tests (line 6). Finally we iteratively extract the top K critical paths from each of the top K critical tests and maintain the globally top K critical paths using a priority queue (line 7:15).

Algorithm 30: report_at(p, s, m)

```
Input: an existing pin $p$ and targeted transition $s$ and timing split $m$
Output: arrival time at $p$ for $s$ and $m$
1 update_timing();
2 return $p.arrival\_time(s, m)$;
```

```
Algorithm 31: report_worst_path($p$, $K$)
  Input: an existing pin $p$ and a path count $K$
  Output: top $K$ critical paths through $p$ in the design
update_timing();
$G' \leftarrow G$;
if $p \neq$ NULL then
    $G' \leftarrow D_p^- \cup D_p^+$;
end
extract a sorted set $T$ of the top $K$ post-CPPR critical tests from
  $G'$ [8];
$Q \leftarrow$ priority queue keyed on post-CPPR slack values;
for $t \in T$ do
    if $Q.size = K$ and $t.slack \geq Q.top\_max$ then
        break;
    end
    $Q \leftarrow Q \cup$ GetCriticalPath($t$, $K$) [8];
    $Q.maintain\_top\_k\_min(K)$;
end
return $Q$;
```

2.5 Experimental Results

OpenTimer is implemented in C++ language on a 2.20 GHz 64-bit Linux machine with 128 GB memory. The application programming interface (API) provided by OpenMP 3.1 is used for our multi-threaded programming. Our machine can execute a maximum of eight threads concurrently. Experiments are undertaken on a set of industry benchmarks released from the TAU 2015 CAD contest [9]. The golden reference is generated from an industry timer and the design modifiers are wrapped in a .ops file which contains tens of millions of operations. Table 2.1 lists the benchmark statistics and the performance of OpenTimer compared to the top performers, “iTimerC 2.0” and “iitRACE,” from the TAU 2015 CAD contest [9].

We begin by comparing OpenTimer with iitRACE. The strength of OpenTimer is clearly demonstrated in the accuracy and runtime values. We have seen a significant performance gap where our timer is much more accurate and far faster than iitRACE. Even though iitRACE achieves better memory usage, such data are less meaningful when accuracy is considered the top priority. Next we compare OpenTimer with iTimerC 2.0. In general, Open-

Table 2.1: Performance Comparison Between OpenTimer and Top-ranked Timers iitRace and iTimerC 2.0 from the TAU 2015 CAD Contest [9]

Circuit	#Gates	#Nets	#OPs	iitRACE			iTimerC 2.0			OpenTimer		
				accuracy	runtime	memory	accuracy	runtime	memory	accuracy	runtime	memory
b19	255.3K	255.3K	5641.5K	63.03 %	629 s	3.0 GB	99.95 %	215 s	5.8 GB	99.95 %	52 s	4.6 GB
cordic	45.4K	45.4K	1607.6K	61.83 %	100 s	0.9 GB	98.88 %	80 s	1.3 GB	98.88 %	18 s	1.3 GB
des_perf	138.9K	139.1K	4326.7K	67.43 %	299 s	4.2 GB	97.02 %	92 s	3.1 GB	99.73 %	30 s	3.0 GB
edit_dist	147.6K	150.2K	3368.3K	64.83 %	857 s	2.0 GB	98.29 %	98 s	3.8 GB	98.30 %	42 s	3.8 GB
fft	38.2K	39.2K	1751.7K	89.66 %	70 s	0.5 GB	98.45 %	49 s	1.2 GB	99.77 %	11 s	1.2 GB
leon2	1616.4K	1517.0K	8438.5K	72.34 %	16832 s	9.9 GB	100.00 %	787 s	27.2 GB	100.00 %	282 s	22.8 GB
leon3mp	1247.7K	1248.0K	8405.9K	62.99 %	4960 s	8.2 GB	100.00 %	609 s	19.8 GB	100.00 %	163 s	17.9 GB
mgc_edit_dist	161.7K	164.2K	3403.4K	64.29 %	1578 s	1.9 GB	100.00 %	135 s	4.1 GB	100.00 %	41 s	3.1 GB
mgc_matrix_mult	171.3K	174.5K	3717.5K	67.93 %	1363 s	2.0 GB	100.00 %	157 s	4.3 GB	100.00 %	31 s	3.1 GB
netcard	1496.0K	1497.8K	11594.6K	87.63 %	6662 s	9.4 GB	99.99 %	691 s	22.9 GB	99.99 %	192 s	20.8 GB
cordic_core	3.6K	3.6K	226.0K	59.42 %	21 s	0.3 GB	95.19 %	29 s	0.2 GB	95.19 %	3 s	0.1 GB
crc32d16N	478	495	28.9K	57.15 %	3 s	0.1 GB	100.00 %	5 s	0.1 GB	100.00 %	1 s	0.1 GB
softusb_navre	6.9K	7.0K	427.8K	40.17 %	21 s	0.1 GB	0.00 %	-	-	99.97 %	4 s	0.5 GB
tip_master	37.7K	38.5K	1300.4K	82.95 %	64 s	0.6 GB	96.42 %	47 s	1.0 GB	97.04 %	9 s	0.8 GB
vga_lcd_1	139.5K	139.6K	2961.5K	99.65 %	260 s	1.6 GB	100.00 %	94 s	2.2 GB	100.00 %	31 s	2.9 GB
vga_lcd_2	259.1K	259.1K	12674.7K	98.57 %	1132 s	13.3 GB	100.00 %	156 s	5.0 GB	100.00 %	65 s	3.9 GB

#Gates: number of gates. #Nets: number of nets. #OPs: number of operations. accuracy: average of path accuracy and value accuracy (%). -: program crash.

Timer outperforms iTimerC 2.0 across nearly all circuit benchmarks in any aspects. We reach the goal by ×2.3 (edit_dist) to ×9.7 (cordic_core) faster and consume less memory for most benchmarks. In addition, our accuracy is higher than iTimerC 2.0 by 7% on average. Unfortunately, we are unable to compare the data on the benchmark softusb_navre because iTimerC 2.0 encountered execution fault.

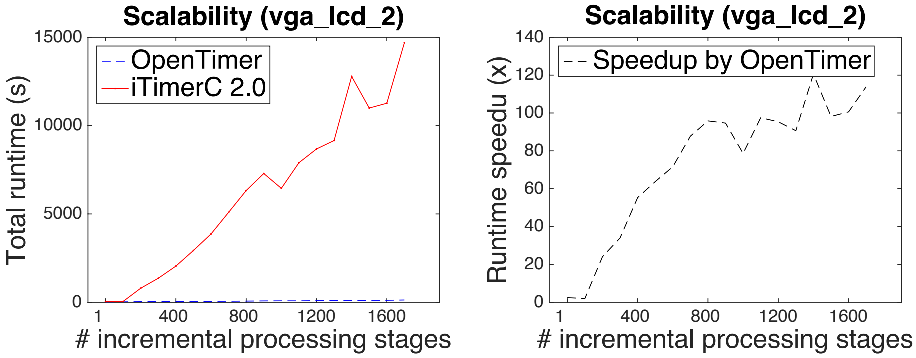


Figure 2.8: Scalability comparison between OpenTimer and iTimerC 2.0.

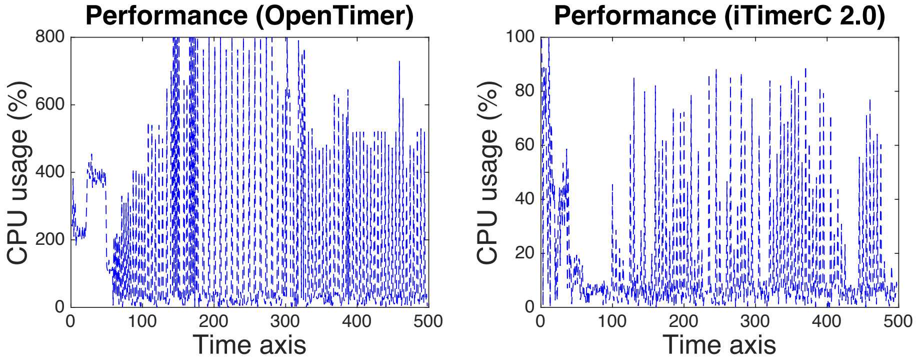


Figure 2.9: Parallelism comparison between OpenTimer and iTimerC 2.0.

Finally we investigate the scalability of our timer and iTimerC 2.0 on accommodating the depth of incremental processing. We omit the comparison with iitRACE because its low accuracy might result in unfairness. In this experiment, we refer to a set of design modifiers followed by at least one timing query as "one stage" of incremental processing. We have divulged, unfortunately, all benchmarks from the TAU 2015 contest have less than 10 incremental processing stages, which is not sufficient to reveal the performance bottleneck. Therefore, we modified the benchmark vga_lcd_2 by inserting a path-based timing query after each complete design modification. The comparison of runtime scalability between OpenTimer and iTimerC 2.0

is demonstrated in Figure 2.8. It can be clearly seen that our runtime scales extremely well as the number of incremental processing stages increases. For instance, OpenTimer accomplished the goal by ×95.8 faster (66 seconds vs. 6324 seconds) than iTimerC 2.0 at the 800th stage. Similar trends can be observed on other stage numbers. We further reveals the cpu usage for both programs in Figure 2.9. It is observed OpenTimer is highly parallel, using up to the hardware-limited thread number, while iTimerC 2.0 does not support any multi-threaded feature. To sum up, these experiments have justified the software quality of OpenTimer.

2.6 Conclusion

In this chapter we have presented OpenTimer, a high-quality incremental timing analysis algorithm with CPPR. We have not only captured the key features that achieve incremental capability, but also parallelized the incremental timing update in a pipeline fashion. Our framework is very flexible and scalable as many critical tasks such as timing propagation and CPPR are scheduled into the pipeline so as to overlap their runtimes. These advantages confer OpenTimer a high degree of differential over existing methods. Comparatively, experimental results have demonstrated the superior performance of OpenTimer in terms of accuracy, runtime, and memory over the top performers from the TAU 2015 CAD contest.

CHAPTER 3

ACCELERATED PATH-BASED TIMING ANALYSIS WITH MAPREDUCE

3.1 Introduction

Static timing analysis (STA) is a crucial step in verifying the expected timing behaviors of an integrated circuit [2]. During the STA, both graph-based timing analysis (GBA) and path-based timing analysis (PBA) are used. GBA performs a linear scan on the circuit graph and estimates the worst timing quantities at each end point. GBA is very fast but the results are pessimistic. Hence, PBA is often performed after GBA to remove unwanted pessimism. Starting from a negative end point, a core PBA procedure peels a set of paths in non-increasing order of criticality and applies path-specific timing update to each of these paths [26]. However, path peeling is a computationally expensive process. The high runtime demand severely restrains the capability of PBA during timing signoff.

Unfortunately, current literature still lacks for novel ideas of fast PBA [27]. As pointed out by the 2014 TAU timing analysis contest, algorithms featuring multi-threaded or massively-parallel accelerations are eagerly in demand [1]. Howbeit, parallel PBA has been reported as a tough challenge primarily because a path can be prototypically various. For instance, a path can exhibit arbitrary lengths and span different logical cones and physical boundaries. Computations in this way are typically hard to be issued in parallel. Although a few prior works claimed to have a solution, the results are usually compromised with accuracy [5, 28].

As a consequence, we introduce in this chapter an ultra-fast PBA framework with MapReduce. The concept of MapReduce is shown in Figure 3.1. A MapReduce program applies parallel map operations to input tasks and generates a set of temporary key/value pairs. Then parallel reduce operations are applied to all values that are associated with the same key in

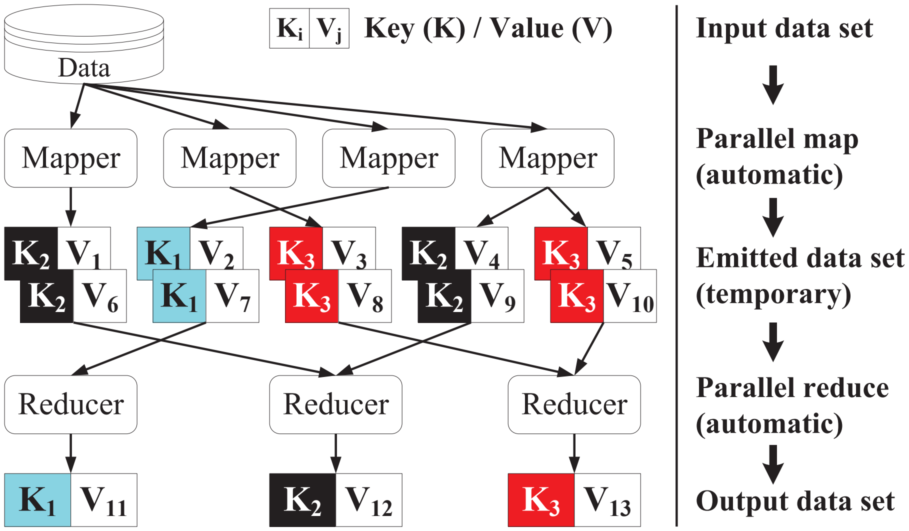


Figure 3.1: The execution flow of a MapReduce job.

order to collate the derived data properly [29]. Users only need to provide desired map/reduce functions while parallelization details are encapsulated in a MapReduce library [30, 31]. This programming paradigm inspires us to rethink the PBA problem as "map" operations followed by "reduces". Specifically, we cast the PBA problem into tasks with keys and values that are sandwiched around massively-parallel map and reduce operations.

Our contributions are summarized as follows. (1) We successfully investigated the applicability of MapReduce to accelerate PBA. Our framework is very general in gaining massively parallel computations, imposing no physical and logic constraints. (2) Our framework increases the productivity as designers can focus on timing-oriented turnaround, leaving all hassle of parallelization details to the MapReduce library. (3) We have seen a substantial speedup from the experimental results. On a large distributed system, millions of cells can be easily analyzed in a few minutes. These features all add up to faster design cycle. Our work can be beneficial for the speedup of the signoff timing closure, on which up to 40% of the design flow is typically spent.

3.2 Path-Based Timing Analysis

PBA has gained much attention in deep submicron era due to its capability of configuring features such as clock-reconvergence-pessimism removal (CRPR) and advanced-on-chip-variation (AOCV) derating for less-pessimistic timing reports [1, 8]. Since most of these features are path-specific, a core yet computationally expensive building block of PBA is to peel a path set from each end point and recompute the timings path-by-path. By analyzing the path with reduced pessimism, many timing violations can be waived which in turn tells better timing signoff. Because of this crucial benefit, studies in accelerating PBA are in demand especially when we move to the many-core era. Simply put, the following aspects are in particular of interests:

- Performance is the top concern. A substantial runtime saving will make a breakthrough in timing signoff.
- Modern circuits are complex. Practical parallelization must scale up with the growth of the circuit size.
- The framework needs to be general and flexible, imposing the least constraints and complexity.
- Adequate granularity control is necessary in order to effectively organize computations at a massive scale.
- Orthogonality should be featured. Compromised solutions to the design methodology are discouraged.

The above issues all combine to challenges in the development of parallel PBA algorithms. If the PBA runtime can be significantly improved, designers are able to utilize PBA on a larger set of paths and perform their analyses earlier in the design closure flow. As a result, researchers must continue to provide viable parallel solutions along with the rapid evolution of the computational power.

3.3 Problem Formulation

The circuit network is input as a directed-acyclic graph $G = \{V, E\}$, where V is the pin set of circuit elements and E is the edge set specifying pin-to-pin connections. Each edge e is associated with a tuple of earliest and latest delays. A path is an ordered sequence of nodes or edges and the path delay is the sum of delays through all edges. In this chapter, we are in particular emphasizing on the *data path*, which is defined as a path from either the primary input pin or the clock pin of a launching flip-flop (FF) to the data pin of a capturing FF. A test is defined w.r.t. an FF as hold or setup check on any data paths captured by this FF. Considering a test set T as well as a positive integer k, the following two tasks are essential for PBA [26, 1].

Task 1 – Sweep report: *The program is asked to sweep all tests and output the top k critical paths for each test.*

Task 2 – Block report: *The program is asked to report the top k critical paths across all tests.*

3.4 MapReduce Framework

In this section we discuss our PBA framework with MapReduce. We first brief the MapReduce programming paradigm and then detail each step of our framework.

3.4.1 MapReduce Programming Paradigm

Since being first introduced by Google in 2004, the MapReduce programming paradigm has been widely applied to many domains such as data mining, database system, and high-performance computing [29]. The spirit of a MapReduce program lies in "*keys*" and "*values*" which are generated and manipulated by user-defined functions "*mapper*" and "*reducer*". A key and a value are simply bytes of strings of arbitrary length which are logically associated with each other and thus can represent generic data types. The MapReduce library automatically schedules parallel map and reduce operations linking mapper and reducer to handle the input data on a distributed system. State-of-the-art libraries for this purpose such as Apache Hadoop and MR-MPI from Sandia National Lab. are readily available [30, 31].

A canonical MapReduce program is presented in Algorithm 32. The first is the map step, which takes a set of data and converts it into another set of

```
Algorithm 32: CanonicalForm($D$, mapper, reducer)
  Input: input data $D$, user-defined mapper and reducer
{$M$ | <tmp_key : tmp_value>} ← Map($D$, mapper) ;
{$C$ | <unique_key : value_list>} ← Collate($M$);
{$R$ | <key : value>} ← Reduce($C$, reducer);
return $R$
```

data produced by the function mapper, where individual elements are represented as temporary key/value pairs. The collate step aggregates across temporary key/value pairs where each unique key appears exactly once and the corresponding value is a concatenated list of all the values associated with the same key [1]. The reduce step then takes a single entry from the aggregated key/value pairs and creates a new key/value pair which stores the output generated by the function reducer. Parallelism is evident since function calls by map and reduce are independent to each other and can be executed on different processors simultaneously. In general, map and reduce are intra-process operations while collate involves inter-process communication because of aggregation.

3.4.2 Formulation of Task Graphs

In order to develop a MapReduce program, computations that can be issued to parallel map and reduce operations must be exploited from our problem. Considering a test t, we observe: (1) every data path captured by this testing FF reaches the same end point; (2) the *source pins* from where a path originates is prototypically consistent, being either the primary input pins or the clock pin of a launching FF. The first feature implies that paths feeding the same end point belong to the same test. By tagging each path with a key indicating the corresponding test index, the program can keep track of the test to which a path belongs. The second feature implies that paths are wrapped in a multi-source single-target graph. This motivates us to decompose a test into several task graphs with regard to different and smaller groups of source pins.

We define g_t for each test t as a set of task graphs $g_t = \{g_t^1, g_t^2, ..., g_t^i\}$ and

[1] In some articles the collate is absorbed into the reduce step.

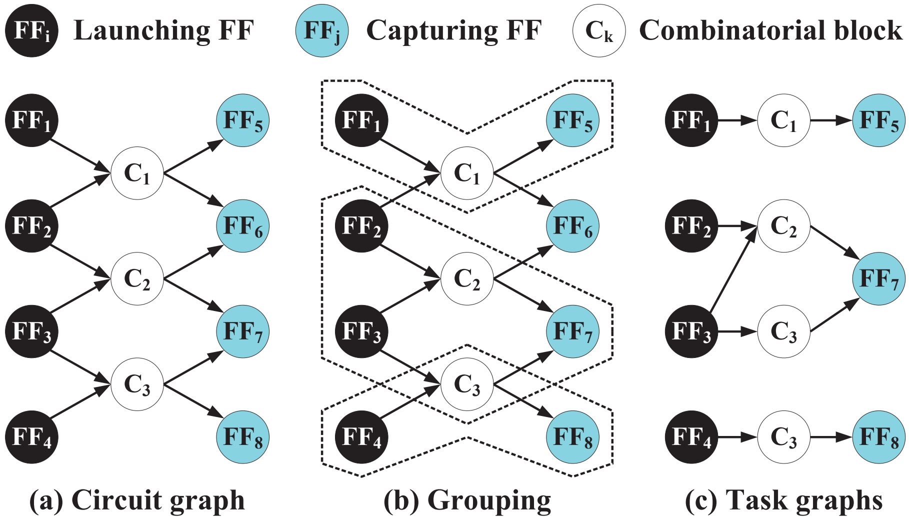


Figure 3.2: An example formulation of the task graphs.

