A Distributed Timing Analysis Framework for Large Designs

Tsung-Wei Huang
Dept. of ECE, UIUC, IL, USA
twh760812@gmail.com

Martin D. F. Wong
Dept. of ECE, UIUC, IL, USA
mdfwong@illinois.edu

Debjit Sinha
IBM Systems, Poughkeepsie, NY, USA
debjit.sinha@us.ibm.com

Kerim Kalafala
IBM Systems, Poughkeepsie, NY, USA
kalafala@us.ibm.com

Natesan Venkateswaran
IBM Systems, Poughkeepsie, NY, USA
natesan@us.ibm.com

ABSTRACT

Given ever-increasing circuit complexities, recent trends are driving the requirement for 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 paper, we introduce a DTA framework for large designs. Our framework supports (1) general design partitions in distributed file systems, (2) non-blocking IO with event-driven loop 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.

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 [9]. 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 [7]. However, it has been reported that a complete MTA on a design with 2 billion transistors can consume 400GB 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 [1, 11]. 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.

DAC '16, June 05-09, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4236-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2897937.2897959

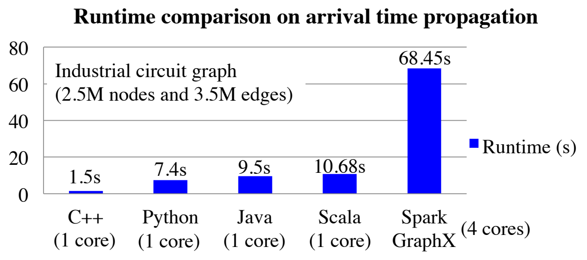


Figure 1: The need of specialized DTA framework.

The real problem is that most EDA tools are developed based on high-performance C/C++. A benchmark for language performance and system model on computing the arrival time from an industry circuit is shown in Figure 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 [10], the performance gap raises a big applicability concern. These evidences have convinced a need of specialized DTA framework. Consequently, we introduce in this paper 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 [3].

We have evaluated our framework on a commodity cluster with hundreds of machines and successfully performed DTA on large industry designs. Experimental results have demonstrated the scalability and effectiveness of our framework.

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 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.

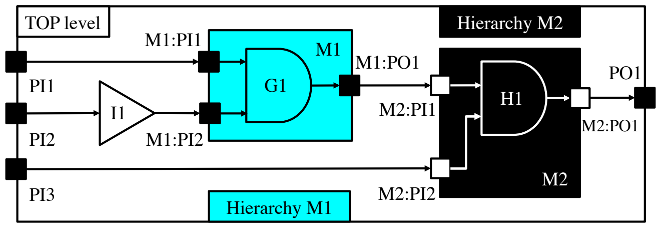


Figure 2: Example of two-level hierarchical partitions.

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.*

3. FRAMEWORK

The overview of our DTA framework is shown in Figure 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 [1, 5, 11].

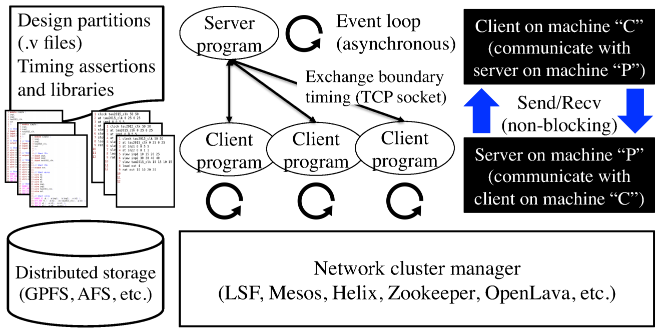

Design partitions (.v files) Timing assertions and libraries
Server program
Event loop (asynchronous)
Client on machine "C" (communicate with server on machine "P")
Exchange boundary timing (TCP socket)
Send/Recv (non-blocking)
Client program
Client program
Client program
Server on machine "P" (communicate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

Figure 3: Overview of our DTA framework.

