Distributed Timing Analysis: Framework and System

Tsung-Wei Huang, PhD candidate
University of Illinois at Urbana-Champaign (UIUC), USA, IL

Agenda

- **A distributed timing analysis framework for large designs**
  - Internship work with IBM Einstimer team @ Fishkill, NY
    - Debjit, Kerim, Natesan, etc.
  - Full timing analysis
  - DAC16
- **OpenTimer 2.0: Distributed timing analysis at scale**
  - System implementation beyond framework
  - Incremental timing and fault tolerance
  - Built upon OpenTimer 1.0
    - 1st place winner in TAU14, TAU16 timing analysis contests
    - 2nd place in TAU15 timing analysis contest
    - Open source in ICCAD15

A Distributed Timing Analysis Framework for Large Designs

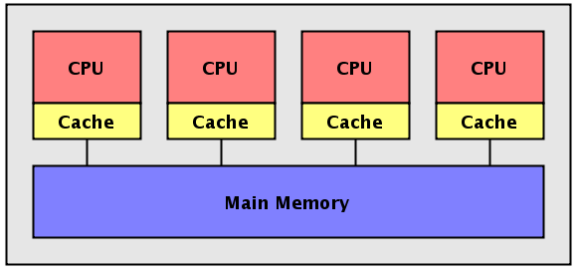
Tsung-Wei Huang, Martin D. F. Wong, (UIUC)
Debjit Sinha, Kerim Kalafala, and Natesan (IBM systems)

2016 ACM/IEEE Design Automation Conference (DAC), Austin, TX

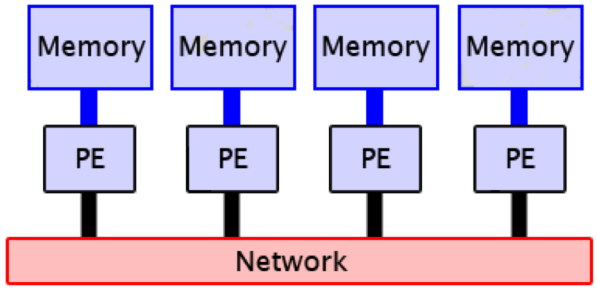
Distributed Timing – Motivation and Goal

- **Motivation**
  - Ever increasing design complexity
    - Hierarchical timing
    - Abstraction
    - Multi-threading timing analysis
  - Too expensive to afford high-end machines
    - ~400 GB memory (internal report)

- **Create a distributed timing engine**
  - Explore a feasible framework
  - Prototype a distributed timer
  - Scalability
  - Performance



Multi-threading in a single machine



Distributed computing on a machine cluster

State-of-the-art Distributed System Packages

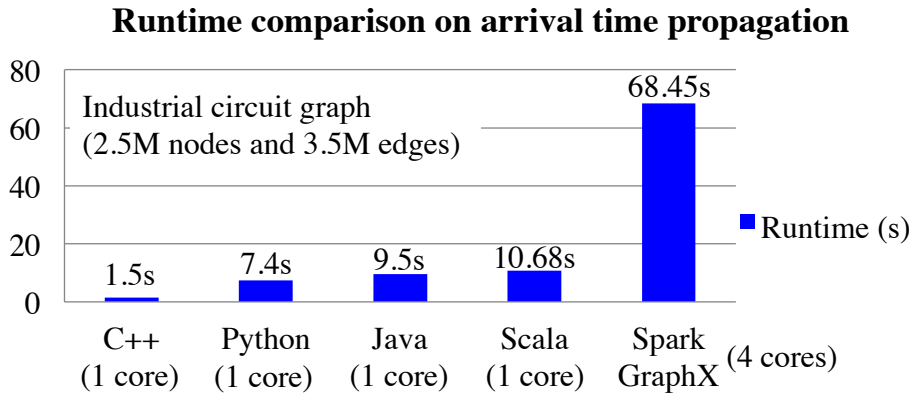
- **Open-source cloud computing platforms (*https://hadoop.apache.org/*)**
  - Hadoop
    - Reliable, scalable, distributed MapReduce platform on HDFS
  - Cassandra
    - A scalable multi-master database with no single points of failure
  - Chukwa
    - A data collection system for managing large distributed systems
  - Hbase
    - A scalable, distributed database that supports structured data storage
  - Zookeeper
    - Coordination service for distributed application
  - Mesos
    - A high-performance cluster manager with scalable fault tolerance
  - Spark
    - A fast and general computing engine for iterative MapReduce