G_T as a union set of all task graphs. Deriving from a test t, a task graph g_t^i is a subgraph spanning all connectivities from a subset of source pins to the data pin of this test. Under the same test, the source pins corresponding to different task graphs are mutually disjointed. We associate each task graph with a key indicating the test index to which this task graph belongs. An example is illustrated in Figure 3.2. We can see three task graphs are derived from the tests on capturing FFs 5, 7, and 8, respectively. Notice that a task graph is indeed a portion of the original circuit graph. Every edge of the task graph comes with the same delay values as the original circuit graph.

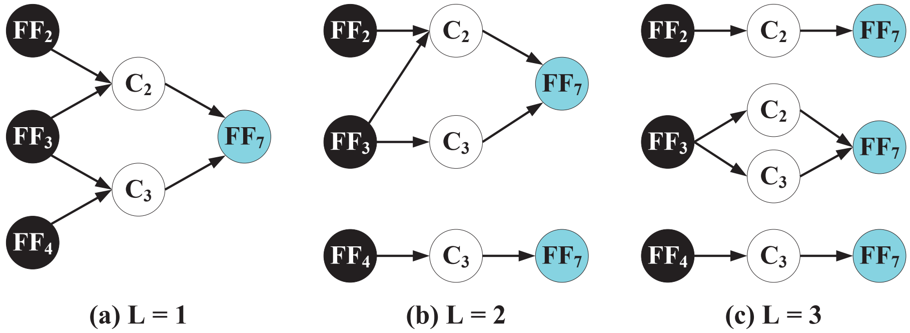


Figure 3.3: Granularity control of the task graph.

The granularity control of the task graphs is an important factor as it arises performance concern such as process communication and computation load. We define L-way partition as a partition of each test into L task graphs such that each task graph has roughly even size on the corresponding set of source pins. Figure 3.3 shows an example of one-way, two-way, and three-way partitions of the test on FF 7 from Figure 3.2. While discovering a suitable granularity level tends to be case-dependent, we consider in this chapter only the case where the number of tests is less than the number of available computing cores. Assuming P cores are available in such the case, up to P task graphs are generated from each test in order to balance the computation load. We should be mindful that dividing a test into multiple task graphs facilitates the parallelism but also gives rise to process communication because of data merging afterward.

The generation of task graphs is presented in Algorithm 33. We first identify all source pins of a given test t through a backtrace starting at the data pin d of this test (line 1:2). The number of task graphs being generated is determined by a comparison between the number of input tests and the number of available computing cores (line 3:8). Then we iteratively group a set of source pins S_d^k in accordance to the specified number of task graphs and perform a depth-first search to induce the corresponding task graph (line 9:15). Each induced task graph is assigned a key indicating its test index and is emitted as a key/value object in the end of each iteration (line 14).

Algorithm 33: Generator(t)

```
   Input: a test $t$
   Output: a set of task graphs $g_t = \{g_t^1, g_t^2, ..., g_t^i\}$
1  $d \leftarrow$ data pin of the test $t$;
2  $S_d \leftarrow$ source pins obtained through a back traversal at $d$;
3  $P \leftarrow$ number of available computing cores;
4  $L \leftarrow 1$;
5  if $|T| < P$ then
6      $L = P$;
7  end
8  $num\_src \leftarrow \lceil |S_d|/L \rceil$;
9  for $i \leftarrow 1$ to $L$ do
10     $S_d^i \leftarrow \{num\_src$ frontmost elements in $S_d\}$;
11     $S_d \leftarrow S_d \setminus S_d^i$;
12     $g_t^i \leftarrow$ subgraph induced from $S_d^i$ to $d$;
13     $key[g_t^i] \leftarrow t$;
14     Emit make_pair($t$, $g_t^i$);
15 end
```

Based on the knowledge constructed so far, we deliver a high-level sketch of our MapReduce-based PBA framework. The map operation is responsible for (1) the generation of task graphs from each test and (2) the path extraction from each task graph. Because of the granularity control, a test might be broken into several task graphs that are distributed to different processors during the map operations. The collate method is required in order to reorganize paths to their right places. Eventually, the reduce operation peels out a desired path set and emits it as the final solution. We conclude this section by the following lemma.

Lemma *Every path exactly and uniquely exists in one task graph.*

Proof The exactness is true because each task graph is an induced connectivity from a set of source pins to the data pin of a test. Since under a same test different sets of source pins of task graphs are mutually disjointed, the existence of every path is uniquely defined.

3.4.3 Mapper and Reducer Functions

Based on the definition of task graphs, we develop the function calls for map and reduce operations. As presented in Algorithm 34, our mapper function takes an arbitrary task graph and extracts the top-k critical paths (line 1). We leave this extraction process as a black box for user preferences. In this chapter, the optimal path ranking algorithm by [8] is used as our default engine. Then it iterates through each path and performs path-specific update according to user-configured features such as CRPR and AOCV (line 3). Each iteration ends with an emission of a key/value pair where the value is a path string and the key is being either 1) the key of the input task graph if sweep is the task objective (line 5:6) or 2) a nominal number instead (line 7:9).

Any key/value pair emitted by our mapper is in fact a solution fragment, where the key indicates the test index to which the value of a path string belongs. It can be inferred that after calling the collate method, there are two possible outcomes: either paths that belongs to the same task graph are aggregated together or all paths are put in a single group, depending on the task objective. Eventually, our reducer takes each unique key/value pair and peels out the final top-k critical paths from the path set stored in each value list. This implementation is given in Algorithm 35.

Algorithm 34: Extracter(g_t^i)

```
Input: an arbitrary task graph $g_t^i$
Output: an emitted set of key/value pairs
$P \leftarrow$ top $k$ critical paths extracted from $g_t^i$;
foreach path $p_i \in P$ do
    $p'_i \leftarrow$ update $p_i$ according to user-configured features;
    $value \leftarrow$ make_string($p'_i$);
    if sweep is the task objective then
        $key \leftarrow key[g_t^i]$;
    else
        $key \leftarrow -1$;
    end
    Emit make_pair($key$, $value$);
end
```

Algorithm 35: Peeler(r)

```
Input: an unique key/value pair $r$
Output: an emitted key/value pair
$key \leftarrow r.key$;
$P \leftarrow$ paths parsed from $r.value$;
sort $P$ in non-increasing order of criticality;
$P' \leftarrow \{k$ frontmost elements in $P\}$;
$value \leftarrow$ make_string($P'$);
Emit make_pair($key$, $value$);
```

Lemma *There are either $|T|$ or $O(P|T|)$ mapper calls on a distributed cluster with P computing processors.*

Proof The execution of each benchmark has two possible conditions, either the number of tests is greater than the number of computing processors or the number of tests is less than the computing resources. For the former case, each test is processed by an independent mapper function and thus there are totally $|T|$ mapper calls. For the later case where the number of tests is less than the available core count, each test is decomposed into $O(P)$ task graphs. Hence, there are totally $O(P|T|)$ mapper calls.

Lemma *There is only one reducer call for block report while there are $|T|$ reducer calls for sweep report.*

Proof For block report, the key/value pairs emitted by the extractor all have the same key value (i.e., -1). Therefore, the collate operation produces only one key/value pair for the following reduce operation. On the other hand, the intermediate key values for sweep report adhere to the test indices of the task graphs. Therefore, the collate operation produces a total of $|T|$ distinct key/value pairs for the following reduce operation.

3.4.4 Main Program

The main program of our PBA framework is shown in Algorithm 36. The first two lines perform map operations that call Algorithm 33 to generate a set of task graphs. Using the task graphs as input, the next three lines follow the canonical form of a MapReduce program, where map operations call Algorithm 34 to perform path extraction on each task graph, and reduce operations call Algorithm 35 to peel out the final solution. Prior to the function return, paths are parsed from the output values of our reducer (line 6:15). Each path is conventionally tagged with the corresponding test index which can be retrieved from the key value (line 10).

Theorem The proposed framework is correct.

Proof Proving the correctness of our framework is equivalent to showing that the path set from the input of a reducer contains the top-k critical

```
Algorithm 36: MapReducePBA($G$)
Input: a circuit graph $G$, a test set $T$
Output: an analyzed path set
$D \leftarrow$ Map($T$, Generator) ;
$G_T \leftarrow$ task graphs parsed/read from $D$;
$M \leftarrow$ Map($G_T$, Extracter) ;
$C \leftarrow$ Collate($M$);
$R \leftarrow$ Reduce($C$, Peeler);
if sweep is the task objective then
    $P \leftarrow \phi$;
    foreach pair $r$ in $R$ do
        $P_r \leftarrow$ paths parsed from $r.value$;
        Tag $P_r$ with the test index $t$ retrieved from $r.key$;
        $P \leftarrow P \cup P_r$;
    end
    return $P$
end
$P \leftarrow$ paths parsed from the value in $R$;
return $P$
```

paths for the corresponding test. Recalling that the input of our reducer is an unique key/value pair. The key indicates the test index and the value is a concatenated list of values with each value storing the top-k critical paths of a task graph generated from this test. It is obvious by set properties that the top-k critical paths for this test must be a subset of the path set stored in the value list. Since our reducer is in fact a sorting process, the output is the value that stores the final top-k critical paths for this test. Notice that for block report the test index is nominal while this fact has no impact on the truth of this proof.

3.5 Data Management

Efficient data management is crucial to a MapReduce program. We discuss in this section some technical details and data management through our implementations.

3.5.1 Data Locality

Exploiting the data locality is an important principle of efficient MapReduce programs. Improving the data locality can reduce the network overhead during the execution, which in turn tells better runtime performance. In order to improve the data locality, each processor stores a replicate of the circuit graph in its own local memory. Despite higher memory demand, accesses to the circuit graph such as generation of task graphs and extraction of critical paths are reached in hand without extra data passing which is normally time-consuming.

3.5.2 Storage Efficiency

The communication load is a non-negligible cost for a MapReduce program in particular during the collate operation. Passing long values of paths gives rise to the problem of frequent memory allocation which is typically time consuming. In order to minimize the communication load, explicit path traces are stored in the memory of each individual machine. Each path is tagged with an unique index which is used to represent the storage address and machine number or the temporary file name. Paths are passing through these indices during collate operation and the final recovery of path traces is done by indexing back to these tags.

3.5.3 Hidden Reduce

Another way to alleviate the communication overhead is to avoid unnecessary data passing during the collate operation. Within the same processor, a reduce operation before the collate call is pre-applied to those path sets having the same key label. We term this reduce operation as “hidden reduce” because it is implicitly processed after each mapper call of path extraction. In other words, multiple data with the same key label in each processor are merged first to reduce the amount of data passing. It is obvious by Theorem 16 the optimality of the final solution is not affected by this hidden reduce operation.

3.6 Experimental Results

Our program is implemented in C++ language on a 64-bit linux operating system. The C++ based MR-MPI API is used as our MapReduce library [31]. Evaluation is taken on an academic computer cluster which has over 500 compute nodes. Each compute node is configured with 16 Intel 2.6 GHz cores and 128 GB RAM. The network infrastructure uses 384-port Mellanox MSX6518-NR FDR InfiniBand in order to offer high-speed interconnect between clusters. Access to the compute nodes for running a program is done via a script submission specifying the number of process cores or threads to be used.

Table 3.1: Statistics of the Benchmarks from the 2014 TAU Timing Analysis Contest [1]

Circuit	$\|v\|$	$\|e\|$	$\|i\|$	$\|o\|$	# Tests	# Paths
combo5	2051804	2228611	432	164	79050	19227963
combo6	3577926	3843033	486	174	128266	19227963
combo7	2817561	3011233	459	148	109568	19227963

$|V|$: # of pins. $|E|$: # of edges. $|I|$: # of primary inputs.
$|O|$: # of primary outputs. # Tests: # of setup/hold tests.
# Paths: maximum # of data paths per test.

Experiments are undertaken on the three largest benchmarks, combo5, combo6, and combo7 from the 2014 TAU timing analysis contest [1]. Each of the three test cases is created by combining a set of industrial circuits (e.g., vga_lcd, systemcde2, aes_core, des_perf, usb_funct, wb_dmav, systemcaes, and tv80) that were already open-source to academia. The test case combo5 is the combination of circuits vga_lcd, usb_funct, des_perf, tv80, wb_dmav, and systemcaes. The test case combo6 is the combination of circuits vga_lcd, aes_core, des_perf, usb_funct, systemcde2, and tv80. Test test case combo7 is the combination of circuits vga_lcd, tv80, aes_core, systemcaes, and vga_lcd. Statistics of these test cases are summarized in Table 3.1. All test cases are million-scale circuit graphs and the number of tests could reach up to 128266 in combo6.

3.6.1 Baseline Setting

We configure CRPR as the baseline application in our PBA framework. CRPR is an important step during the signoff timing cycle. Without CRPR, the signoff timing analyzer reports worse violation than the true timing properties owned by the physical circuits. The 2014 TAU timing analysis contest has addressed this issue in order to motivate novel ideas for fast and accurate path-based CRPR [1]. The optimal path ranking algorithm proposed by the first-place winner, UI-Timer, is applied to our path extractor [8]. In order to enable CRPR, the third line of Algorithm 34 is implemented as follows: For each path being iterated, the common clock segment is found by a simple walk through the corresponding launching clock path and the capturing clock path. The path slack is then adjusted by the amount of pessimism on the common segment. With CPPR, the values of path slacks are in general increased after the clock network pessimism is removed. The number of failing tests was able to be reduced by even more than a half [8].

3.6.2 Performance Characterization

We begin by discussing the generic performance of our MapReduce-based PBA. Evaluation is undertaken through cross combinations of path count (i.e., k) and core count in running our program. We request 1 to 10 compute nodes with each configured by 10 cores. That is, the core count varies from 10 to 100 using 10 as the scaling interval. A special case with only 1 core is also evaluated in order to demonstrate the baseline without any parallelism. The path count starts at 1 and varies from 10 to 100 using 10 as the scaling interval. A total of 121 combinations of path count and core counts are executed for each benchmark.

The number of key/value pairs processed on each circuit benchmark is illustrated in Figure 3.4. It can be observed that for each circuit graph the number of key/value pairs processed by map and reduce operations grows as the path count increases. Notice that the path count is the only factor that contributes to the growth of the number of key/value pairs since the construction of key/value pairs is dedicated to paths. The largest number appears in the report of 100 paths, in which the program generated 3953344 key/value pairs for combo5, 7114972 key/value pairs for combo6, and 6696880

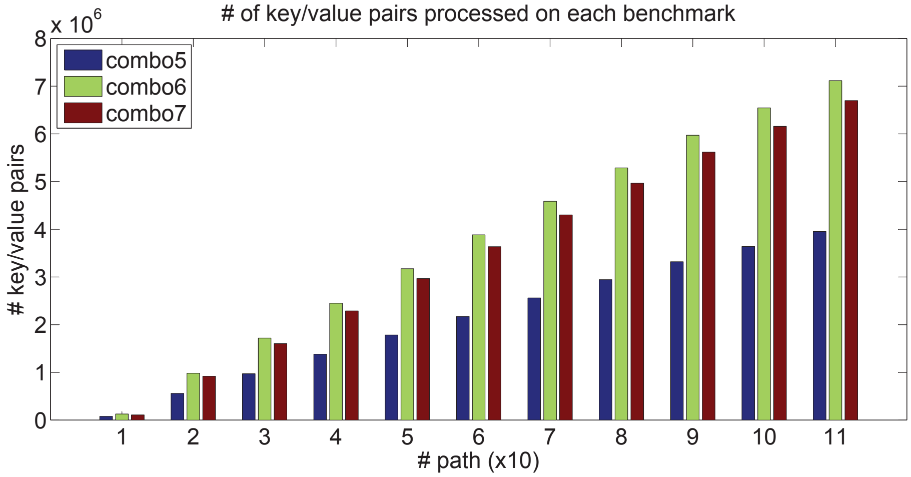


Figure 3.4: Bar chart of the number of key/value pairs processed on each circuit benchmark.

key/value pairs for combo7. In general, the more the number of key/value pairs is, the higher the runtime and memory storage the program demands.

The overall performance of our MapReduce-based PBA is shown in Figure 3.5. The left two columns of plots show the runtime value and memory usage of our program under block report, while the right two columns show the plots under sweep report. We first discuss the runtime performance of our program. In a rough view, the runtime scales down drastically as the core count increases. Using only a single core without any parallelism, the program took up to (i.e., among all path settings) 14.03 (13.92) minutes, 37.76 (39.53) minutes, and 27.41 (27.07) minutes to accomplish block (sweep) reports for combo5, combo6, and combo7, respectively. It can be seen that the runtime significantly goes down when MapReduce begins distributing works across processors. Even using only 10 processors, the runtime values can be significantly reduced to 2.92 (2.91) minutes, 8.22 (8.24) minutes, and 4.63 (5.55) minutes under block (sweep) reports of combo5, combo6, and combo7, respectively. The slope of the runtime reduction can be clearly seen in the sliced 2D plot fixing path count to 100 in Figure 3.6. Within a single minute, all tests can be accomplished using approximately 40 cores, 100 cores, and 80 cores, for combo5, combo6, and combo7, respectively.

Figure 3.7 discovers the runtime portions taken by map operations, collate operations (i.e., process communication or "Comm" for short), and reduce operations. We measure the runtime portion as an average value across all

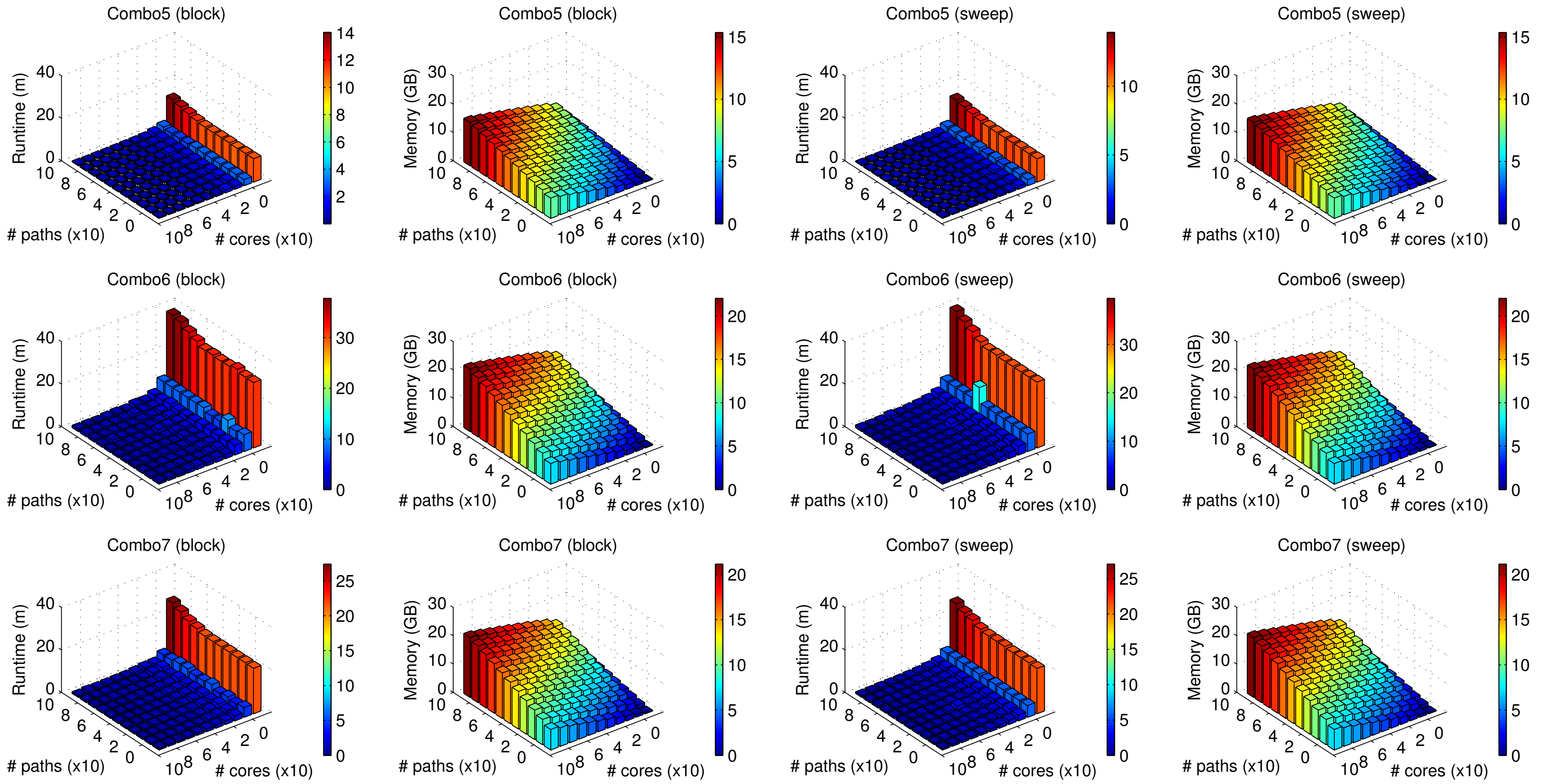


Figure 3.5: Performance characterization of our MapReduce-based PBA on circuit benchmarks combo5, combo6, and combo7 under block report and sweep report. Within a single minute, all tests can be accomplished using approximately 40 cores, 100 cores, and 80 cores, for combo5, combo6, and combo7, respectively.