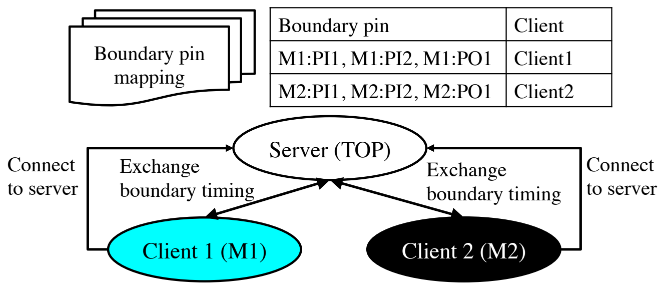
3.1 Distributed Storage and Cluster Manager

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.

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.

Figure 4 presents an example of the server-client model for the hierarchical partitions in Figure 2. 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



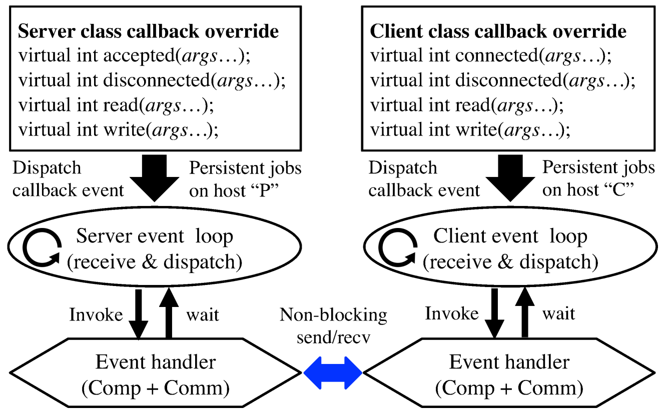
Boundary pin	Client
M1:PI1, M1:PI2, M1:PO1	Client1
M2:PI1, M2:PI2, M2:PO1	Client2

Figure 4: Server-client model for the two-level hierarchical partitions in Figure 2.

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.

3.3 Non-blocking IO and Event Loop

Network latency is typically at least ten times higher than in-memory reference [2]. 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 light-weight 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…);
```

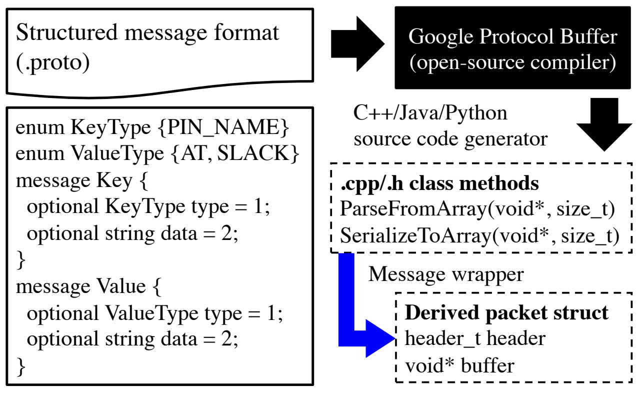
Dispatch callback event
Persistent jobs on host "P"
Dispatch callback event
Persistent jobs on host "C"
Server event loop (receive & dispatch)
Client event loop (receive & dispatch)
Invoke
wait
Non-blocking send/recv
Invoke
wait
Event handler (Comp + Comm)
Event handler (Comp + Comm)

Figure 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 5 presents the event-driven environment in our framework. Our framework applies the open-source package, *libevent*, as the event engine [2]. 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.

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 big-data community [3]. 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 6.



```
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;
}
```

.cpp/.h class methods

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

Derived packet struct

```
header_t header
void* buffer
```

Structured message format (.proto)
Google Protocol Buffer (open-source compiler)
C++/Java/Python source code generator
Message wrapper

Figure 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 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 4K bytes received) due to the network C10K issue [2]. 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. 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.

4.1 Server Program

The main body of the server program is presented in Algorithm 1. Algorithm 1 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 [2]. We add a listener event to B to note the callback *AcceptClientConnection* (in Algorithm 2) for any new TCP client connections (line 4). Finally, the event base is dispatched and the program enters an event loop (line 5).

Algorithm 1: Server(D, H, U)