Backup Slides

- ***Are these packages really suitable for our applications?***
  - Google/Hadoop MapReduce programming paradigm
  - Spark in-memory iterative batch processing
- ***What are the potential hurdles for EDA to use big-data tools?***
  - Big-data tools are majorly written in JVM languages
  - EDA applications highly rely on high-performance C/C++
  - Rewrites of numbers of codes
- ***What are the differences between EDA and big data?***
  - Computation intensive vs Data intensive
  - EDA data is more connected than many of social network

An Empirical Experiment on Arrival Time Propagation

- **Benchmark**
  - Timing graph from ICCAD 2016 CAD contest (*superblue18*)
    - 2.5M pins
    - 3.5M edges
- **Implementation**
  - Spark GraphX – 4 cores
  - Scala, Java, Python – 1 core
  - C++ – 1 core



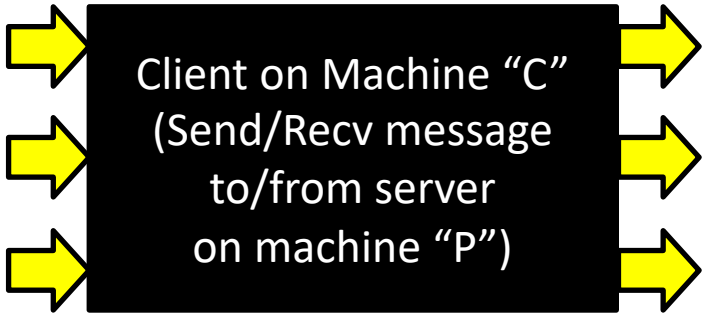
Implementation	Spark 1.4 (RDD + GraphX Pregel)	Java (SP)	C++ (SP)
Runtime (s)	68.45	9.5	**1.50**

Overhead of GraphX and distributed message passing

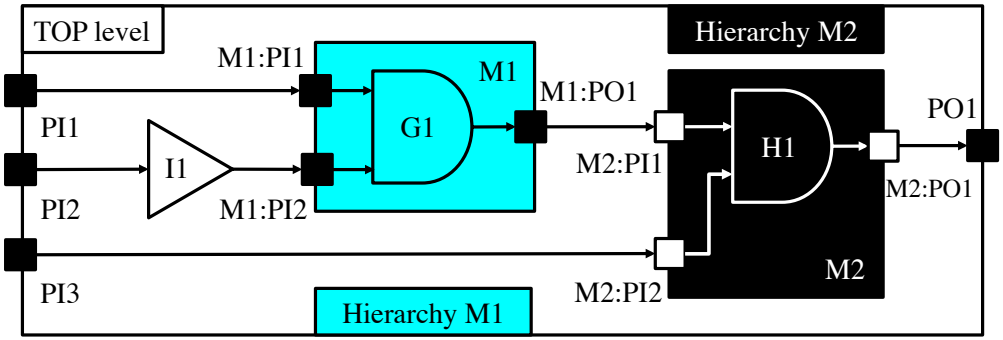
Overhead of JVM

The Proposed Framework for Distributed Timing

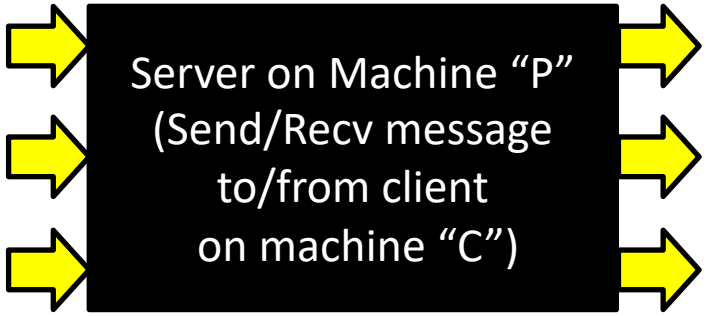
- **Focus on general design partitions**
  - Logical, physical, or hierarchical
  - Files stored in a distributed file system
- **Single-server multiple-client model**
  - Server is the centralized communicator
  - Clients exchange boundary timing with server