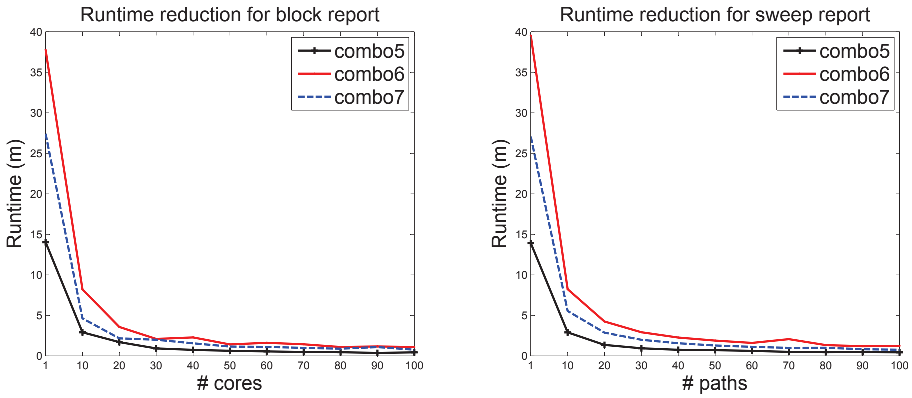


Figure 3.6: Runtime reduction versus core count.

different settings of path counts and core counts. We have observed that reduce operations spend the least amount of time (< 1%) comparing to the others since it involves only string parsing and value sorting. On the other hand, the time spent on map operations occupies the majority of the entire runtime. This is because map operations are responsible for the generation of task graphs and the extraction of critical paths, which are relatively expensive computations. For all benchmarks, more than 90% of the entire runtime is taken by map operations. The rest portion of the runtime is occupied by the collate operation, from which we can see about 4–5% of the entire runtime is spent on the process communication. In fact, without applying the trick mentioned in Section 3.5.2, the process communication burdens the entire runtime by over 20%.

Next we discuss the memory cost of our program. The amount of memory usage is measured by the peak moment during the execution across all processors (i.e., including the master processor). Generally speaking, the amount of memory usage grows as the increase of either path count or core count. The peak memory usage we observed are approximately 15 GB, 22 GB, and 21 GB for combo5, combo6, and combo7, all under sweep report with 100 cores and 100 paths, respectively. We provide two extra sliced plots from the sweep report in Figure 3.8 to show clearer memory cost in terms of the growth of (1) core count with path count fixing to 100 and (2) path count with core count fixing to 100. As the path count or the core count increases, the amount of memory usage grows gradually except for the sharp spot at the 10-core level where the distributed MapReduce begins taking effect.

To sum up, the experimental results have demonstrated the performance

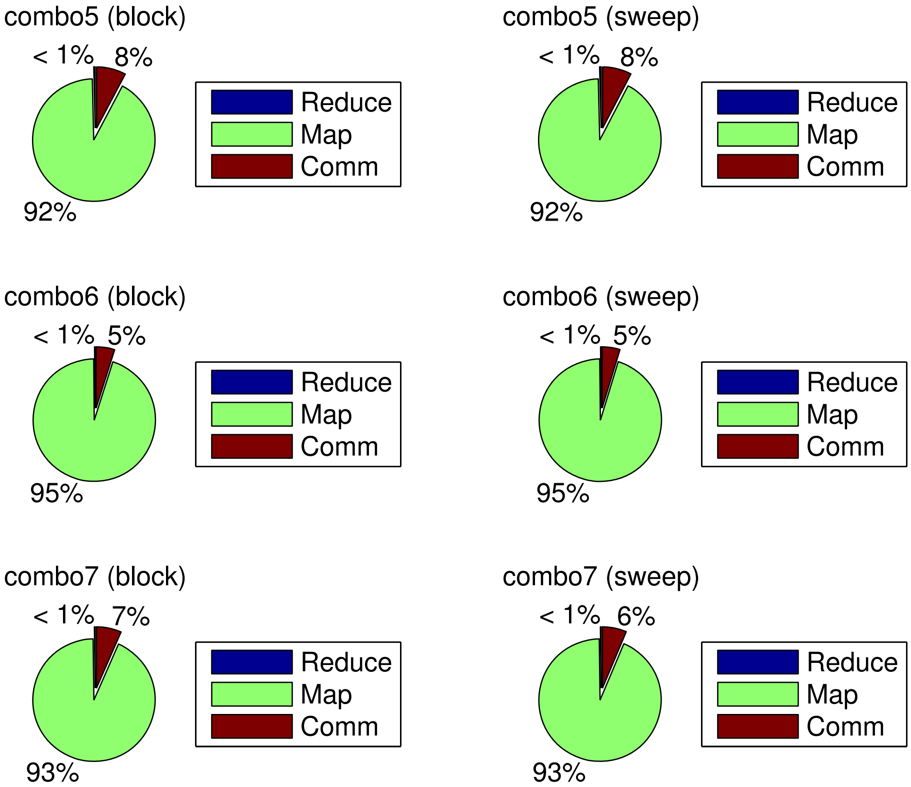


Figure 3.7: Runtime portion of map operations, reduce operations, and process communication.

of our PBA framework with MapReduce. It is highly scalable as we have seen a significant runtime reduction as the core count grows. Even in the first level at which only 10 cores are involved in parallelism, the runtime is decreased by 75–86% across all runs. From the storage point of view, the memory consumption of our approach is fairly reasonable. At the highest peak we have observed in running combo6 with 100 cores and 100 paths, the total amount of memory demanded by our program is about 22 GB. In other words, the average amount of memory usage per processor is less than 1 GB. This evidence has justified the practical viability of our approach. The substantial speedup we have obtained is beneficial for the discovery of a way to fast timing closure.

3.6.3 Comparison with Multi-Threading

We evaluated in this section the competence of our approach over the implementation using multi-threading, another popular type of parallel pro-

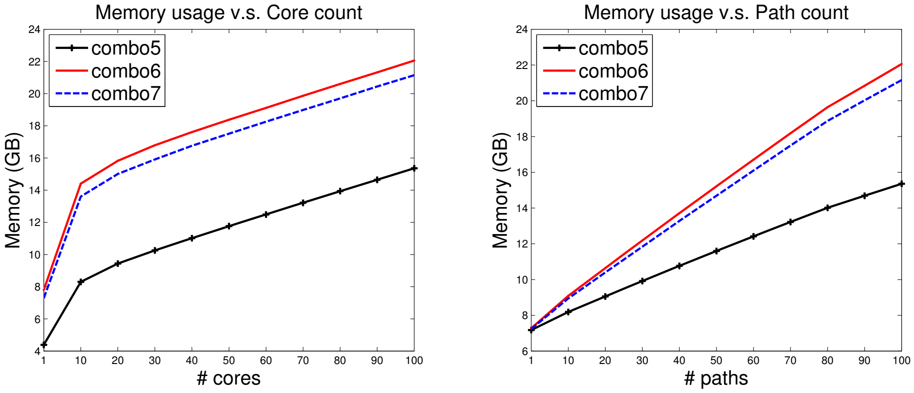


Figure 3.8: Memory usage in terms of path count and core count.

gramming with shared-memory model. The inherent architecture of a multi-threaded program is distinct from that of distributed computation such as the MapReduce programming environment. In multi-threaded programming, multiple threads or processors can operate independently on a standalone machine but share the same memory resources. The memory bandwidth of the machine typically dominates the entire runtime performance. As a result, the scalability of multi-threaded computation is typically not as decent as the one of distributed computation. Several libraries for using shared memory such as OpenMP and POSIX are reachable in the public domain [22, 32].

We refit our MapReduce program to the multi-threaded version by replacing the mapper calls and reducer calls with parallel for loop (e.g., #pragma omp statement) using the API from OpenMP 3.0 [22]. In our cluster each compute node is configured with 16 Intel 2.60 GHz cores and 128 GB RAM in a standalone machine. Up to 16 threads or 16 processors can be concurrently executed using either multi-threaded computation or distributed MapReduce operations. Due to the architectural limitation of multi-threading, evaluations are undertaken in a single compute node using different core counts from 1 to 16. The performance differences between multi-threading and MapReduce are interpreted in terms of runtime values and memory usage, as illustrated in Figure 3.9. For page efficiency, we discuss only the experiment of block report with the single-most critical path.

The strength of MapReduce over multi-threading is clearly demonstrated by the runtime plot in Figure 3.9. In comparison to multi-threading, our MapReduce program obtains higher runtime speedup and better scalability as core count grows up. The largest difference we observed was in combo6

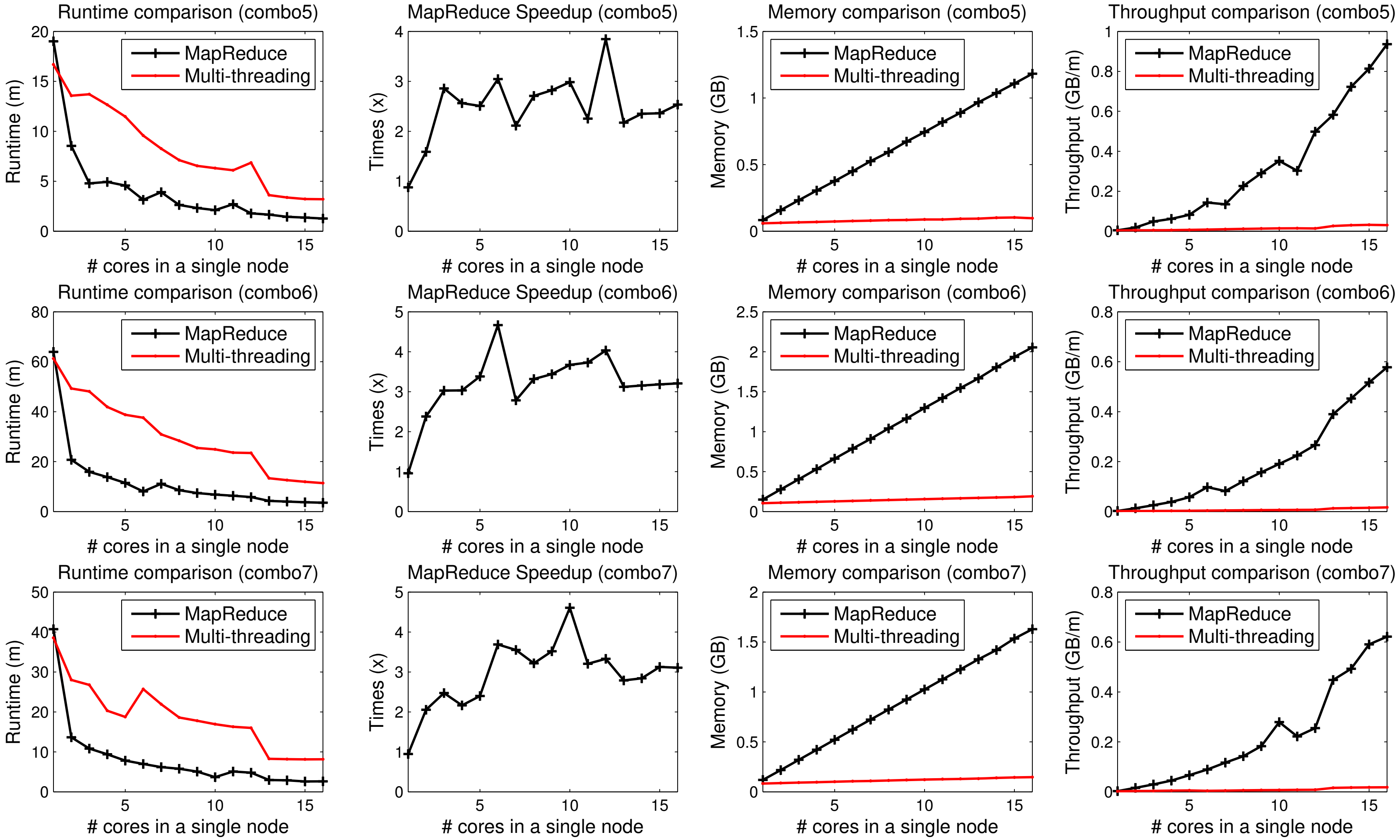


Figure 3.9: Performance comparison between MapReduce and multi-threading on a single compute node.

with two cores, where our MapReduce program accomplished all tests by 32 minutes faster than the multi-threaded implementation. Similar trends can also be discovered in other two cases. The reason for having our MapReduce program perform worse at the level of one core comes from the redundant overhead of key/value processing because of the null parallelism. Nevertheless, such negative margins are solely less than 3 minutes.

It is expected that our MapReduce program consumes higher memory requirements than the multi-threaded implementation. The distributed computation of MapReduce requires an individual block of memory to be allocated for each processor. As shown in the memory comparison in Figure 3.9, the memory cost of our MapReduce program is linearly proportional to the growth rate of the core count. On the other hand, the amount of memory usage in multi-threading is relatively constant regardless of the increase of core count. Despite less memory cost by multi-threading, the performance of concurrent access to the same global memory block is limited by the memory bandwidth. It can be clearly seen in Figure 3.9 the process throughput grows poorly compared to the curve achieved by distributed MapReduce. As a consequence, the runtime performance of multi-threading is not as promising as distributed MapReduce even in a standalone machine.

3.7 Conclusion

In this chapter we have presented a fast PBA framework with MapReduce. To the best knowledge of the authors, this work is the first attempt to handle the PBA problem using the MapReduce programming paradigm. We have achieved a success in accelerating PBA by a substantial order of magnitude in comparison to non-MapReduce implementations such as single core and multi-threading. The experimental results have demonstrated the pronounced performance of our approach whereby million-scale circuit graphs can be quickly and correctly analyzed within a few minutes on a distributed computer cluster. Our work can be beneficial in assisting designers in speeding up the lengthy design cycles of signoff timing.

CHAPTER 4

A DISTRIBUTED TIMING ANALYSIS FRAMEWORK FOR LARGE DESIGNS

4.1 Introduction

As design complexities continue to grow larger, the need to efficiently analyze circuit timing with billions of transistors across multiple modes and corners is quickly becoming the major bottleneck to the overall chip design closure process [28]. In order to alleviate long runtimes, designers break down the design into several hierarchical partitions or boxes, apply macro-modeling (abstraction) to each hierarchical box, and run multi-threaded timing analysis (MTA) on a single machine [2]. However, it has been reported that a complete MTA on a design with 2 billion transistors can consume 400 GB memory. Building such a high-end computer is costly and unscalable to the ever-increasing design complexities. As a result, trends are shifting toward distributed timing analysis (DTA).

Nevertheless, very little research have been done on DTA. State-of-the-art distributed systems such as Hadoop MapReduce, Cassandra, Shark, Mesos, and Spark are mainly developed for big-data applications [30, 33]. Nonetheless, big-data applications have many distinctive characteristics compared to timing analysis. First, big-data applications are data-intensive whereas timing analysis is more computation-driven. Second, parallelism is natural in big-data processing. Large data sets can be arbitrarily broken down to independent pieces followed by massively parallel MapReduce operations. However, timing analysis is highly iterative and loop-dependent, making it hard to integrate with MapReduce paradigm. Besides, these systems mainly work on functional or Java virtual machine (JVM) languages such as Scala, Java, and R. Implementations using high-performance C/C++ are ill-supported.

The real problem is that most EDA tools are developed based on high-performance C/C++. A benchmark for language performance and system

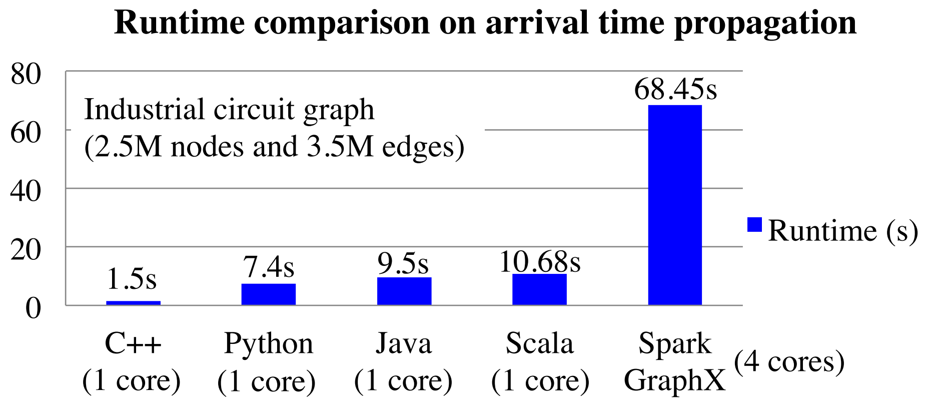


Figure 4.1: The need of specialized DTA framework.

model on computing the arrival time from an industry circuit is shown in Figure 4.1. It is observed that C++ is faster than mainstream big-data languages such as Python, Java, and Scala. Compared to the well-known Spark GraphX, a big-data model for distributed graph processing, the performance gap raises a big applicability concern. These evidences have convinced a need of specialized DTA framework. Consequently, we introduce in this chapter a DTA framework for large designs. Key features of our framework are highlighted as follows:

- **General design partitions:** Our framework is developed for general design partitions. Logical, physical, or hierarchical design partitions are all stored in a distributed file system.

- **Multi-program-multi-data (MPMD) paradigm:** Our framework follows the MPMD paradigm. Through a common communication interface, designers can create customized codes for different partitions.

- **Non-blocking socket IO:** Our framework is developed using C/C++ socket library. We configure non-blocking transmission control protocol (TCP) channels so as to overlap communication and computation.

- **Event-driven environment:** Our framework is event-driven. Data updates are executed asynchronously in response to user-registered callbacks. The event loop also enables persistent in-memory processing.

- **Efficient messaging interface:** Our framework is message-efficient. The overhead between structured data serialization and TCP byte stream de-serialization is leveraged using scalable Protocol Buffer [34].

We have evaluated our framework on a commodity cluster with hundreds of machines and successfully performed DTA on large industry designs.

4.2 Problem Formulation

The input is a set of partitions broken from a flat design across different logical cones, physical locations, or hierarchical boundaries (chip, unit, macro). Each design partition file acts as a black box to others and contains a directed acyclic timing graph. Multiple partitions are implicitly connected together through a top-level design file. An example of two-level hierarchical partitions is shown in Figure 4.2. The top-level design has three primary inputs PI1, PI2, and PI3, and one primary output PO1. It connects to two hierarchical macros M1 and M2 through their primary inputs M1:PI1, M1:PI2, M2:PI1, and M2:PI2, and primary outputs M1:PO1 and M2:PO1, respectively. In addition, a set of timing assertion files specifying the initial timing condition on source ports (PI1, PI2, PI3, and PO1) is also given.

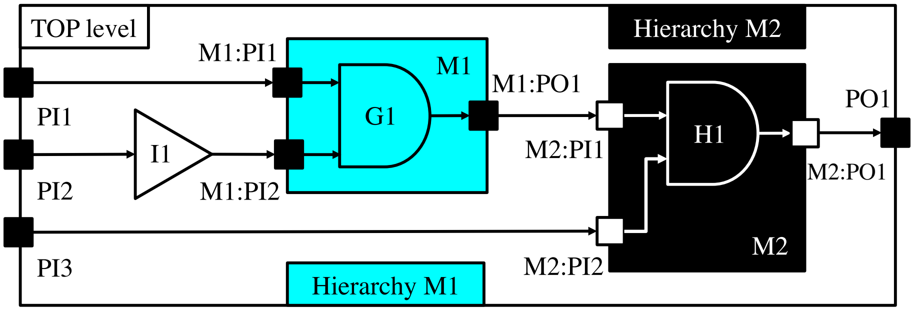


Figure 4.2: Example of two-level hierarchical partitions.