```
Input: input data $D$, host $H$, user data $U$
Output: timing analysis report
1 $G \leftarrow$ parse_timing_graph($D$);
2 $S \leftarrow$ create_TCP_server_socket($H$);
3 $B \leftarrow$ create_event_base();
4 add_listener_event($B$, $S$, $U$, AcceptClientConnection);
5 dispatch_event_base($B$);
```

Algorithm 2: AcceptClientConnection(L, U)

```
Input: listener $L$, user data $U$
1 $B \leftarrow$ get_event_base($L$);
2 $S \leftarrow$ get_socket_info($L$);
3 add_socket_read_event($B$, $S$, ServerReadCallback, $U$);
```

Algorithms 2 and 3 present the two callbacks in server's program. Algorithm 2 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 read callback is given in Algorithm 3. 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).

Algorithm 3: ServerReadCallback(S, M, U)

```
Input: socket descriptor $S$, message $M$, user data $U$
1  $\Delta \leftarrow \phi$;
2  foreach complete packet $i \in M$ do
3      $\Omega \leftarrow$ deserialize_data($i$);
4      switch $i.type$ do
5          case BoundaryRegistration
6              foreach key-value pair $(k, v) \in \Omega$ do
7                  map_boundary_pin_to_socket($k$, $S$);
8              end
9              if all clients are online then
10                 foreach source port $r$ in top-level design do
11                     $v \leftarrow$ initial_timing_assertion($r$);
12                     $\Delta \leftarrow \Delta \cup \{$make_kv_pair($r$, $v$)$\}$;
13                 end
14             end
15         end
16         case UpdateBoundaryTiming
17             foreach key-value pair $(k, v) \in \Omega$ do
18                 $\Delta \leftarrow \Delta \cup \{(k, v)\}$;
19             end
20         end
21     endsw
22 end
23 $\Theta \leftarrow$ propagate_timing_and_get_new_boundary_pins($\Delta$);
24 foreach key-value pair $(k, v) \in \Theta$ do
25     $j \leftarrow$ serialize_data($k$, $v$);
26     $c \leftarrow$ get_boundary_pin_client_socket($k$);
27     send_packet($c$, $j$, UpdateBoundaryTiming);
28 end
```

4.2 Client Program

The main body of the client program is given in Algorithm 4. 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 4 and line 4:8 in Algorithm 5). This step is necessary for server program to keep track of the mapping between a boundary pin and its client identity. As presented in Algorithm 6, the read callback in client program resembles the procedure in Algorithm 3. From the view point of client, there is no need of branch for boundary pin registration. We only maintain a candidate set of pins received from 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 4: Client(D, H, U)

```
Input: input data $D$, server host $H$, user data $U$
Output: timing analysis report
1 $G \leftarrow$ parse_timing_graph($D$);
2 $S \leftarrow$ create_TCP_client_socket($H$);
3 $B \leftarrow$ create_event_base();
4 add_connect_event($B$, $S$, $U$, Connect);
5 dispatch_event_base($B$);
```

Algorithm 5: Connect(L, U)

```
Input: listener $L$, user data $U$
1 $B \leftarrow$ get_event_base($L$);
2 $S \leftarrow$ get_socket_info($L$);
3 add_socket_read_event($B$, $S$, ClientReadCallback, $U$);
4 $\Delta \leftarrow \Phi$;
5 foreach boundary pin $p$ in the design do
6     $\Delta \leftarrow \Delta \cup$ make_kv_pair($r$, NULL);
7 end
8 send_packet($S$, serialize_data($\Delta$), BoundaryRegistration);
```

4.3 Timing Propagation

We have presented our framework and developed the program architecture for distributed timing. Although design-

Algorithm 6: 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
            foreach key-value pair $(k, v) \in \Omega$ do
                $\Delta \leftarrow \Delta \cup \{(k, v)\}$;
            end
        end
    endsw
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
```

ers can customize their timing routines (in particular, line 23 in Algorithm 3 and line 12 in Algorithm 6), processing the timing propagation exhibits high similarities to finding the shortest and the longest paths in a graph [7, 8]. In this regard, we introduce two techniques that are generically useful for the development of timing propagation based on our framework.

4.3.1 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 [7]. 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 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 [8] can be applied as well.