Client machine IP: 1.23.456.789



An example of three partitions for a two-level hierarchical design



Server machine IP: 140.110.44.32

Key Components in our Framework

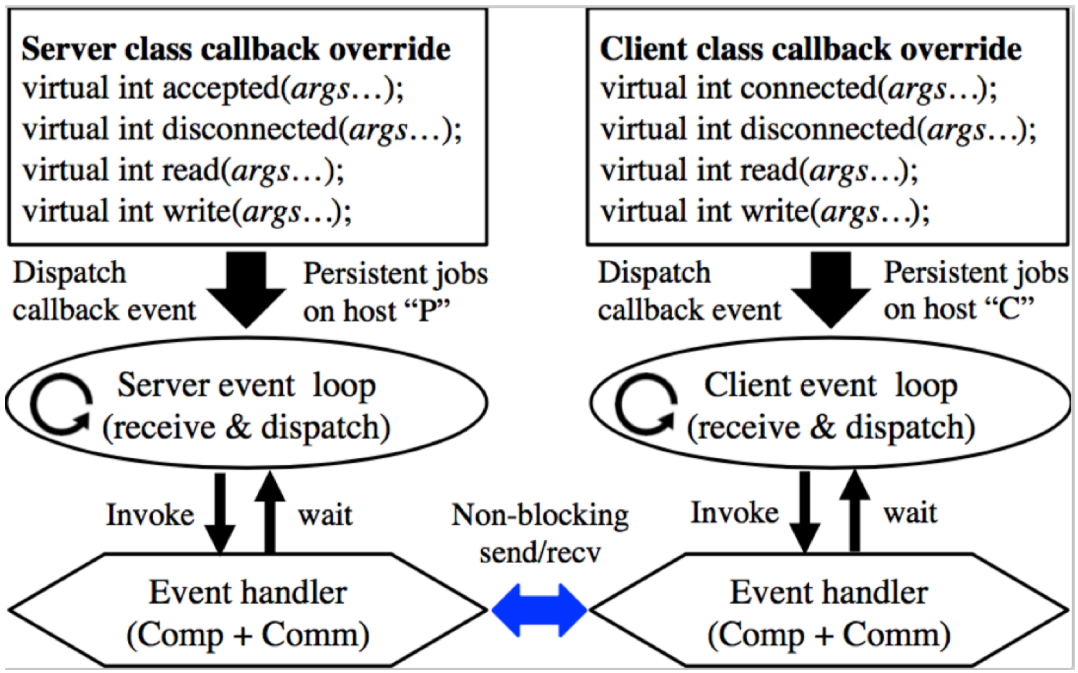
- **Non-blocking socket IO**
  - Program returns to users immediately (no deadlock)
  - Overlap communication and computation
- **Event-driven environment**
  - Replace tedious polling with autonomous callback for message events
  - Program persists in memory for efficient data processing
- **Efficient messaging interface**
  - Network see bytes only (unstructured)
  - Serialization and deserialization of timing data

Non-blocking IO and Event-driven Loop with Libevent

- **Libevent (*http://libevent.org/*)**
  - Open-source under BSD license
  - Actively maintained
  - C-based library
  - Non-blocking socket
  - Reactor model

```
// Magic inside dispatch call
while (!event_base_empty(base)) {
  // non-block IO by OS kernel
  active_list ← get_active_events
  foreach(event e in active_list) {
    invoke the callback for event e
  }
}
```