Objective: *Given a set of design partition files and timing assertions, develop a distributed timing framework over a network cluster and perform distributed timing analysis.*

4.3 Framework

The overview of our DTA framework is shown in Figure 4.3. The input is a set of design partitions and timing assertions. Files are stored in distributed file system such as general parallel file system (GPFS), andrew file system

(AFS), and/or hadoop distributed file system (HDFS). Our framework has one program for server and multiple programs for clients. Each program performs the timing analysis on one design partition. Communications are handled indirectly through the server program. Programs are launched on multiple machines through a network cluster manager such as LSF, Mesos, Helix, Zookeeper, and OpenLava, that supports remote job execution [30, 33].

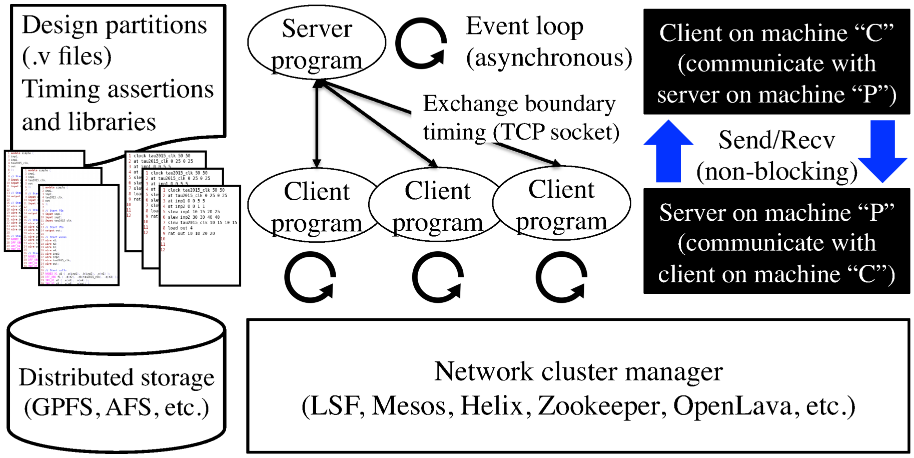

Design partitions
(.v files)
Timing assertions
and libraries
Server program
Event loop (asynchronous)
Client on machine "C" (communicate with server on machine "P")
Exchange boundary timing (TCP socket)
Send/Recv (non-blocking)
Client program
Client program
Client program
Server on machine "P" (communicate with client on machine "C")
Distributed storage (GPFS, AFS, etc.)
Network cluster manager
(LSF, Mesos, Helix, Zookeeper, OpenLava, etc.)

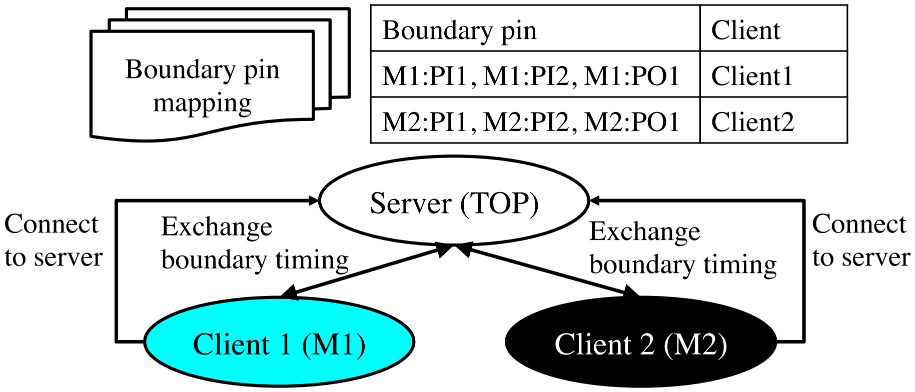
Figure 4.3: Overview of our DTA framework.

4.3.1 Distributed Storage and Cluster Manager

To avoid complete copies of the data set on machines, the input files are stored in a distributed file system. A distributed file system offers location-independent addressing that is shared by being simultaneously mounted on a cluster of multiple machines. It aims for *transparency* in that users access the system in the same way as a local file system. Multiple data sets live together and can be accessed by any machines. Besides, our framework requires a cluster manager to work with the distributed file system. Each machine node runs application programming interface (API) offered by the cluster manager to manage and configure services such as remote job execution and status query over cluster nodes. Our framework is not restricted to certain distributed file systems and cluster managers. The common features such as distributed file mounting, remote job execution, and machine status query offered by the state of the art are sufficient for our development.

4.3.2 Software Architecture

The software architecture of our framework follows the multiple-program multiple-data (MPMD) paradigm. We define a *communication group* as one *server* program along with multiple *client* programs. Forming a communication group is particularly useful for the standard design partition flow. The server program works on the top-level design while client programs handle other design partitions. Our architecture can be easily extended to recursive partitions (i.e., a partition spawns child partitions and so on in a tree manner) by creating a new communication group for each additional layer of partitions. The server program can be viewed as a *communicator*, dealing with all timing exchanges among partitions based on TCP socket send/receive calls. Both server and client programs perform the real tasks on timing propagations. Through a common communication interface, designers can customize or safely evolve their timing routines for individual partitions.



Boundary pin	Client
M1:PI1, M1:PI2, M1:PO1	Client1
M2:PI1, M2:PI2, M2:PO1	Client2

Figure 4.4: Server-client model for the two-level hierarchical partitions in Figure 4.2.

Figure 4.4 presents an example of the server-client model for the hierarchical partitions in Figure 4.2. The server is responsible for the top-level design and two clients are required for hierarchical macros M1 and M2. Besides, server maintains the mapping between each boundary pin and the corresponding client so that up-to-date timing can be delivered to the correct host. For instance, server starts propagating the timing from primary input PI1 and stops at the hierarchical primary input M1:PI1. The up-to-date timing is then sent to the client 1 for further propagation and so on.

4.3.3 Non-Blocking IO and Event Loop

Network latency is typically at least ten times higher than in-memory reference [35]. This can cause performance degradation if the program is blocked by waiting for communication. It is desirable that communication can be executed autonomously by an intelligent *non-blocking* controller. A non-blocking send/receive call initiates a send/receive request but does not complete it. The call returns immediately to the user's program, leaving the communication taken over by another lightweight thread from operating system (OS) kernels. Computation can run simultaneously while waiting for the send/receive to complete. This implies a need of an extra procedure polling the communication status from the perspective of program development. However, the network speed is hardware-dependent and it might end up with nothing but a waste of time on polling.

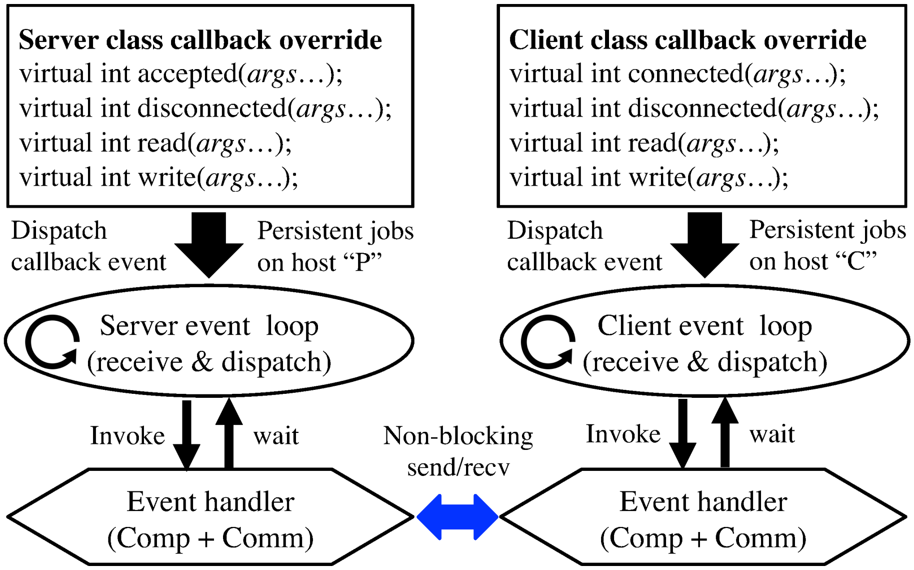
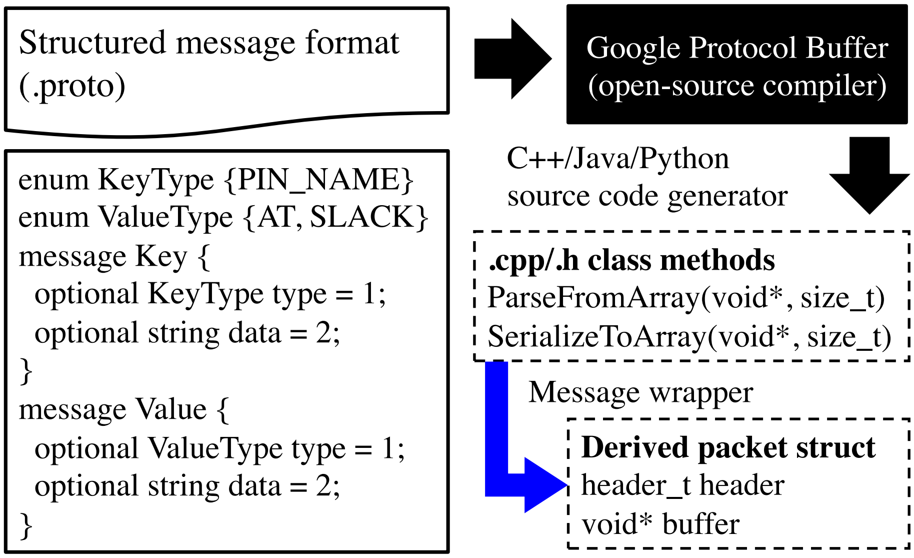

Server class callback override
virtual int accepted(*args*...);
virtual int disconnected(*args*...);
virtual int read(*args*...);
virtual int write(*args*...);
Client class callback override
virtual int connected(*args*...);
virtual int disconnected(*args*...);
virtual int read(*args*...);
virtual int write(*args*...);
Dispatch callback event
Persistent jobs on host "P"
Dispatch callback event
Persistent jobs on host "C"
Server event loop (receive & dispatch)
Client event loop (receive & dispatch)
Invoke
wait
Non-blocking send/recv
Invoke
wait
Event handler (Comp + Comm)
Event handler (Comp + Comm)

Figure 4.5: Event-driven environment in our framework. Jobs are persistent in memory through event loops. Non-blocking socket IO enables overlap of computation (comp) and communication (comm).

In contrast to actively polling the communication status, event-driven programming is a more favorable solution. Figure 4.5 presents the event-driven environment in our framework. Our framework applies the open-source package, *libevent*, as the event engine [35]. We define callbacks for various socket events such as new connection online, message send/receive, and connection offline. Applications then dispatch the program into an *event loop* and these callbacks are autonomously invoked by an event handler. Designers can terminate the programs through special events such as interactive query, time-out, and signal interrupt. As a byproduct of the event loop, jobs are *persistent in memory*, which is an important feature for computation-driven timing applications.

4.3.4 Efficient Messaging Interface

Reducing the messaging overhead is pivotal especially considering the conversion between structured data (e.g., class, pointer, random memory access) in application level and unstructured TCP byte stream in the communication world. Structured data need *serialization* before message send and unstructured TCP byte stream needs *de-serialization* after message read. Apparently, hand-crafting and hard-defining this infrastructure is error-prone and inflexible. Instead, we employ the widely used tool, *protocol buffer*, from the big-data community [34]. Protocol Buffer is Google's language-neutral and extensible mechanism for message serialization and de-serialization. It compiles user-defined message format into C++ classes that offer heavily optimized methods (e.g., compression, decoding) for data conversion. The concept is illustrated in Figure 4.6.


Structured message format (.proto)
Google Protocol Buffer (open-source compiler)
C++/Java/Python source code generator
.cpp/.h class methods
Message wrapper
Derived packet struct

```
enum KeyType {PIN_NAME}
enum ValueType {AT, SLACK}
message Key {
  optional KeyType type = 1;
  optional string data = 2;
}
message Value {
  optional ValueType type = 1;
  optional string data = 2;
}
```

```
ParseFromArray(void*, size_t)
SerializeToArray(void*, size_t)
```

```
header_t header
void* buffer
```

Figure 4.6: Integration of Google's protocol buffer into our messaging interface for data conversion between application-level development and socket-level streams.

As shown in Figure 4.6, we define *key* and *value* for our application. A key and a value are simply bytes of strings of arbitrary length which are logically associated with each other and thus can represent generic timing data. For instance, a key can be the pin name and the value stores the corresponding timing numeric such as arrival time and slack. However, simply using key-value data is not sufficient since non-blocking socket IO might invoke the callback wherever the message is incomplete (e.g., every 4 K bytes received) due to the network C10K issue [35]. In order to handle the data appropriately, we wrap the data into a *packet* which contains, in addition to the data field, a header indicating the message size. It is the task of the receiver to inspect the header and determine when the length of byte stream is enough for processing the data.

4.4 Distributed Timing Algorithm

In this section, we develop a distributed timing algorithm based on our framework. We shall discuss the flow of the server program and client programs, and the callbacks corresponding to different events. Due to the space restriction, we focus on the generic concept of timing propagation.

4.4.1 Server Program

The main body of the server program is presented in Algorithm 37. Algorithm 37 takes three arguments, the input data D, the host H of the server program, and user data U for callback convention, and generates the timing analysis report. It first parses the timing graph from the input data D and initiates a TCP server socket binding to host H (line 1:2). Then an event base B is created (line 3). An event base holds a set of events and polls to determine which events are active [35]. We add a listener event to B to note the callback *AcceptClientConnection* (in Algorithm 38) for any new TCP client connections (line 4). Finally, the event base is dispatched and the program enters an event loop (line 5).

Algorithms 38 and 39 present the two callbacks in server's program. Algorithm 38 is invoked when a new client connection arrives. An event callback of message read is created for the new client socket (line 3). The detail of

Algorithm 37: Server(D, H, U)

```
Input: input data $D$, host $H$, user data $U$
Output: timing analysis report
$G \leftarrow$ parse_timing_graph($D$);
$S \leftarrow$ create_TCP_server_socket($H$);
$B \leftarrow$ create_event_base();
add_listener_event($B$, $S$, $U$, AcceptClientConnection);
dispatch_event_base($B$);
```

Algorithm 38: AcceptClientConnection(L, U)

```
Input: listener $L$, user data $U$
$B \leftarrow$ get_event_base($L$);
$S \leftarrow$ get_socket_info($L$);
add_socket_read_event($B$, $S$, ServerReadCallback, $U$);
```

read callback is given in Algorithm 39. It iterates each complete packet over the TCP byte stream M (line 2) and de-serializes the data into key-value pairs Ω (line 3). At each iteration, the program branches in response to different packet types, which can be either the notice of a new boundary pin where we build the mapping to the corresponding client identity (line 5:8), or timing update at boundary pins in which we maintain a candidate set Δ of pins for timing propagation (line 16:20). In the former case, the source ports are added to the candidate set Δ when all required clients are online (line 9:14). Then, we carry out the timing propagation from the candidate set and return a set Θ of key-value pairs where the key k indicates a boundary pin at which this timing propagation stops and the value stores up-to-date timing (line 23). Finally, each of these key-value pairs is sent to the corresponding client (line 24:28).

4.4.2 Client Program

The main body of the client program is given in Algorithm 40. In a rough view, the procedure is identical to the counterpart of server except the callback for being connected sends server a packet registering the identity of each boundary pin in the design (line 4 in Algorithm 40 and line 4:8 in Algorithm 41). This step is necessary for the server program to keep track of the

Algorithm 39: ServerReadCallback(S, M, U)

```
Input: socket descriptor $S$, message $M$, user data $U$
$\Delta \leftarrow \phi$;
foreach complete packet $i \in M$ do
    $\Omega \leftarrow$ deserialize_data($i$);
    switch $i.type$ do
        case BoundaryRegistration do
            foreach key-value pair $(k, v) \in \Omega$ do
                map_boundary_pin_to_socket($k$, $S$);
            end
            if all clients are online then
                foreach source port $r$ in top-level design do
                    $v \leftarrow$ initial_timing_assertion($r$);
                    $\Delta \leftarrow \Delta \cup \{$make_kv_pair($r$, $v$)$\}$;
                end
            end
        end
        case UpdateBoundaryTiming do
            foreach key-value pair $(k, v) \in \Omega$ do
                $\Delta \leftarrow \Delta \cup \{(k, v)\}$;
            end
        end
    end
end
$\Theta \leftarrow$ propagate_timing_and_get_new_boundary_pins($\Delta$);
foreach key-value pair $(k, v) \in \Theta$ do
    $j \leftarrow$ serialize_data($k$, $v$);
    $c \leftarrow$ get_boundary_pin_client_socket($k$);
    send_packet($c$, $j$, UpdateBoundaryTiming);
end
```

mapping between a boundary pin and its client identity. As presented in Algorithm 42, the read callback in the client program resembles the procedure in Algorithm 39. From the viewpoint of client, there is no need of branch for boundary pin registration. We only maintain a candidate set of pins received from the server for timing propagation. After timing propagation, boundary pins with up-to-date timing values are packeted and sent to the server (line 12:16).

```
Algorithm 40: Client($D$, $H$, $U$)
  Input: input data $D$, server host $H$, user data $U$
  Output: timing analysis report
  $G \leftarrow$ parse_timing_graph($D$);
  $S \leftarrow$ create_TCP_client_socket($H$);
  $B \leftarrow$ create_event_base();
  add_connect_event($B$, $S$, $U$, Connect);
  dispatch_event_base($B$);
```

```
Algorithm 41: Connect($L$, $U$)
  Input: listener $L$, user data $U$
  $B \leftarrow$ get_event_base($L$);
  $S \leftarrow$ get_socket_info($L$);
  add_socket_read_event($B$, $S$, ClientReadCallback, $U$);
  $\Delta \leftarrow \Phi$;
  foreach boundary pin $p$ in the design do
      $\Delta \leftarrow \Delta \cup$ make_kv_pair($r$, NULL);
  end
  send_packet($S$, serialize_data($\Delta$), BoundaryRegistration);
```

4.4.3 Timing Propagation

We have presented our framework and developed the program architecture for distributed timing. Although designers can customize their timing routines (in particular, line 23 in Algorithm 39 and line 12 in Algorithm 42), processing the timing propagation exhibits high similarities to finding the shortest and the longest paths in a graph [2, 36]. In this regard, we introduce two techniques that are generically useful for the development of timing propagation based on our framework.

Algorithm 42: ClientReadCallback(S, M, U)

```
Input: socket descriptor $S$, message $M$, user data $U$
$\Delta \leftarrow \phi$;
foreach complete packet $i \in M$ do
    $\Omega \leftarrow$ deserialize_data($i$);
    switch $i.type$ do
        case UpdateBoundaryTiming do
            foreach key-value pair ($k$, $v$) $\in \Omega$ do
                $\Delta \leftarrow \Delta \cup \{(k, v)\}$;
            end
        end
    end
end
$\Theta \leftarrow$ propagate_timing_and_get_new_boundary_pins($\Delta$);
foreach key-value pair ($k$, $v$) $\in \Theta$ do
    $j \leftarrow$ serialize_data($k$, $v$);
    send_packet($S$, $j$, UpdateBoundaryTiming);
end
```

Frontier Propagation

The timing graph is given as a directed acyclic graph. Maintaining the topological ordering of the graph during the timing propagation is a common and important way to correct results [2]. We refer to this topologically ordered propagation as *frontier propagation*. Since our framework is non-blocking and asynchronous, frontier propagation can start moving forward whenever a new timing update arrives at a boundary pin, and stop at the pin with at least one incoming arc that has not experienced the frontier propagation. An illustrative example of forward propagation is shown in Figure 4.7. The arrival of up-to-date timing at pin F:o invokes the callback to push frontier propagation forward until pin I:o due to the waiting for message at pin U:o. If resources are available, advanced techniques such as pipelined frontier propagation proposed by [36] can be applied as well.

Speculative Propagation

It can be observed in Figure 4.7 that the network delay might result in resource un-utilization (thread waiting for work). This is because there are no

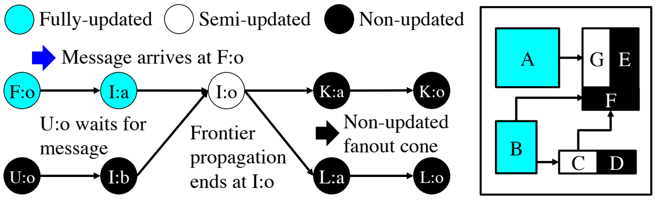


Figure 4.7: Frontier propagation follows the topological ordering of the timing graph.

active events at the moment frontier propagation stops and the main thread becomes idle. To enable further overlap of communication and computation, an un-utilized thread can continue to perform *speculative propagation* from the pin at which frontier propagation stops. The concept of speculative propagation is shown in Figure 4.8. Speculative propagation aims to find the dominant minimum or maximum paths (i.e., slew, delay, arrival time, etc.) earlier, which can potentially reduce a significant amount of computation efforts on frontier propagation and thus speed up the entire process. Nonetheless, the duration of being spare is in fact non-deterministic due to the unpredictable network traffic. The degree of being speculative must be carefully restrained to prevent runtime from being overwhelmed by speculative works. A viable solution is to iteratively inspect the event base by the time speculative propagation starts. If an active event exists, the speculative propagation ceases and returns the program back to the event handler. Otherwise, we perform speculative propagation for only one level and repeat the same procedure for the next iteration.