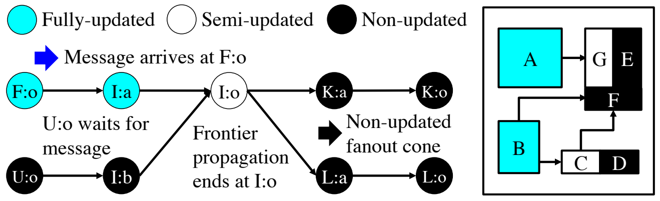


Figure 7: Frontier propagation follows the topological ordering of the timing graph.

4.3.2 Speculative Propagation

It can be observed in Figure 7 that the network delay might result in resource un-utilization (thread waiting for work). This is because there is no 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 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.

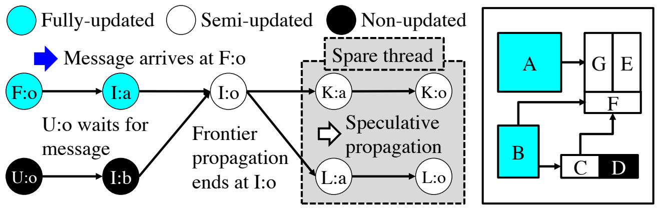


Figure 8: Spare thread performs speculative propagation in order to gain advanced saving of frontier work.

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 [2], and our messaging interface is built upon flexible `protocol buffer` [3]. Evaluation is taken on a computer cluster which has over 500 compute nodes. Each compute node is configured with 16 Intel 2.60GHz cores and 64GB 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 [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 [6] and large hierarchical designs generated by an industry standard timer. The benchmark statistics are summarized in Table 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.

5.2 Performance

The overall performance of our framework is listed in Table 1. In order to alleviate the uncertainty of network delay,

Table 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.

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

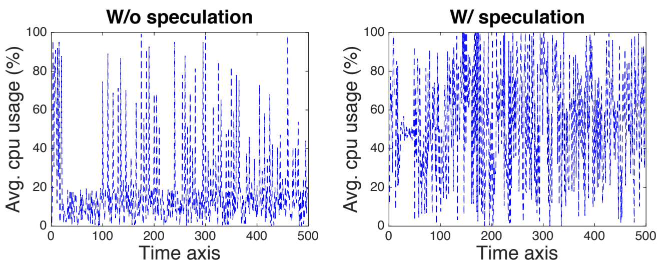


Figure 9: Average cpu utilization over time across all machines.

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 9.

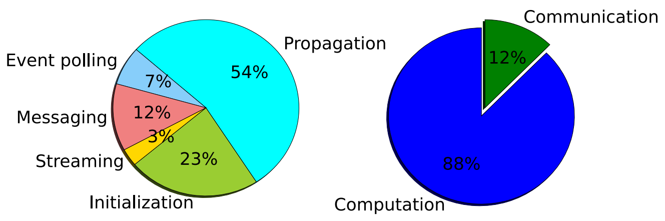


Figure 10: Runtime profile of our framework.

An in-depth view of the runtime profile is illustrated in Figure 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%.

6. CONCLUSION

In this paper, 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. 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. In the future, we plan to explore fault tolerance, distributed thread scheduling, and distributed common path pessimism removal.

7. REFERENCES

[1] Apache Hadoop, https://hadoop.apache.org/

[2] Libevent, https://libevent.org

[3] Protocol Buffer, https://developers.google.com/protocol-buffers/

[4] Adaptive Computing, http://www.adaptivecomputing.com/

[5] OpenLava, http://www.openlava.org

[6] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 Contest: Incremental Timing Analysis and Incremental CPPR," *Proc. IEEE/ACM ICCAD*, 2015

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

[8] T.-W. Huang and Martin D. F. Wong, "OpenTimer: A High-Performance Timing Analysis Tool," *Proc. IEEE/ACM ICCAD*, pp. 895–902, 2015.

[9] O. Levitsky, "Sign Off Quality Hierarchical Timing Constraints: Wishful Thinking or Reality?" *TAU workshop*, 2014.

[10] R. Xin, J. Rosen, M. J. Franklin, and I. Stoica, "GraphX: a resilient distributed graph system on Spark", *ACM GRADES*, 2013

[11] 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," *NSDI*, 2012