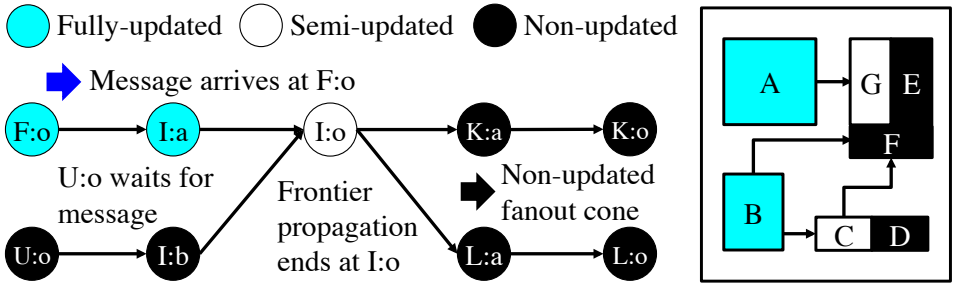
An example of an event-driven loop



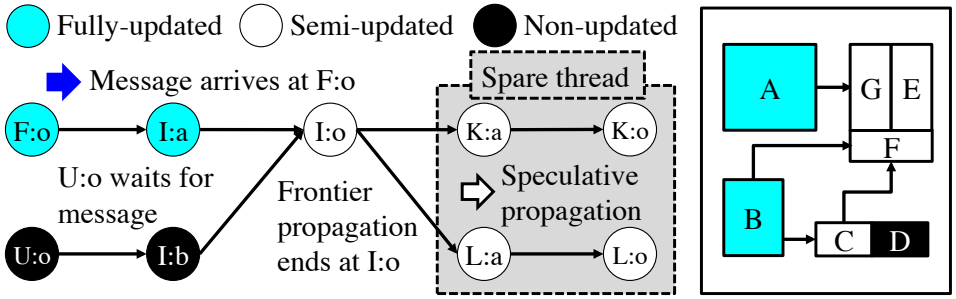
Interface class in our framework (override virtual methods for event callback)

Callback Implementation

- **Client read callback**
  - Receive boundary timing
  - Propagate timing
  - Send back to the server
- **Server read callback**
  - Keep boundary mapping
  - Receive boundary timing
  - Propagate timing
  - Send to the client
- **Timing propagation**
  - Frontier vs Speculative



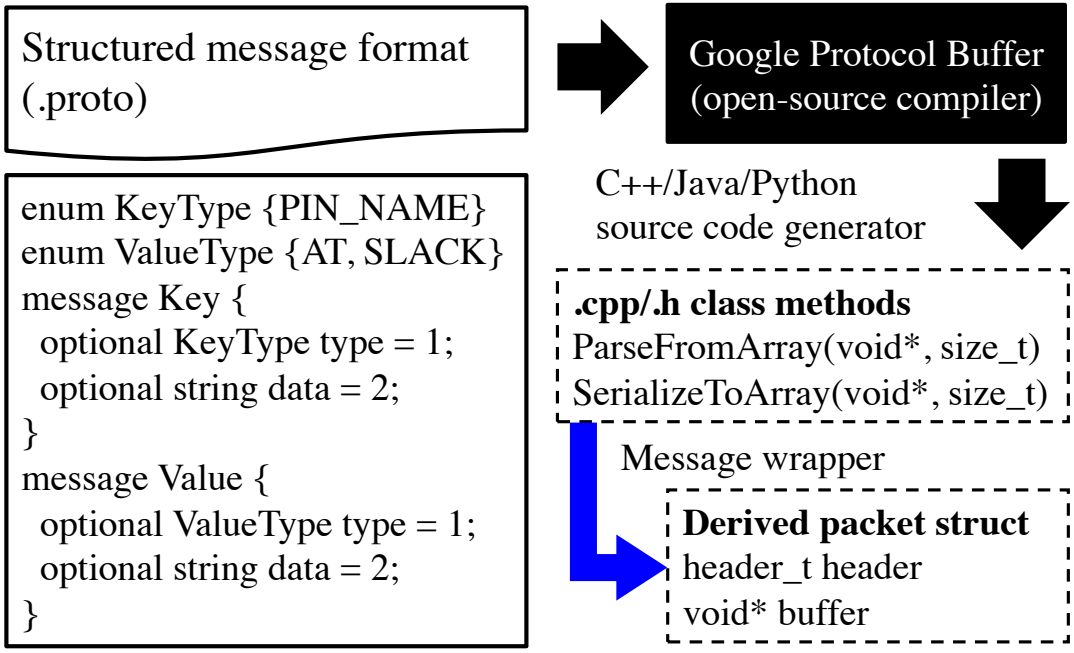
Frontier timing propagation follows the topological order of the timing graph