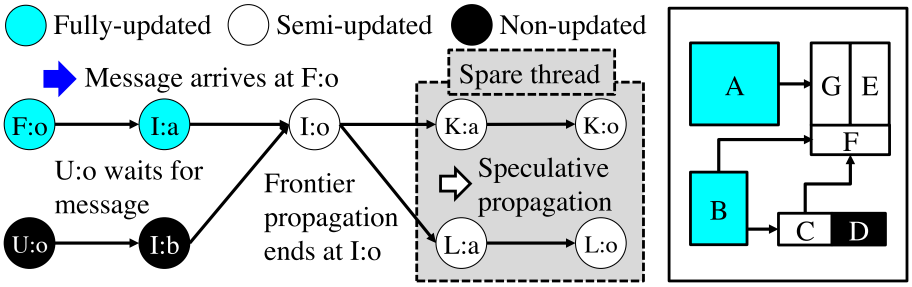


Figure 4.8: Spare thread performs speculative propagation in order to gain advanced saving of frontier work.

4.5 Experimental Results

Our program is implemented in C++ language on a 64-bit Linux operating system. We use POSIX socket library and `libevent` package for our event-driven network programming [35], and our messaging interface is built upon flexible `protocol buffer` [34]. Evaluation is taken on a computer cluster which has over 500 compute nodes. Each compute node is configured with 16 Intel 2.60 GHz cores and 64 GB RAM. The network infrastructure uses 384-port Mellanox MSX6518-NR FDR InfiniBand with gigabit ethernet control network and the disk system was configured to GPFS. Accessing to the compute nodes for running a program is done via a script submission to the network cluster manager which is designed based on the Torque resource manager with the Moab workload manager for running distributed jobs [37].

4.5.1 Benchmark Suite

We evaluate our framework based on a set of realistic benchmarks, including open-source designs used in recent timing community [9] and large hierarchical designs generated by an industry standard timer. The benchmark statistics are summarized in Table 4.1. These design statistics are reported from a flat point of view. All benchmarks are million-scale circuits in terms of the size of the timing graph. Each benchmark consists of several partitions and one top-level graph that hooks up the entire design. Initial timing assertions are applied to the source ports of the top-level graph.

4.5.2 Performance

The overall performance of our framework is listed in Table 4.1. In order to alleviate the uncertainty of network delay, we present for each design the average values on ten runs of complete timing analysis (arrival time and required arrival time propagations, endpoint slack calculation, etc.). It can be seen that the our framework is highly efficient and effective in terms of runtime values. For instance, it uses less than a half hour to reach the goal on large designs such as DesignB, DesignC, and DesignD. The result can scale to hundreds of partitions (see DesignA). We observed the non-blocking event-driven feature of our framework achieves effective overlap of communication

Table 4.1: Benchmark Statistics and Overall Performance of Our Framework

Circuit	$\|G\|$	$\|N\|$	$\|V\|$	$\|E\|$	$\|P\|$	L	W/o speculation				W/ speculation			
							cpu	mem	msg	usage	cpu	mem	msg	usage
DesignA	2.2M	1.1M	7.3M	12.4M	250	436	63s	1.6GB	0.7MB	17.3%	76s	1.7GB	1.6MB	64.2%
DesignB	14.5M	9.3M	39.0M	117.0M	37	3216	392s	2.9GB	2.0MB	9.1%	346s	3.1GB	5.7MB	73.1%
DesignC	23.3M	11.3M	76.9M	107.0M	30	2023	478s	4.7GB	2.3MB	19.5%	473s	4.8GB	8.1MB	57.8%
DesignD	42.7M	20.8M	128.1M	178.4M	50	5741	1239s	5.1GB	4.9MB	20.1%	1107s	5.1GB	9.7MB	69.4%

$|G|$: # of gates. $|N|$: # of nets. $|V|$: # of nodes. $|E|$: # of edges. $|P|$: # of partitions. L: # of levels. cpu: runtime. mem: peak memory on a program. msg: amount of message passing. usage: avg cpu utilization on a program.

and computation, and quick response to message update. From the memory perspective, the peak usage for a single program is only about 5 GB in DesignD.

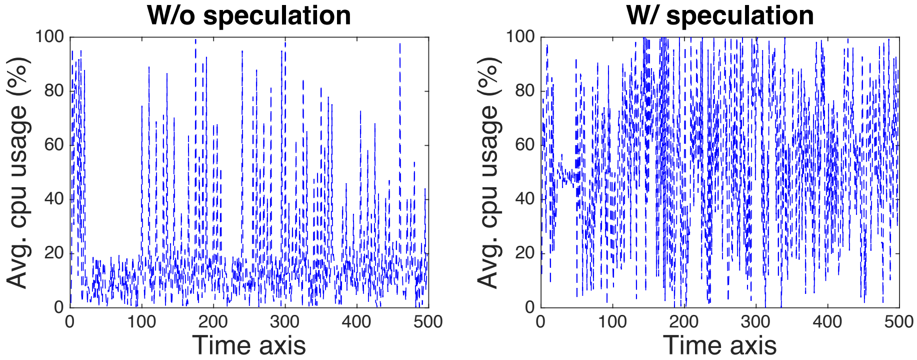


Figure 4.9: Average CPU utilization over time across all machines.

We next discuss the performance difference between implementations with and without speculative propagation. While the effectiveness of speculative propagation highly depends on network traffic and the graph topology, it can be seen in DesignB the total runtime is speeded up by 11.2% compared to the non-speculative counterpart. Being speculative is in particular beneficial for design with long chain of partition dependencies, which can be implicitly reflected on the number of levels in the graph. As a result, higher utilization of thread also translates into increased CPU usage, as shown in Figure 4.9.

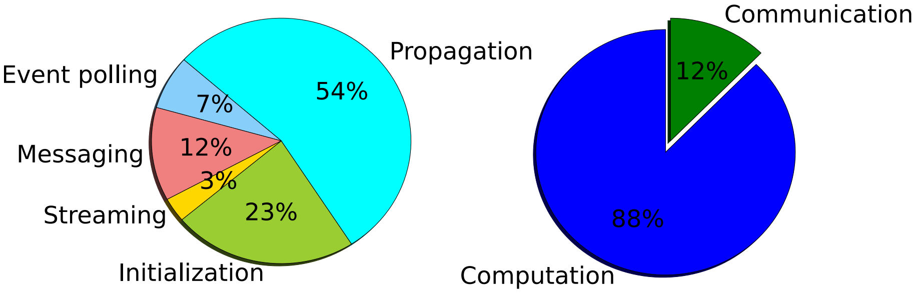


Figure 4.10: Runtime profile of our framework.

An in-depth view of the runtime profile is illustrated in Figure 4.10. It is expected that timing propagation consumes the majority of the runtime by about 54.4%. Initialization (data loading and client-pin mapping), event polling on non-blocking socket IO, and data streaming (serialization and de-serialization) take about 23.0%, 7.1%, and 3.2%, respectively. The time spent on message passing, which in fact is hardware-dependent, occupies approximately 12.3%. In a rough overview, the ratio of computation to communication is about 87.7% to 12.3%.

4.6 Conclusion

In this chapter, we have presented a distributed timing analysis framework for large designs. Our framework is built around five elements: general design partitions in distributed file systems, multiple-programs multiple-data programming paradigm, non-blocking socket IO, event-driven environment, and flexible messaging interface. These elements together let our framework achieve high scalability, quick response to message update, and effective overlap of communication and computation. We have developed algorithms for distributed timing as well as generic propagation schemes on the top of our framework and evaluated the performance on industry designs with millions of gates and hundreds of hierarchical partitions.

CHAPTER 5

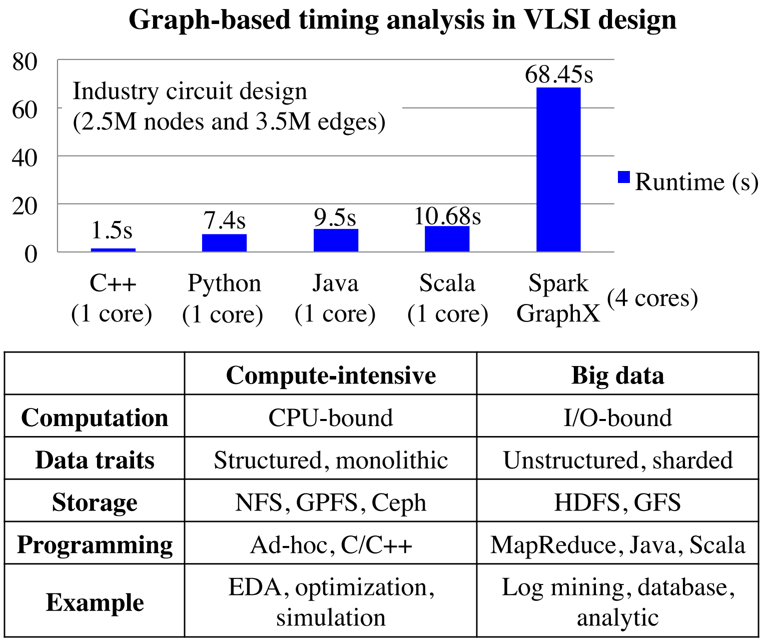
DTCRAFT: AN OPEN-SOURCE DISTRIBUTED EXECUTION ENGINE FOR COMPUTE-INTENSIVE APPLICATIONS

5.1 Introduction

Electronic design automation (EDA) has been an immensely successful field in assisting designers in implementing very large scale integration (VLSI) circuits with billions of transistors. EDA was on the forefront of computing (around 1980) and has fostered many of the largest computational problems such as graph theory and mathematical optimizations. As the design complexity continues to increase, the recent industry is seeking novel platform innovation that offers agile programming environment together with massively-parallel integration to leverage numerous computations of circuit designs [38]. While similar studies have been extensively made in big data-focused challenges over the past few years, EDA experts remain unclear about how these provenly effective techniques can be extended to silicon domain in a systematic and scalable manner. Nevertheless, the research counterparts for EDA-inspired engineering including large-scale optimizations, modeling, and simulations are still nascent.

Recently, cluster computing frameworks such as MapReduce, Spark, and Dryad have been widely used for big data processing [30, 29, 39, 40, 33]. The availability of allowing users without any experience of distributed systems to develop applications that access large cluster resources has demonstrated great success in many big data analytics. Existing platforms, however, mainly focus on big data processing. Research for high-performance or *compute-driven* counterparts such as large-scale optimizations and engineering simulations has failed to garner the same attention. As horizontal scaling has proven to be the most cost-efficient way to increase compute capacity, the need to efficiently leverage numerous computations is quickly becoming the next challenge [38, 41].

Compute-intensive applications have many different characteristics from big data. First, developers are obsessed about performance. Striving for high performance typically requires intensive CPU computations and efficient memory managements, while big data computing is more data-intensive and I/O-bound. Second, performance-critical data are more connected and structured than that of big data. Design files cannot be easily partitioned into independent pieces, making it difficult to fit into MapReduce paradigm [30]. Also, it is fair to claim most compute-driven data are *medium-size* as they must be kept in memory for performance purpose [38]. The benefit of MapReduce may not be fully utilized in this domain. Third, performance-optimized programs are normally hard-coded in C/C++, whereas the mainstream big data languages are Java, Scala, and Python. Rewriting these ad-hoc programs that have been robustly present in the tool chain for decades



	Compute-intensive	Big data
Computation	CPU-bound	I/O-bound
Data traits	Structured, monolithic	Unstructured, sharded
Storage	NFS, GPFS, Ceph	HDFS, GFS
Programming	Ad-hoc, C/C++	MapReduce, Java, Scala
Example	EDA, optimization, simulation	Log mining, database, analytic

Figure 5.1: An example of VLSI timing analysis and the comparison between compute-intensive applications and big data [36, 42].

To prove the concept, a recent research study has reported an experiment comparing the performance of running VLSI timing analysis under different languages and system frameworks [42]. As shown in Figure 5.1, the handcrafted C/C++ program is much faster than many of mainstream big data languages such as Python, Java, and Scala. It can even outperform one of the best big data cluster computing frameworks, the distributed Spark/GraphX-based implementation, by 45× faster. Many industry experts have realized that big data are not an easy fit to their domains, for example, semiconductor design optimizations and engineering simulations. Unfortunately, the ever-increasing design complexity will far exceed what many old ad-hoc methods have been able to accomplish. In addition to having researchers and practitioners acquire new domain knowledge, we must rethink the approaches of developing software to enable the proliferation of new algorithms combined with readily reusable toolboxes. To this end, the key challenge is to discover an elastic programming paradigm that lets developers place computations at customizable granularity wherever the data are – which is believed to deliver the next leap of engineering productivity and unleash new business model opportunities [38].

One of the main challenges to achieve this goal is to define a suitable programming model that abstracts the data computation and process communication effectively. The success of big data analytics in allowing users without any experience of distributed computing to easily deploy jobs that access large cluster resources is a key inspiration to our system design [30, 39, 40, 33]. We are also motivated by the fact that existing big data systems such as Hadoop and Spark are facing the bottleneck in support for compute-optimized codes and general dataflow programming [41]. For many compute-driven or resource-intensive problems, the most effective way to achieve scalable performance is to force developers to exploit the parallelism. Prior efforts have been made to either breaking data dependencies based on domain-specific knowledge of physical traits or discovering independent components across multiple application hierarchies [42]. Our primary focus is instead on the generality of a programming model and, more importantly, the simplicity and efficiency of building distributed applications on top of our system.

While this project was initially launched to address a question from our industry partners, "*How can we leverage the numerous computations of semiconductor designs to improve the engineering productivity?*", our design philosophy is a general system that is useful for compute-intensive applications such as graph algorithms and machine learning. As a consequence, we introduce in this chapter *DtCraft*, a general-purpose distributed execution engine for building high-performance parallel applications. DtCraft is built on Linux machines with modern C++17, enabling end users to utilize the robust C++

standard library along with our parallel framework. A DtCraft application is described in the form of a *stream graph*, in which vertices and edges are associated with each other to represent generic computations and real-time data streams. Given an application in this framework, the DtCraft runtime automatically takes care of all concurrency controls including partitioning, scheduling, and work distribution over the cluster. Users do not need to worry about system details and can focus on high-level development toward appropriate granularity. We summarize three major contributions of DtCraft as follows:

- **New programming paradigm.** We introduce a powerful and flexible new programming model for building distributed applications from sequential stream graphs. Our programming model is very simple yet general enough to support generic dataflow including feedback loops, persistent jobs, and real-time streaming. Stream graph components are highly customizable with meta-programming. Data can exist in arbitrary forms, and computations are autonomously invoked wherever data are available. Compared to existing cluster computing systems, our framework is more elastic in gaining scalable performance.

- **Software-defined infrastructure.** Our system enables fine-grained resource controls by leveraging modern OS container technologies. Applications live inside secure and robust Linux containers as work units which aggregate the application code with runtime dependencies on different OS distributions. With a container layer of resource management, users can tailor their application runtime toward tremendous performance gain.

- **Unified framework.** We introduce the first integration of user-space dataflow programming with resource container. For this purpose, many network programming components are re-devised to fuse with our system architecture. The unified framework empowers users to utilize rich APIs of our system to build highly optimized applications.

We believe DtCraft stands out as a unique system considering the ensemble of software tradeoffs and architecture decisions we have made. With these features, DtCraft is suited for various applications both on systems that search for transparent concurrency to run compute-optimized codes, and on those that prefer distributed integration of existing developments with vast expanse of legacy codes in order to bridge the performance gap. We have evaluated DtCraft on micro-benchmarks including machine learning, graph algorithms, and large-scale semiconductor engineering problems. We have shown DtCraft outperforms one of the best cluster computing systems in big data community by more than an order of magnitude. Also, we have demonstrated DtCraft can be applied to wider domains that are known difficult to fit into existing big data ecosystems.

5.2 The DtCraft System

The overview of the DtCraft system architecture is shown in Figure 5.2. The system kernel contains a *master* daemon that manages *agent* daemons running on each cluster node. Each job is coordinated by an *executor* process that is either invoked upon job submission or launched on an agent node to run the tasks. A job or an application is described in a stream graph formulation. Users can specify resource requirements (e.g. CPU, memory, disk usage) and define computation callbacks for each vertex and edge, while the whole detailed concurrency controls and data transfers are automatically operated by the system kernel. A job is submitted to the cluster via a script that sets up the environment variables and the executable path with arguments passed to its `main` method. When a new job is submitted to the master, the scheduler partitions the graph into several *topologies* depending on current hardware resources and CPU loads. Each topology is then sent to the corresponding agent and is executed in an executor process forked by the agent. For those edges within the same topology, data are exchanged via efficient shared memory. Edges between different topologies are communicated through TCP sockets. Stream overflow is resolved by per-process key-value store, and users are perceived with virtually infinite data sets without deadlock.

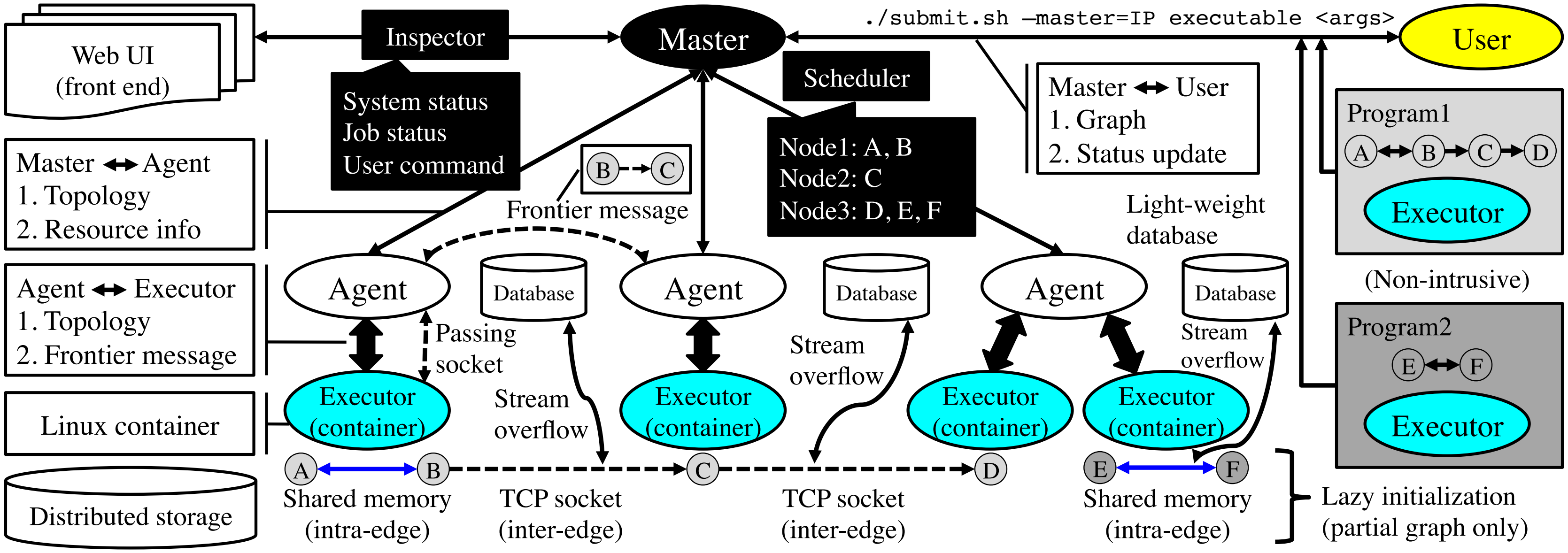


Figure 5.2: The system architecture of DtCraft. The kernel consists of a master daemon and one agent daemon per working machine. User describes an application in terms of a sequential stream graph and submits the executable to the master through our submission script. The kernel automatically deals with concurrency controls including scheduling, process communication, and work distribution that are known difficult to program correctly. Data is transfered through either TCP socket streams on inter-edges or shared memory on intra-edges, depending on the deployment by the scheduler. Application and workload are isolated in secure and robust Linux containers.