If multi-threading is available, spare thread performs speculative propagation in order to gain advanced saving of frontier work

Efficient Messaging Interface based on Protocol Buffer

- **Message passing**
  - Expensive
  - TCP byte stream
  - Unstructured
- **Data conversion**
  - Serialization
  - Deserialization
- **Protocol buffer**
  - Customized protocol
  - Simple and efficient
  - Built-in compression



Integration of Google's open-source protocol buffer into our messaging interface greatly facilitates the data conversion between application-level developments and socket-level TCP byte streams.

Evaluation – Software and Hardware Configuration

- **Written in C++ language on a 64-bit Linux machine**
- **3rd-party library**
  - Libevent for event-driven network programming
  - Google's protocol buffer for data serialization and deserialization
- **Benchmarks**
  - 250 design partitions generated by IBM EinsTimer
  - Millions-scale graphs generated from TAU and ICCAD contests
- **Evaluation environment**
  - UIUC campus cluster (https://campuscluster.illinois.edu/)
  - Each machine node has 16 Intel 2.6GHz cores and 64GB memory
  - 384-port Mellanox MSX6518-NR FDR InfiniBand (gigabit Ethernet)
  - Up to 500 machine nodes (8000 cores in total)

Evaluation – Performance

- **Overall performance**

Single thread | Two thread

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%

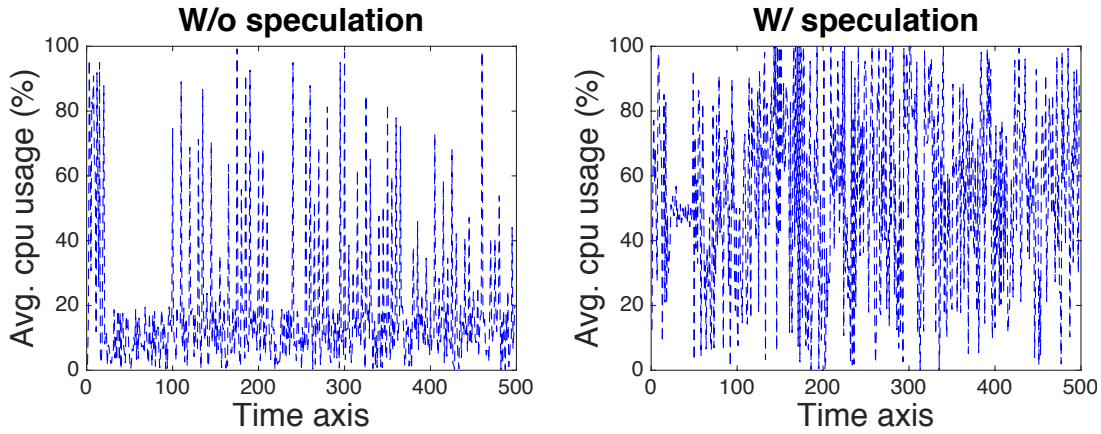
$|G|$: # of gates. $|N|$: # of nets. $|V|$: # of nodes. $|E|$: # of edges. $|P|$: # of partitions. L: # of levels. cpu: runtime. mem: peak memory on a program. msg: amount of message passing. usage: avg cpu utilization on a program.

- **Scalability**
  - Scale to 250 machines (DesignA)
  - Handle large designs (128M pins DesignD)
- **Runtime efficiency**
  - Less than 1 hour on large designs (DesignC and DesignD)
- **Memory usage**
  - Peak usage is only about 5GB on a machine (DesignD)