5.2.1 Stream Graph Programming Model

DtCraft is strongly tight to modern C++ features, in particular the concurrency libraries, lambda functions, and templates. We have struck a balance between the ease of the programmability at user level and the modularity of the underlying system that needs to be extensible with the advance of software technology. The main programming interface including gateway classes is sketched as follows:

```
class Vertex {
  function<void()> on;
  once_flag flag;
  Adjacency<DeviceWriter> writers;  // weak pointers
  Adjacency<DeviceReader> readers;  // weak pointers
};

class Stream {
  weak_ptr<DeviceWriter> writer;
  weak_ptr<DeviceReader> reader;
  function<Signal(Vertex&, DeviceWriter&)> on_os();
  function<Signal(Vertex&, DeviceReader&)> on_is();
};

class Graph {
  template <typename C> // vertex
  auto insert(C&&...);

  template <typename O, typename I> // stream
  auto insert(const auto&, O&&, const auto&, I&&)

  template <typename... U>
  auto containerize(U&&...);
};

class Executor : Reactor {
  Executor(Graph&);
};
```

Programmers formulate an application into a stream graph and define computation callbacks in the format of standard function object for each vertex and edge (stream). Vertices and edges are highly customizable subject to the inheritance from classes `Vertex` and `Stream` that interact with our back-end. The vertex callback is a constructor-like call-once barrier that is used to synchronize all adjacent edge streams at the beginning. Each edge is associated with two callbacks, one for output stream at the tail vertex and another one for input stream at the head vertex. Our stream interface follows the structure of standard C++ `iostream` library. We have developed specialized *stream buffer* classes in charge of performing reading and writing operations on stream objects. The stream buffer class hides from users a great deal of work such as non-blocking communication, stream overflow and synchronization, and error handling. Vertices and edges are explicitly connected together through the `Graph` and its method `insert`. Users can configure the resource requirements for different portions of the graph using our container method `containerize`. Finally, an executor class forms the graph along with application-specific parameters into a simple closure and dispatches it to the remote master for execution.

5.2.2 A Concurrent Ping-Pong Example

To understand our programming interface, we describe a concrete example of a DtCraft application. The example we have chosen is a representative class in many software libraries – *concurrent ping-pong*, as it represents a fundamental building block of many iterative or incremental algorithms. The flow diagram of a concurrent ping-pong and its runtime on our system are illustrated in Figure 5.3. The ping-pong consists of two vertices, called "*Ball*", which asynchronously sends a random binary character to each other, and two edges that are used to capture the data streams. Iteration stops when the internal counter of a vertex reaches a given threshold.

```
auto Ball(Vertex& v, auto& k) {
  v.writers.at(k).lock->ostream((rand()%2));
  return Stream::DEFAULT;
};
auto PingPong(auto& v, auto& r, auto& k, auto& c) {
```

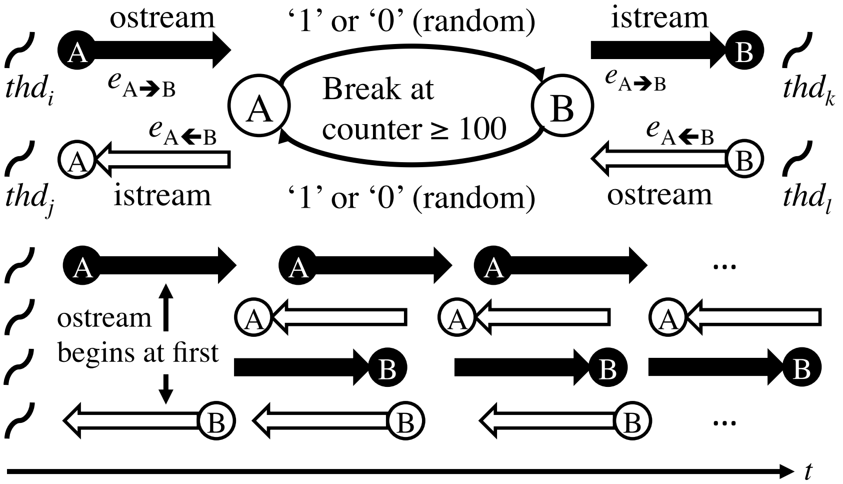


Figure 5.3: Flow diagram of the concurrent ping-pong example. Computation callbacks on streams are simultaneously invoked by multiple threads.

```
  int data;
  reader.istream(data);
  if((c+=data) >= 100) return Stream::REMOVE_THIS;
  return Ball(v, k)
}

Graph G;
key_type AB, BA;
auto count_A {0}, count_B {0};
auto A = G.insert([&](auto& v){ Ball(v, AB); })
auto B = G.insert([&](auto& v){ Ball(v, BA); })
AB = G.insert(
  A, [&](auto& v, auto& writer) {}, // ostream
  B, [&](auto& v, auto& reader) {   // istream
    return PingPong(v, reader, BA, count_B);
  }
);
BA = G.insert(
  B, [&](auto& v, auto& writer) {}, // ostream
  A, [&](auto& v, auto& reader) {   // istream
    return PingPong(v, reader, AB, count_A)
  }
```

```
);
G.containerize(A, "memory=1KB", "num_cpus=1");
G.containerize(B, "memory=1KB", "num_cpus=1");
Executor(G).dispatch();
```

As presented in the above code snippet, we define a function `Ball` that writes a binary data through the stream `k` on vertex `v`. We define another function `PingPong` to retrieve the data arriving in vertex `v` followed by `Ball` if the counter has not reached the threshold. We next define vertices and streams using the class method `insert` from the graph, as well as their callbacks based on `Ball` and `PingPong`. The vertex first reaching the threshold will close the underlying stream channels via a return of `Stream::REMOVE_THIS`. This is a handy feature of our system. Users do not need to invoke extra function call to signal our stream back-end. Closing one end of a stream will subsequently force the other end to be closed, which in turn updates the stream ownership on corresponding vertices. We configure each vertex with 1 KB memory and 1 CPU. Finally, an executor instance is created to wrap the graph into a closure and dispatch it to the remote master for execution.

5.2.3 Advantages of the Proposed Model

DtCraft provides a programming interface similar to those found in C++ standard libraries. Users can learn how to develop a DtCraft application at a faster pace. The same code that executes distributively can be also deployed on a local machine for debugging purpose. No programming changes are necessary except the options passed to the submission script. Note that our framework needs only a *single entity* of executable from users. The system kernel is not intrusive to any user-defined entries, for instance, the arguments passed to the `main` method. We encourage users to describe stream graphs with C++ lambda and function objects. This functional programming style provides a very powerful abstraction that allows the runtime to bind callable objects and captures different runtime states.

Although conventional dataflow thinks applications as "computation vertices" and "dependency edges" [39, 40, 43, 44], our system model does not impose explicit boundary (e.g., DAG restriction). As shown in previous code snippets, vertices and edges are logically associated with each other and are combined to represent generic stream computations including feedback controls, state machines, and asynchronous streaming. Stream computations are by default long-lived and persist in memory until the end-of-file state is lifted. In other words, our programming interface enables straightforward *in-memory* computing, which is an important factor for iterative and incremental algorithms. This feature is different from existing data-driven cluster computing frameworks such as Dryad, Hadoop, and Spark that rely on either frequent disk access or expensive extra caching for data reuse [30, 40, 33]. In addition, our system model facilitates the design of real-time streaming engines. A powerful streaming engine has the potential to bridge the performance gap caused by application boundaries or design hierarchies. It is worth noting that many engineering applications and companies existed "*pre-cloud*", and the most techniques they applied were ad-hoc C/C++ [38]. To improve the engineering turnaround, our system can be explored as a distributed integration of existing developments with legacy codes.

Another powerful feature of our system over existing frameworks is *guided scheduling* using Linux containers. Users can specify *hard* or *soft* constraints configuring the set of Linux containers on which application pieces would like to run. The scheduler can preferentially select the set of computers to launch application containers for better resource sharing and data locality. While transparent resource control is successful in many data-driven cluster computing systems, we have shown that compute-intensive applications has distinctive computation patterns and resource management models. With this feature, users can implement diverse approaches to various problems in the cluster at any granularity. In fact, we are convinced by our industry partners that the capability of explicit resource controls is extremely beneficial for domain experts to optimize the runtime of performance-critical routines. Our container interface also offers users secure and robust runtime, in which different application pieces are isolated in independent Linux instances. To our best knowledge, DtCraft is the first districuted execution engine that incorporates the Linux container into dataflow programming.

In summary, each system has its own merits in certain application domain, and it is impossible to provide thorough comparison with prior works due to the page limit. However, we believe DtCraft stands out as a unique system given the following attributes: (1) a compute-driven distributed system completely designed from modern C++17; (2) a new asynchronous stream-based programming model in support for general dataflow; (3) a container layer integrated with user-space programming to enable fine-grained resource controls and performance tunning. Developers are encouraged to investigate the structure of their applications and the properties of proprietary systems. Careful graph construction and refinement can improve the performance substantially.

While there are many benefits about DtCraft, we indeed compromise a few complexities on code verbosity and data managements compared to MapReduced-based cluster computing systems. Nevertheless, DtCraft is notable for problem domains that rely on dataflow controls to optimize the computation performance. Developers are encouraged to investigate the structure of their applications and the properties of proprietary systems. Careful graph construction and refinement can improve the performance substantially.

5.3 System Implementation

DtCraft aims to provide a unified framework that works seamlessly with the C++ standard library. Like many distributed systems, network programming is an integral part of our system kernel. While our initial plan was to adopt third-party libraries, we have found considerable incompatibility with our system architecture (discussed in later sections). Fixing them would require extensive rewrites of library core components. Thus, we decided to re-design these network programming components from ground-up, in particular the event library and serialization interface that are fundamental to DtCraft. We shall also discuss how we achieve distributed execution of a given graph, including scheduling and transparent communication.

5.3.1 Event-Driven Environment

DtCraft supports event-based programming style to gain benefits from asynchronous computations. Writing an event reactor has traditionally been the domain of experts and the language they obsessed about is C [35]. The biggest issue we found in widely used event libraries is the inefficient support for object-oriented design and modern concurrency. Our goal is thus to incorporate the power of C++ libraries with low-level system controls such as non-blocking mechanism and I/O polling. Due to the space limit, we present only the key design principles of our event reactor as follows:

```
class Event : enable_shared_from_this <Event> {
  enum Type {
    TIMEOUT,
    PERIODIC,
    READ,
    WRITE
  };
  const function<Signal(Event&)> on;
};

class Reactor {
  Threadpool threadpool;
  unordered_set<shared_ptr<Event>> eventset;

  template <typename T, typename... U>
  future<shared_ptr<T>> make(U&&... u) {
    auto e = make_shared<T>(forward<U>(u)...);
    return promise([&, e=move(e)](){
      _insert(e);  // insert an event into reactor
      return e;
    });
  }
};
```

Unlike existing libraries, our event is a *flattened* unit of operations including timeout and I/O. Events can be customized given the inheritance from class `Event`. The event callback is defined in a *function* object that can work closely with lambda and polymorphic function wrappers. Each event instance is created by the reactor and is only accessible through C++ *smart pointer* with shared ownership among those inside the callback scope. This gives us a number of benefits such as precise polymorphic memory managements and avoidance of ABA problems that are typically hard to achieve with raw pointers. We have implemented the reactor using *task-based* parallelism. A significant problem of existing libraries is the condition handling in multi-threaded environment. For example, a thread calling to insert or remove an event can get a nonsense return if the main thread is too busy to handle the request [35, 45]. To enable proper concurrency controls, we have adopted C++ *future* and *promise* objects to separate the acts between the provider (reactor) and consumers (threads). Multiple threads can thus safely create or remove events in arbitrary orders. In fact, our unit test has shown 4–12× improvements in throughput and latencies over existing libraries [35, 45].

5.3.2 Serialization and Deserialization

We have built a dedicated serialization and deserialization layer called *archiver* on top of our stream interface. The archiver has been intensively used in our system kernel communication. Users are strongly encouraged, though not necessary, to wrap their data with our archiver as it is highly optimized to our stream interface. Our archiver is similar to the modern *template-based* library `Cereal`, where data types can be reversibly transformed into different representations such as binary encodings, JSON, and XML [46]. However, the problem we discovered in `Cereal` is the lack of proper size controls during serialization and deserialization. This can easily cause exception or crash when non-blocking stream resources become *partially* unavailable. While extracting the size information in advance requires twofold processing, we have found such burden can be effectively leveraged using modern C++ template techniques. A code example of our binary archiver is given as follows:

```
class BinaryOutputArchiver {
  ostream& os;
  template <typename... U>
  constexpr streamsize operator()(U&&... u) {
    return archive(forward<U>(u)...);
  }
};
```

We developed our archiver based on extensive templates to enable a unified API. Many operations on stack-based objects and constant values are prescribed at compile time using constant expression and forwarding reference techniques. The archiver is a lightweight layer that performs serialization and deserialization of user-specified data members directly on the stream object passed to the callback. We also offer a packager interface that wraps data with a size tag for complete message processing. Both archiver and packager are defined as *callable* objects to facilitate dynamic scoping in our multi-threaded environment.

5.3.3 Input and Output Streams

One of the challenges in designing our system is choosing an abstraction for data processing. We have examined various options and concluded that developing a dedicated stream interface is necessary to provide users a simple but robust layer of I/O services. To facilitate the integration of safe and portable streaming execution, our stream interface follows the idea of C++ `istream` and `ostream`. Users are perceived with the API similar to those found in C++ standard library, while our *stream buffer* back-end implements the entire details such as device synchronization and low-level non-blocking data transfers.

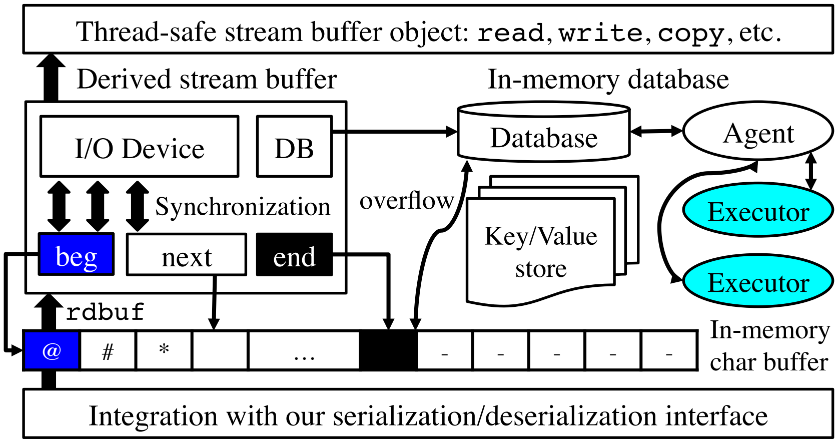


Figure 5.4: DtCraft provides a dedicated stream buffer object in control of reading and writing operations on devices.

Figure 5.4 illustrates the structure of a stream buffer object in our system kernel. A stream buffer object is a class similar to C++ `basic_streambuf` and consists of three components, *character sequence*, *device*, and *database pointer*. The character sequence is an in-memory linear buffer storing a particular window of the data stream. The device is an OS-level entity (e.g.

TCP socket, shared memory) that derives reading and writing methods from an interface class with static polymorphism. Our stream buffer is *thread safe* and is directly integrated with our serialization and deserialization methods. To properly handle the buffer overflow, each stream buffer object is associated with a raw pointer to a database owned by the process. The database is initiated when a master, an agent, or an executor is created, and is shared among all stream buffer objects involved in that process. Unless the ultimate disk usage is full, users are virtually perceived with unbounded stream capacity in no worry about the deadlock.

5.3.4 Kernel: Master, Agent, and Executor

Master, agent, and executor are the three major components in the system kernel. There are many factors that have led to the design of our system kernel. Overall regard is the reliability and efficiency in response to different message types. We have defined a reliable and extensible message structure of type *variant* to manipulate a heterogeneous set of message types in a uniform manner. Each message type has data members to be serialized and deserialized by our archiver. The top-level class can inherit from a *visitor* base with dynamic polymorphism and derive dedicated handlers for certain message types.

To efficiently react to each message, we have adopted the event-based programming style. Master, agent, and executor are persistent objects derived from our reactor with specialized events binding to each. While it is expectedly difficult to write non-sequential codes, we have found a number of benefits of adopting event-driven interface, for instance, asynchronous computations, natural task flow controls, and concurrency. We have defined several master events in charge of graph scheduling and status report. For agent, most events are designated as a proxy to monitor current machine status and fork an executor to launch tasks. Executor events are responsible for the communication with the master and agents as well as the encapsulation of asynchronous vertex and edge events. Multiple events are executed efficiently on a shared thread pool in our reactor.

Communication Channels

The communication channels between different components in DtCraft are listed in Table 5.1. By default, DtCraft supports three types of communication channels, *TCP socket* for network communication between remote hosts, *domain socket* for process communication on a local machine, and *shared memory* for in-process data exchange. For each of these three channels, we have implemented a unique device class that effectively supports non-blocking I/O and error handling. Individual device classes are pluggable to our stream buffer object and can be extended to incorporate device-specific attributes for further I/O optimizations.

Table 5.1: Communication Channels in DtCraft

Target	Protocol	Channel	Latency
Master–User	TCP socket	Network	High
Master–Agent	TCP socket	Network	High
Agent–Executor	Domain socket	Local processes	Medium
Intra-edge	Shared memory	Within a process	Low
Inter-edge	TCP socket	Network	High

Since master and agents are coordinated with each other in the distributed environment, the communication channels run through reliable TCP socket streams. We enable two types of communication channels for graphs, shared memory and TCP socket. As we shall see in the next section, the scheduler might partition a given graph into multiple topologies running on different agent nodes. Edges crossing the partition boundary are communicated through TCP sockets, while data within a topology is exchanged through shared memory with extremely low latency cost. To prevent our system kernel from being bottlenecked by data transfers, master and agents are only responsible for control decisions. All data are sent between vertices managed by the executor. Nevertheless, achieving point-to-point communication is non-trivial for inter-edges. The main reason is that the graph structure is offline unknown and our system has to be general to different communication patterns deployed by the scheduler. We have managed to solve this by means of file descriptor passing through environment variables. The agent exports a list of open file descriptors to an environment variable which will be in turn inherited by the corresponding executor under fork.

Application Container

DtCraft leverages existing OS container technologies to enable isolation of application resources from one another. Because these technologies are platform-dependent, we implemented a *pluggable isolation module* to support multiple isolation mechanisms. An isolation module containerizes a process based on user-specified attributes. By default, we apply the Linux *control groups* (cgroups) kernel feature to impose per-resource limits (CPU, memory, block I/O, and network) on user applications. With cgroups, we are able to consolidate many workloads on a single node while guaranteeing the quota assigned to each application. In order to achieve secure and robust runtime, our system runs applications in isolated *namespaces*. We currently support IPC, network, mount, PID, UTS, and user namespaces. By essentially separating processes into independent namespaces, user applications are ensured to be invisible from others and will be unable to make connections outside of the namespaces. External connections such as inter-edge streaming are managed by agents through device descriptor passing techniques. Our container implementation also supports process snapshots, which is beneficial for checkpointing and live migration.

Graph Scheduling

Scheduler is an asynchronous master event that is invoked when a new graph arrives. Given a user-submitted graph, the goal of the scheduler is to find a deployment of each vertex and each edge considering the machine loads and resource constraints. A graph might be partitioned into a set of *topologies* that can be accommodated by the present resources. A topology is the basic unit of a task (container) that is launched by an executor process on an agent node. A topology is not a graph because it may contain dangling edges along the partition boundary. Once the scheduler has decided the deployment, each topology is marshaled along with graph parameters including the UUID, resource requirements, and input arguments to form a closure that can be sent to the corresponding agent for execution. An example of the scheduling process is shown in Figure 5.5. At present, two schedulers persist in our system, a *global scheduler* invoked by the master and a *local scheduler* managed by the agent. Given user-configured containers, the global scheduler performs resource-aware partition based on the assumption that the graph must be completely deployed at one time. The global scheduling problem is formulated into a bin packing optimization where we additionally take into account the number of edge cuts to reduce the latency. An application is rejected by the global scheduler if its mandatory resources (must acquire in order to run) exceed the maximum capability of machines. As a graph can be partitioned into different topologies, the goal of the local scheduler is to synchronize all inter-edge connections of a topology and dispatch it to an executor. The local scheduler is also responsible for various container setups including resource update, namespace isolation, and fault recovery.