Evaluation – Profiling

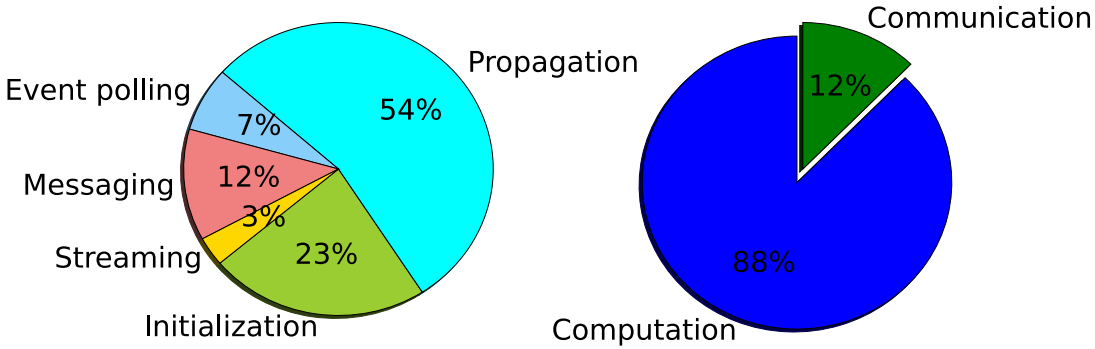
- **CPU utilization**
  - W/o speculation
  - W speculation

W/ speculation on DesignD
+49% cpu rate
+4.8MB on message passing



Average cpu utilization over time across all machines

- **Runtime profile**
  - 7% event polling
  - 3% streaming
  - 23% initialization
  - 54% propagation
  - 12% communication



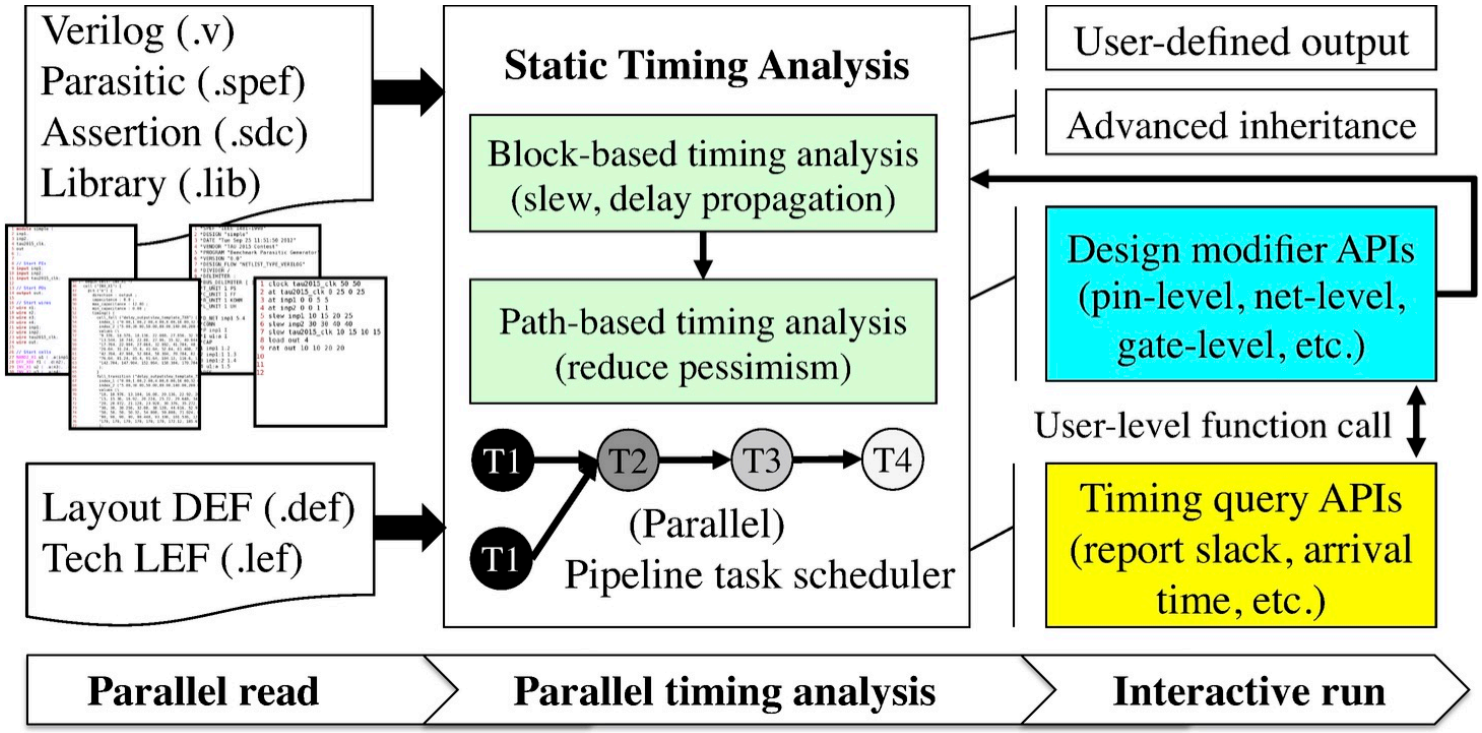
Runtime profile of our framework (12% on communication and 88% on computation)