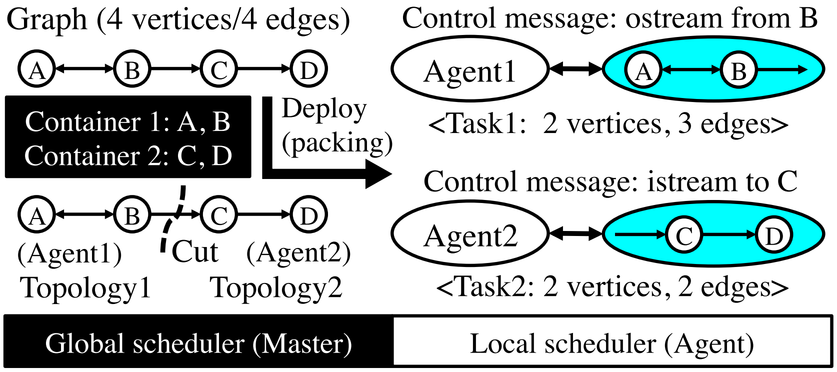

Graph (4 vertices/4 edges)
A
B
C
D
Container 1: A, B
Container 2: C, D
Deploy (packing)
A
B
C
D
(Agent1)
Cut
(Agent2)
Topology1
Topology2
Control message: ostream from B
Agent1
A
B
<Task1: 2 vertices, 3 edges>
Control message: istream to C
Agent2
C
D
<Task2: 2 vertices, 2 edges>
Global scheduler (Master)
Local scheduler (Agent)

Figure 5.5: An application is partitioned into a set of topologies by the global scheduler, which are in turn sent to remote agents (local scheduler) for execution.

Although our scheduler does not force users to explicitly containerize applications (resort to our default heuristics), empowering users fine-grained controls over resources can guide the scheduler toward tremendous performance gain. Due to the space limitation, we are unable to discuss the entire details of our schedulers. We believe developing a scheduler for distributed dataflow under multiple resource constraints deserves independent research effort. As a result, DtCraft delegates the scheduler implementation to a pluggable module that can be customized by organizations for their purposes.

Topology Execution

When the agent accepts a new topology, a special asynchronous event, *topology manager*, is created to take over the task. The topology manager spawns (fork-exec) a new executor process based on the parameters extracted from the topology, and coordinates with the executor until the task is finished. Because our kernel requires only a single entity of executable, the executor is notified by which execution mode to run via environment variables. In our case, the topology manager exports a variable to "*distributed*", as opposed to aforementioned "*submit*" where the executor submits the graph to the master. Once the process controls are finished, the topology manager delivers the topology to the executor. A set of executor events is subsequently triggered to launch asynchronous vertex and edge events.

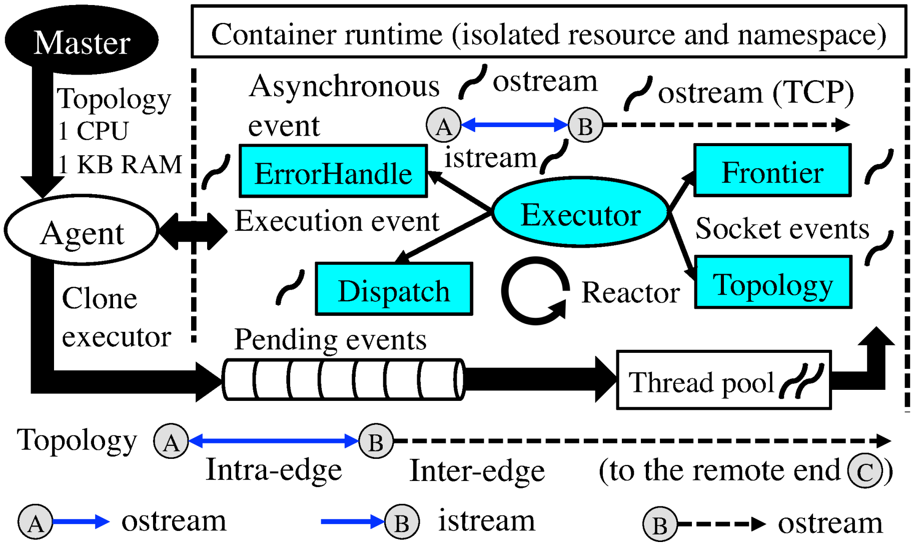


Figure 5.6: A snapshot of the executor runtime in distributed mode.

A snapshot of the executor runtime upon receiving a topology is shown in Figure 5.6. Roughly speaking, the executor performs two tasks. First, the executor initializes the graph from the given topology which contains a key set to describe the graph fragment. Since every executor resides in the same executable, an intuitive method is to initialize the whole graph as a parent reference to the topology. However, this can be cost-inefficient especially when vertices and edges have expensive constructors. To achieve a generally effective solution, we have applied lazy lambda technique to suspend the initialization (see the code below). The suspended lambda captures all required parameters to construct a vertex or an edge, and is lazily invoked by the executor runtime. By referring to a topology passed from the "future", only necessary vertices and edges will be constructed. The meaning of future means the runtime scheduling decided by the master. The executor on the distributed mode will not know the exact topology until the agent sends it to the corresponding executor.

```
template <typename V, typename... U>
VertexDescriptor Graph::insert_vertex(U&&... u) {
  auto key = generate_key();  // deterministic key
  tasks.emplace_back(  // lazy initialization
    [u..., key](Topology* t) {
      // local mode and distributed mode
      if(t == nullptr || t->has_key(key)) {
        auto v = make_shared<V>(u...);
        pm.set_value(move(v));
      }
      else t->insert(key);  // submit mode
    }
  );
  return key;
}
```

The second task is to initiate a set of events for vertex and edge callbacks. We have implemented an I/O event for each device based on our stream buffer object. Because each vertex callback is invoked only once, it can be absorbed into any adjacent edge events coordinated by modern C++ threading `once_flag` and `call_once`. Given an initialized graph, the executor iterates over every edge and creates shared memory I/O events and TCP socket I/O events for intra-edges and inter-edges, respectively. Notice that the device descriptor for inter-edges are fetched from the environment variables inherited from the agent.

5.4 Fault Tolerance Policy

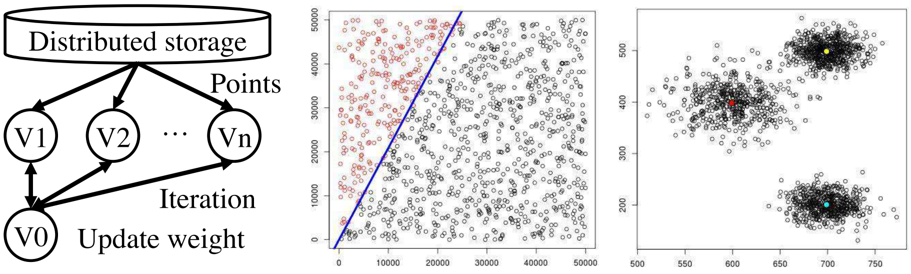
Our system architecture facilitates the design of fault tolerance on two fronts. First, master maintains a centralized mapping between active applications and agents. Every single error, which could be either heartbeat timeout on the executor or unexpected I/O behaviors on the communication channels, can be properly propagated. In case of a failure, the scheduler performs a linear search to terminate and re-deploy the application. Second, our container implementation can be easily extended to support periodic checkpointing. Executors are freezed to a stable state and are thawed after the checkpointing. The solution might not be perfect, but adding this functionality is already an advantage over our system framework, where all data transfers are exposed to our stream buffer interface and can be dumped without lost. However, depending on application properties and cluster environment, periodic checkpointing can be very time-consuming. For instance, many incremental optimization procedures have sophisticated memory dumps whereas the subsequent change between process maps are small. Restarting applications from the beginning might be faster than checkpoint-based fault recovery. Therefore, DtCraft leaves the decision of checkpointing to users. Users can configure this feature on a *per-application* basis to discover the performance tradeoff on proprietary systems.

5.5 Experimental Results

We have implemented DtCraft in C++17 on a Linux machine with GCC 7. Given the huge amount of existing cluster computing frameworks, we are unable to conduct comprehensive comparison subject to the space limit. Instead, we compare with one of the best cluster computing engines, Apache Spark 2.0 [33], that has been extensively studied by other research works as baseline. To further investigate the benefit of DtCraft, we compared with an application hand-crafted with domain-specific optimizations [42]. The performance of DtCraft is evaluated on three sets of experiments. The first two experiments took classic algorithms from machine learning and graph applications and compared the performance of DtCraft with Spark. We have analyzed the runtime performance over different numbers of machines on an academic cluster [24]. The third experiment applied DtCraft to solve a large-scale semiconductor design problem. Our goal is to explore DtCraft as a distributed solution to mitigate the end-to-end engineering efforts along the design flow. The evaluation has been undertaken on a large cluster in Amazon EC2 cloud [47]. Overall, we have shown the performance and scalability of DtCraft on both standalone applications and cross-domain applications that have been coupled together in a distributed manner.

5.5.1 Machine Learning

We implemented two iterative machine learning algorithms, logistic regression and k-means clustering, and compared our performance with Spark. One key difference between the two applications is the amount of computation they performed per byte of data. The iteration time of k-means is dominated by computations, whereas logistic regression is less compute-intensive [33]. The source codes we used to run on Spark are cloned from the official repository of Spark. For the sake of fairness, the DtCraft counterparts are implemented based on the algorithms of these Spark codes. Figure 5.7 shows the stream graph of logistic regression and k-means clustering in DtCraft, and two sample results that are consistent with Spark's solutions.



(a) Stream graph (b) Logistic regression (c) *k*-means

Figure 5.7: Stream graph to represent logistic regression and k-means jobs in DtCraft.

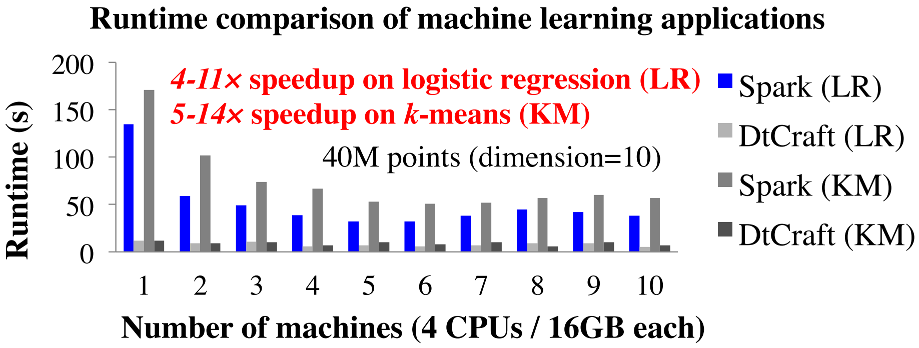


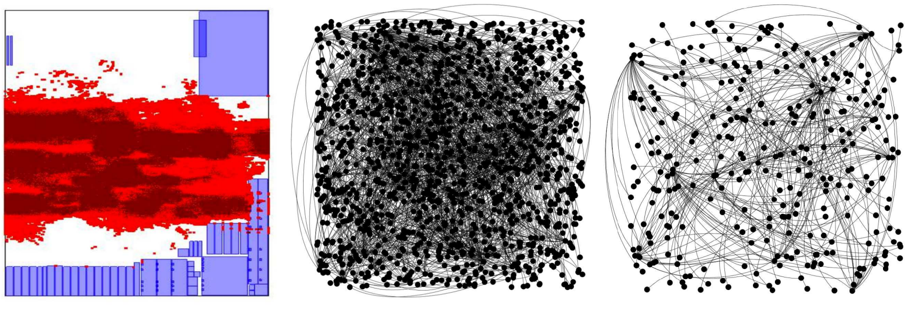
Figure 5.8: Runtimes of DtCraft versus Spark on logistic regression and k-means.

Figure 5.8 shows the runtime performance of DtCraft versus Spark. Unless otherwise noted, the value pair enclosed by the parenthesis (CPUs/GB) denotes the number of cores and the memory size per machine in our cluster.

We ran both logistic regression and k-means for 10 iterations on 40M sample points. It can be observed that DtCraft outperformed Spark by 4–11× and 5–14× faster on logistic regression and k-means, respectively. Although Spark can mitigate the long runtime by increasing the cluster size, the performance gap to DtCraft is still remarkable (up to 8× on 10 machines). In terms of communication cost, we have found hundreds of Spark RDD partitions shuffling over the network. In order to avoid disk I/O overhead, Spark imposed a significant burden on the first iteration to cache data for reusing RDDs in the subsequent iterations. In contrast, our system architecture enables straightforward *in-memory* computing, incurring no extra overhead of caching data on any iterations. Also, our scheduler can effectively leverage the machine overloads along with network overhead for higher performance gain.

5.5.2 Graph Algorithm

We next examine the effectiveness of DtCraft by running a graph algorithm. Graph problems are challenging in concurrent programming due to the iterative, incremental, and irregular computing patterns. We considered the classic shortest path problem on a circuit graph with 10M nodes and 14M edges released by [42]. The visualization of the circuit is shown in Figure 5.9. We implemented the Pregel-style shortest path finding algorithm in DtCraft, and compared it with Spark-based Pregel variation downloaded from the official GraphX repository [44]. As gates are closely connected with each other to form compact signal paths, finding a shortest (delay-critical) path can exhibit a wild swing in the evaluation of a value [48, 38].



(a) Circuit (1.01mm^2) (b) Graph (3M gates) (c) A signal path

Figure 5.9: Visualization of our graph benchmark.

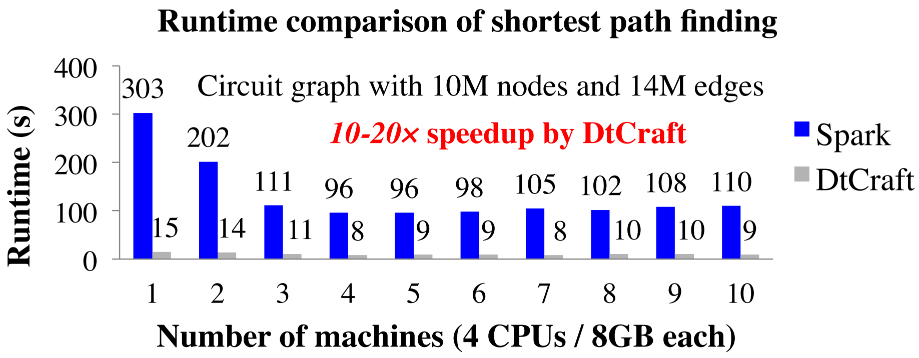


Figure 5.10: Runtimes of DtCraft versus Spark on finding a shortest path in our circuit graph benchmark.

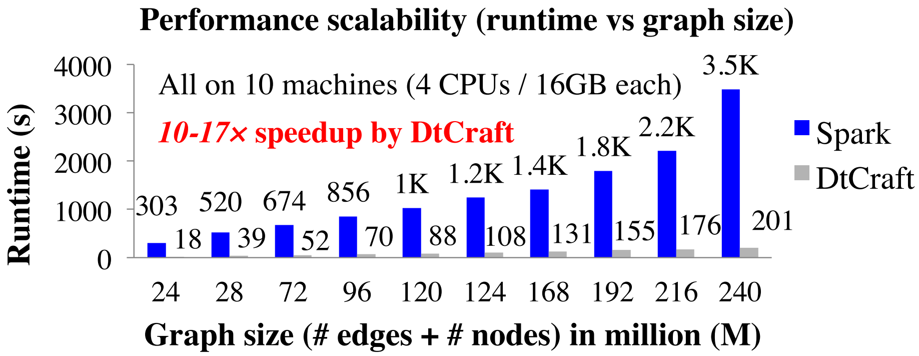


Figure 5.11: Runtime scalability of DtCraft versus Spark on different graph sizes.

Figure 5.10 shows the runtime comparison across different machine counts. In general, DtCraft reached the goal by 10–20× faster than Spark. Our program can finish all tests within a minute regardless of the machine usage. We have observed intensive network traffic among Spark RDD partitions whereas in our system most data transfers were effectively scheduled to shared memory. To further examine the runtime scalability, we duplicated the circuit graph and created random links to form larger graphs, and compared the runtimes of both systems on different graph sizes. As shown in Figure 5.11, the runtime curve of DtCraft is far scalable against Spark. The highest speedup is observed at the graph of size 240M, in which DtCraft is 17× faster than Spark. To summarize this micro-benchmarking, we believe the performance gap between Spark and DtCraft is due to the system architecture and language features we have chosen. While we compromise with users on explicit dataflow description, the performance gain in exchange can scale up to more than an order of magnitude over one of the best cluster computing systems.

5.5.3 Stochastic Simulation

We applied DtCraft to solve a large-scale stochastic simulation problem, Markov Chain Monte Carlo (MCMC) simulation. MCMC is a popular technique for estimating by simulation the expectation of a complex model. Despite notable success in domains such as astrophysics and cryptography, the practical widespread use of MCMC simulation had to await the invention of computers. The basis of an MCMC algorithm is the construction of a *transition kernel*, $p(x, y)$, that has an invariant density equal to the target density. Given a transition kernel (a conditional probability), the process can be started at an initial state x_0 to yield a draw x_1 from $p(x_0, x_1)$, x_2 from $p(x_1, x_2)$, ..., and $p(x_{S-1}, x_S)$, where S is the desired number of simulations. After a transient period, the distribution of x is approximately equal to the target distribution. The problem is the size of S can be made very large and the only restriction comes from computer time and capacity. To speed up the process while catching the accuracy, the recent industry is driving the need of distributed simulation [49].

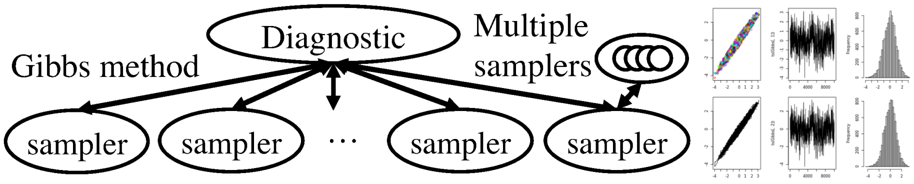


Figure 5.12: Stream graph (101 vertices and 200 edges) for distributed Markov Chain Monte Carlo simulation.

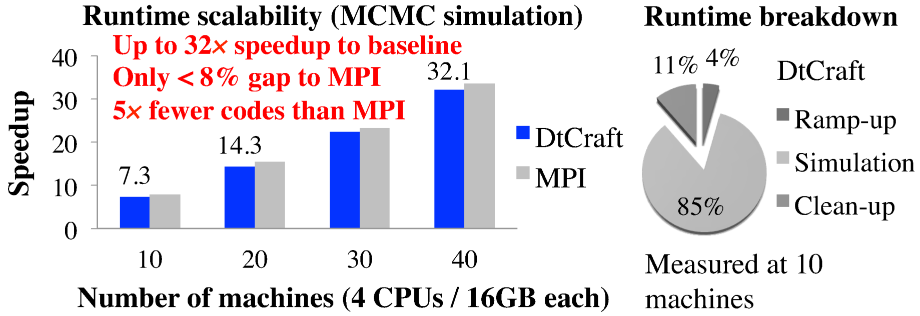


Figure 5.13: Runtime of DtCraft versus hard-coded MPI on MCMC simulation.

We consider Gibbs algorithm on 20 variables with 100000 iterations to obtain a final sample of 100000 [50]. The stream graph of our implementation is shown in Figure 5.12. Each Gibbs sampler represents an unique prior and will deliver the simulation result to the diagnostic vertex. The diagnostic vertex then performs statistical tests including outlier detection and convergence check. To measure our solution quality, we implemented a hard-coded C MPI program as the golden reference. As shown in Figure 5.13, the DtCraft-based solution achieved up to 32× seedup on 40 Amazon EC2 m4.xlarge machines over the baseline serial simulation, while keeping the performance margin within 8% to MPI. Nevertheless, it should be noted that our system enables many features such as transparent concurrency, application container, and fault tolerance, which MPI handles insufficiently. We have observed the majority of runtime is taken by simulation (85%) while ramp-up time (scheduling) and clean-up time (release containers, report to users) are 4% and 11%, respectively. This experiment justified DtCraft as an alternative to MPI, considering the tradeoff around performance, transparency, and programmability.