OpenTimer 2.0: Distributed Timing Analysis at Scale

Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong

On-going Research: OpenTimer 2.0

- **A distributed timing analysis engine**
  - Built upon open-source tool OpenTimer 1.0

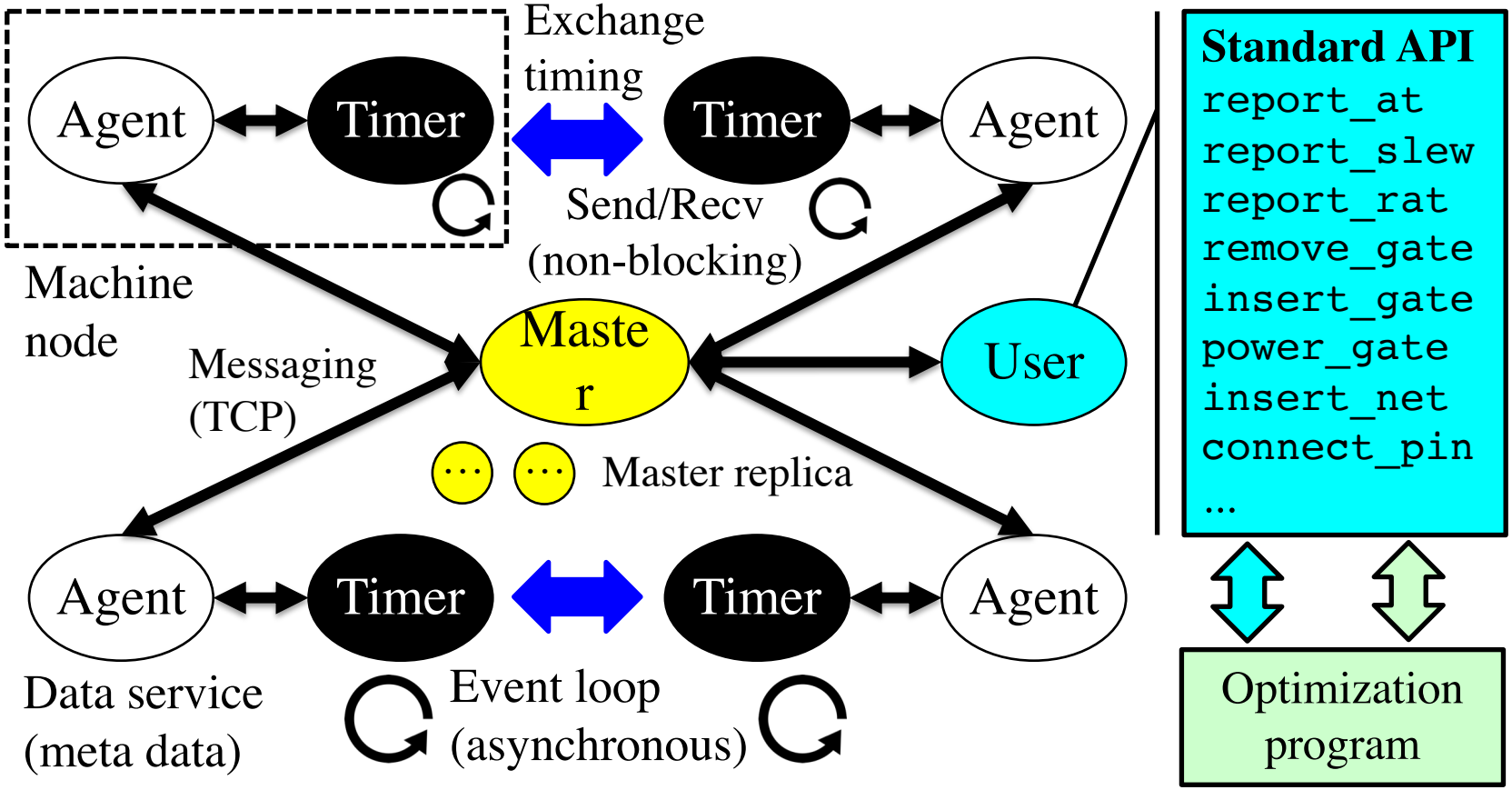


- ✓ C++11
- ✓ Industry format
- ✓ STA engine