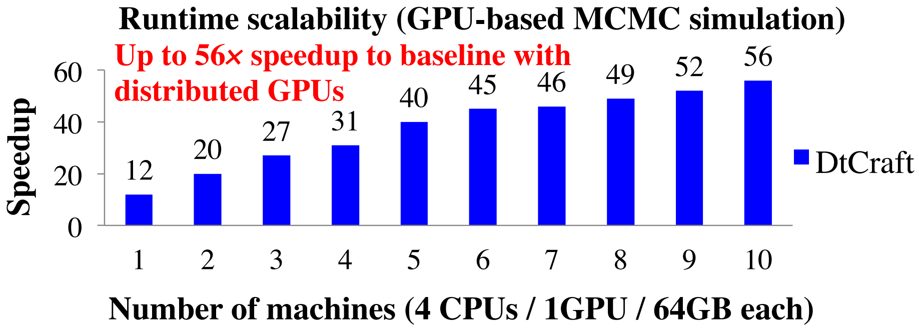


Figure 5.14: Accelerated MCMC simulation with distributed GPUs using DtCraft.

There are a number of approaches using GPU to accelerate Gibbs sampling. Due to memory limitation, large data sets require either multiple GPUs or iterative streaming to a single GPU. A powerful feature of DtCraft is the capability of distributed heterogeneous computing. Recall that our system offers a container layer of resource abstraction and users can interact with the scheduler to configure the set of computers on which their applications would like to run. We modified the container interface to include GPUs into resource constraints and implemented the GPU-accelerated Gibbs sampling algorithm by [50]. Experiments were run on 10 Amazon EC2 p2.xlarge instances. As shown in Figure 5.14, DtCraft can be extended to a hybrid cluster for higher speedup (56× faster than serial CPU with only 10 GPU machines). Similar applications that rely on off-chip acceleration can make use of DtCraft to broaden the performance gain.

5.5.4 Electronic Design Automation (EDA)

The recent semiconductor industry is driving the need of massively parallel integration to leverage the technology scaling [38]. We applied DtCraft to solve a large-scale EDA optimization problem, *physical design*, a pivotal stage that encompasses several steps from circuit partition to timing closure (see Figure 5.15). Each step has domain-specific solutions and engages with others through different internal databases. We used open-source tools and our internal developments for each step of the physical design [4, 9, 36]. Individual tools have been developed based on C++ with default I/O on files, which can fit into DtCraft without significant rewrites of codes. Altering the I/O channels is unsurprisingly straightforward because our stream interface is compatible with C++ file streams. We applied DtCraft to handle a typical physical design cycle under multiple timing scenarios. As shown in Figure 5.16, our implementation ran through each physical design step and coupled them together in a distributed manner. Generating the timing report is the most time-consuming step. We captured each independent timing scenario by one vertex and connected it to a synchronization barrier to derive the final result. Users can interactively access the system via a service vertex.

We derived a benchmark with two billion transistors from ICCAD15 and TAU15 contests [9, 51]. The DtCraft-based solution is evaluated on 40 Amazon EC2 m4.xlarge machines [47]. The baseline we considered is a batch run over all steps on a single machine that mimicked the normal design flow. The overall performance is shown in Figure 5.17. The first benefit of our solution is the saving of disk I/O (65 GB vs 11 GB). Most data are exchanged on the fly including those that would otherwise come with redundant auxiliaries through disk (50 GB parasitics in the timing step). Another benefit we have observed is the asynchrony of DtCraft. Computations are placed wherever stream fragments are available rather than blocking for the entire object to be present. These advantages have translated to effective engineering

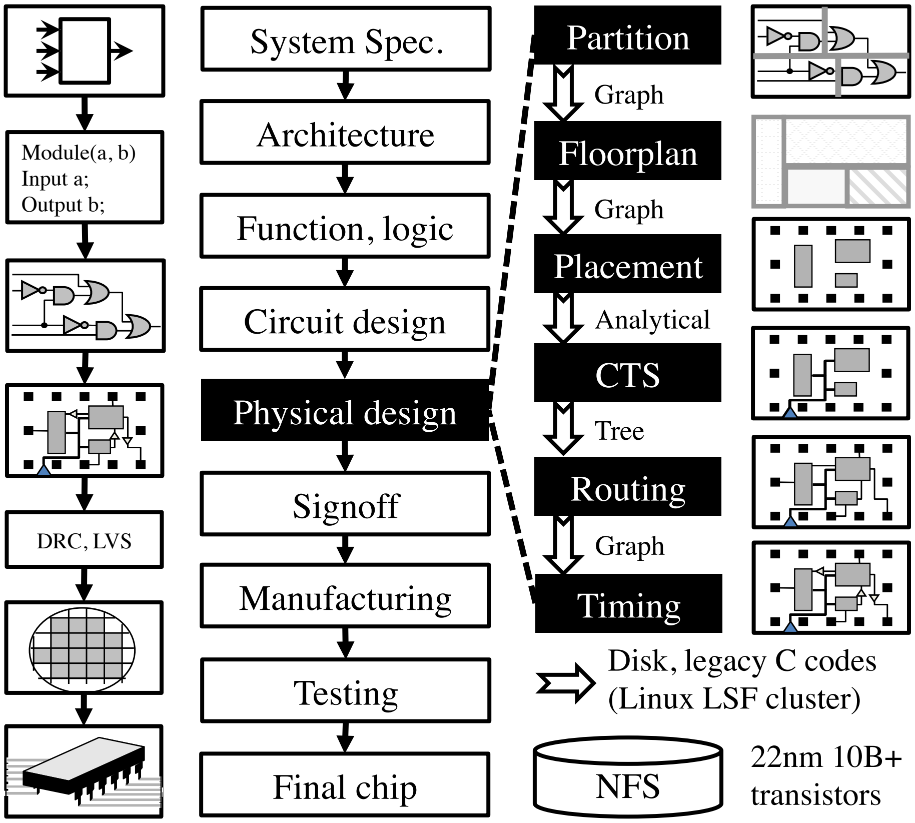


Figure 5.15: Electronic design automation of VLSI circuits and optimization flow of the physical design stage.

turnaround – *13 hours saving over the baseline*. From designers' perspective, this value convinces not only a faster path to the design closure but also the chance for breaking cumbersome design hierarchies, which has the potential to tremendously improve the overall solution quality [42, 38].

We next demonstrate the speedup relative to the baseline on different cluster sizes. In addition, we included the experiment in presence of a failure to demonstrate the fault tolerance of DtCraft. One machine is killed at a random time step, resulting in partial re-execution of the stream graph. As

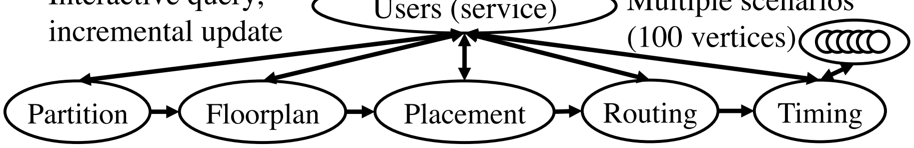


Figure 5.16: Stream graph (106 vertices and 214 edges) of our DtCraft-based solution for the physical design flow.

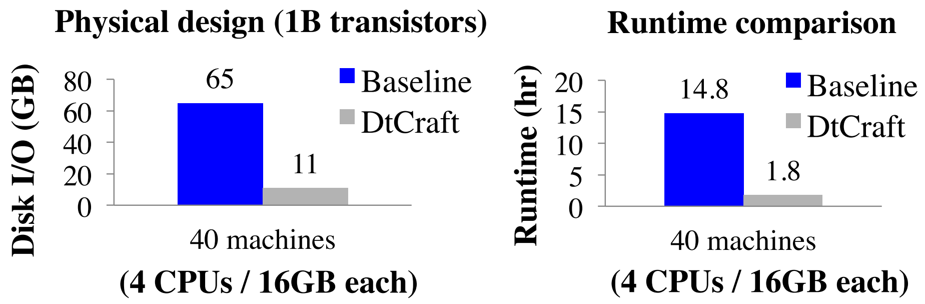


Figure 5.17: Performance of DtCraft versus baseline in completing the physical design flow.

shown in Figure 5.18, the speedup of DtCraft scales up as the cluster size increases. The highest speedup is achieved at 40 machines (160 cores and 640 GB memory in total), where DtCraft is 8.1× and 6.4× faster than the baseline. On the other hand, we have observed approximately 10–20% runtime overhead on fault recovery. We did not see pronounced difference from our checkpoint-based fault recovery mechanism. This should be in general true for most EDA applications since existing optimization algorithms are designed for "*medium-size data*" (million gates per partition) to run in main memory [38, 42]. In terms of runtime breakdown, computation takes the majority while about 15% is occupied by system transparency.

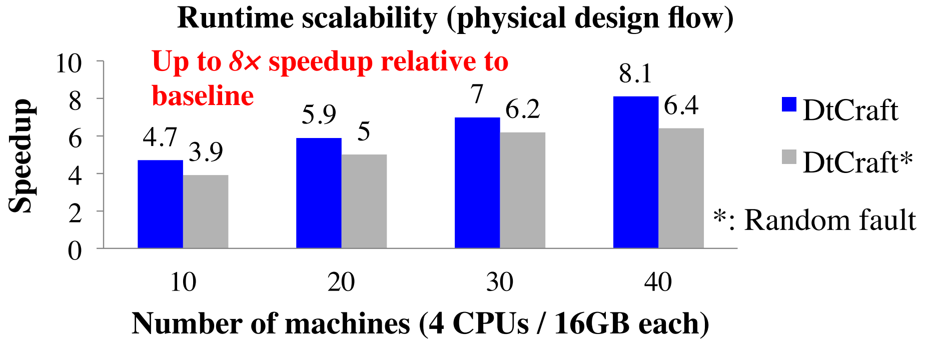


Figure 5.18: Runtime scalability in terms of speedup relative to the baseline on different cluster sizes.

Since timing analysis exhibits the most parallelism, we investigate into the performance gain by using DtCraft. To discover the system capability, we compare with the distributed timing analysis algorithm (ad-hoc approach) proposed by [42]. To further demonstrate the programmability of DtCraft, we compared the code complexity in terms of the number of lines of codes between our implementation and the ad-hoc approach. The overall comparison is shown in Figure 5.19. Because of the problem nature, the runtime scalability is even remarkable as the compute power scales out. It is expected the ad-hoc approach is faster than our DtCraft-based solution. Nevertheless, the ad-hoc approach embedded many hard codes and supports neither transparent concurrency nor fault tolerance, which is difficult for scalable and robust maintenance. In terms of programmability, our programming interface can significantly reduce the amount of the codes by 15×. The corresponding engineering efforts can be far beyond this number. Although this comparison might not be fair, it indeed reflected the potential engineering productivity that can be improved by DtCraft.

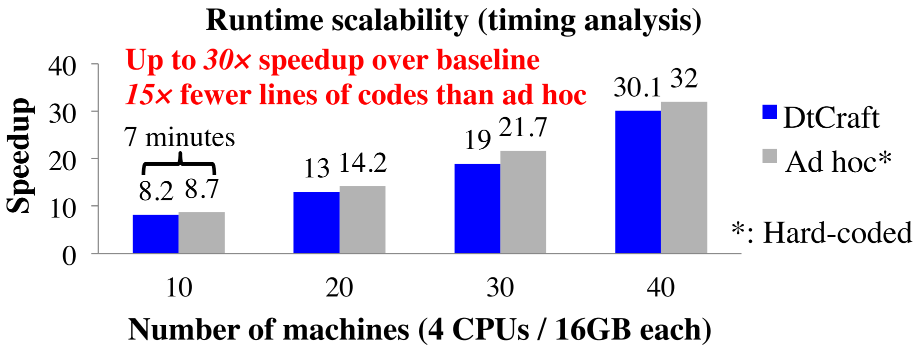


Figure 5.19: Performance comparison on distributed timing analysis between DtCraft-based approach and the ad-hoc algorithm by [42].

To conclude this experiment, we have introduced a platform innovation to solve a large-scale semiconductor optimization problem with low integration cost. To our best knowledge, this is the first work in the literature that achieves a distributed EDA flow integration. In addition, DtCraft also opens new opportunities for improving commercial tools, for example, distributed EDA algorithms and tool-to-tool integration. While this experiment demonstrates merely a successful prototype, we believe DtCraft can be extended to consider more general and complex design flows.

5.6 Conclusion

We have presented DtCraft, a distributed execution engine for high-performance parallel applications. DtCraft is developed based on modern C++17 on Linux machines. Developers can fully utilize rich features of C++ standard libraries along with our parallel framework to build highly optimized applications. Experiments on classic machine learning and graph applications have shown DtCraft outperforms the state-of-the-art cluster computing system by more than an order of magnitude. We have also successfully applied DtCraft to solve large-scale semiconductor optimization problems that are known difficult to fit into existing big data ecosystems. For many similar industry applications, DtCraft can be employed to explore integration and optimization issues, thereby offering new revenue opportunities for existing company assets.

CHAPTER 6

CONCLUSION AND FUTURE WORK

We have introduced in this thesis (1) an efficient PBA to remove pessimism from conventional STA flow, (2) an MapReduced-based distributed PBA framework that scales up to hundreds of machines, (3) a high-performance timing analysis tool, OpenTimer, (4) a distributed timing analysis framework for large design, and (5) a general-purpose distributed execution engine. We have released the source of our research to the public domain as vehicle for EDA and system research. There are many organizations and individuals using OpenTimer as either their business products or for contributions to the EDA community. OpenTimer has been selected as the golden timer in the TAU 2016 and TAU 2017 Timing Analysis Contests, and IEEE/ACM ICCAD 2015 CAD Contest. IEEE CEDA also used OpenTimer in their OpenDesign flow, aiming to promote open-source idea into EDA.

With our DtCraft system in place, there are a number of open opportunities. For example, we can create a software stack on top of DtCraft and build API for distributed machine learning applications and graph algorithms. Another important direction is to discover a suitable integration with POSIX-compliant distributed file systems. We are particularly interested in such one that supports both block- and object-based storage types. With the support of distributed storage, it is likely we can support MapReduce-like API to deal with data-intensive applications. We believe DtCraft can play an import role in speeding up the convergence between big data and big compute.

REFERENCES

[1] J. Hu, D. Sinha, and I. Keller, “TAU 2014 contest on removing common path pessimism during timing analysis,” in *Proc. ACM ISPD*, 2014, pp. 153–160.

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

[3] J. Zejda and P. Frain, “General framework for removal of clock network pessimism,” in *Proc. IEEE/ACM ICCAD*, 2002, pp. 632–639.

[4] “Tau 2014 contest: Pessimism removal of timing analysis,” http://sites.google.com/site/taucontest2014.

[5] S. Bhardwaj, K. Rahmat, and K. Kucukcaka, “Clock-reconvergence pessimism removal in hierarchical static timing analysis,” U.S. Patent 8 434 040, Apr 30, 2013.

[6] D. Hathaway, J. P. Alvarez, and K. P. Belkbale, “Network timing analysis method which eliminates timing variations between signals traversing a common circuit path,” U.S. Patent 5 636 372, Jan 3, 1997.

[7] A. K. Ravi, “Common clock path pessimism analysis for circuit designs using clock tree networks,” U.S. Patent 8 205 178, Jan 19, 2012.

[8] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, “UI-Timer: An ultra-fast clock network pessimism removal algorithm,” in *Proc. IEEE/ACM ICCAD*, 2014, pp. 758–765.

[9] “Incremental timing analysis and incremental CPPR,” http://sites.google.com/site/taucontest2015.

[10] V. Garg, “Common path pessimism removal: An industry perspective,” in *Proc. IEEE/ACM ICCAD*, 2014, pp. 592–595.

[11] C.-H. Tsai and W.-J. Mak, “A fast parallel approach for common path pessimism removal,” in *Proc. IEEE/ACM ASPDAC*, 2015, pp. 372–377.

[12] Y.-M. Yang, Y.-W. Chang, and I. H.-R. Jiang, “iTimerC: Common path pessimism removal using effective reduction methods,” in *Prof. IEEE/ACM ICCAD*, 2014, pp. 600–605.

[13] C. Kalonakis, C. Antoniadis, P. Giannakou, D. Dioudis, G. Pinitas, and G. Stamoulis, “TKtimer: Fast and accurate clock network pessimism removal,” in *Proc. IEEE/ACM ICCAD*, 2014, pp. 606–610.

[14] M. A. Bender and M. F. Colton, “The LCA problem revisited,” in *Proc. 4th Latin American Symposium on Theoretical Informatics*, 2000, pp. 88–94.

[15] H. Aljazzar and S. Leue, “K*: A heuristic search algorithm for finding the k shortest paths,” in *Artificial Intelligence*, 2011, pp. 2129–2154.

[16] D. Eppstein, “Finding the k shortest paths,” in *Proc. IEEE FOCS*, 1994, pp. 154–165.

[17] E. Q. V. Martins and M. M. B. Pascoal, “A new implementation of yen’s ranking loopless paths algorithm,” in *A Quaterly Journal of Operation Research*, 2003.

[18] W. Qiu and D. M. H. Walker, “An efficient algorithm for finding the k longest testable paths through each gate in a combinational circuit,” in *Proc. IEEE ITC*, 2003, pp. 592–601.

[19] J. Y. Yen, “Finding the k shortest loopless paths in a network,” *Manage. Sci.*, vol. 17, no. 11, pp. 712–716, 1971.

[20] M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, “Min-max heaps and generalized priority queue,” in *Commun. ACM*, 1986, pp. 996–1000.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Chapter 24: Single-source shortest paths, introduction to algorithm,” 2009.

[22] “OpenMP: Parallel programming API,” http://www.openmp.org.

[23] “OpenMPI: Open-source high-performance computing,” http://www.open-mpi.org.

[24] “Illinois campus cluster,” https://campuscluster.illinois.edu.

[25] T.-W. Huang and M. D. F. Wong, “OpenTimer: A high-performance timing analysis tool,” in *Proc. IEEE/ACM ICCAD*, 2015, pp. 895–902.

[26] S. Cristian, N. H. Rachid, and R. Khalid, “Efficient exhaustive path-based static timing analysis using a fast estimation technique,” U.S. Patent 8 079 004, Dec 13, 2011.

[27] R. Molina, "EDA vendors should improve the runtime performance of path-based timing analysis," in *Eletronic Design*, 2003.

[28] O. Levitsky, "Sign off quality hierarchical timing constraints: Wishful thinking or reality?" in *TAU Workshop*, 2014.

[29] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[30] "Apache Hadoop," http://hadoop.apache.org/.

[31] "MapReduce MPI library," http://mapreduce.sandia.gov/.

[32] "POSIX," https://computing.llnl.gov.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USNIX NSDI*, 2012.

[34] "ProtocolBuffer," https://developers.google.com/protocol-buffers/.

[35] "Libevent," http://libevent.ogr/.

[36] T.-W. Huang and M. D. F. Wong, "Accelerated path-based timing analysis with MapReduce," in *Proc. ACM ISPD*, 2015, pp. 103–110.

[37] "Adaptive computing," http://www.adaptivecomputing.com/.

[38] L. Stok, "The next 25 years in EDA: A cloudy future?" *IEEE Design Test*, vol. 31, no. 2, pp. 40–46, April 2014.

[39] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *ACM EuroSys*, 2007, pp. 59–72.

[40] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *USNIX OSDI*, 2008, pp. 1–14.

[41] "The future of big data," https://www2.eecs.berkeley.edu/patterson2016/.

[42] T.-W. Huang, M. D. F. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *ACM/IEEE DAC*, 2016, pp. 116:1–116:6.

[43] D. Charousset, R. Hiesgen, and T. C. Schmidt, "CAF - the C++ actor framework for scalable and resource-efficient applications," in *ACM AGERE!*, 2014, pp. 15–28.

[44] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *ACM SIGMOD*, 2010, pp. 135–146.

[45] "Libev," http://software.schmorp.de/pkg/libev.html.

[46] "Cereal," http://uscilab.github.io/cereal/index.html.

[47] "Amazon EC2," https://aws.amazon.com/ec2/.

[48] I. S. Bustany, D. Chinnery, J. R. Shinnerl, and V. Yutsis, "ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement," in *ACM ISPD*, 2015, pp. 157–164.

[49] T. Kiss, H. Dagdeviren, S. J. E. Taylor, A. Anagnostou, and N. Fantini, "Business models for cloud computing: Experiences from developing modeling simulation as a service applications in industry," in *WSC*, 2015, pp. 2656–2667.

[50] A. Terenin, S. Dong, and D. Draper, "GPU-accelerated Gibbs sampling," *CoRR*, vol. abs/1608.04329, 2016.

[51] "ICCAD CAD contest," http://cad-contest.el.cycu.edu.tw/CAD-contest-at-ICCAD2015/.