- ✓ Block-based
- ✓ Path-based
- ✓ Incremental
- ✓ Lazy evaluation
- ✓ CPPR
- ✓ Multi-threaded

Architecture of OpenTimer 1.0 (ICCAD14, ICCAD15, TCAD16)

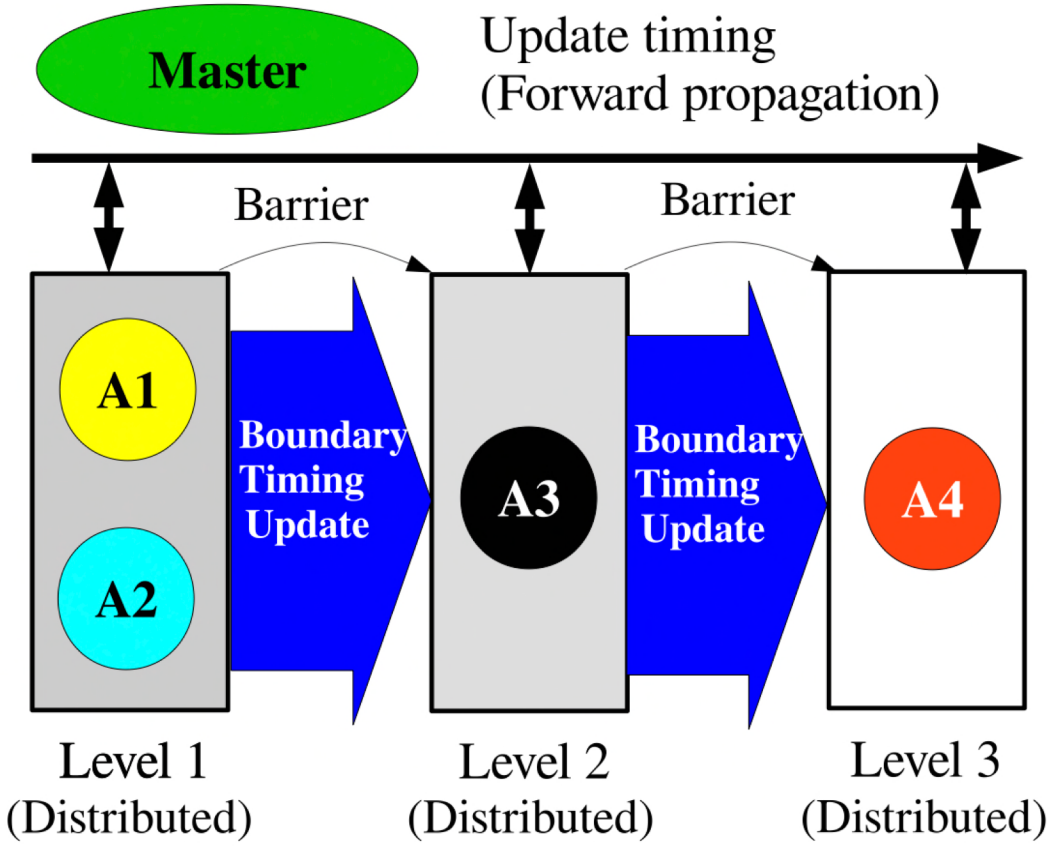
Architecture of OpenTimer 2.0

- **Master (coordinator) – agent (timer instance) model**



Feature 1: Incremental Timing

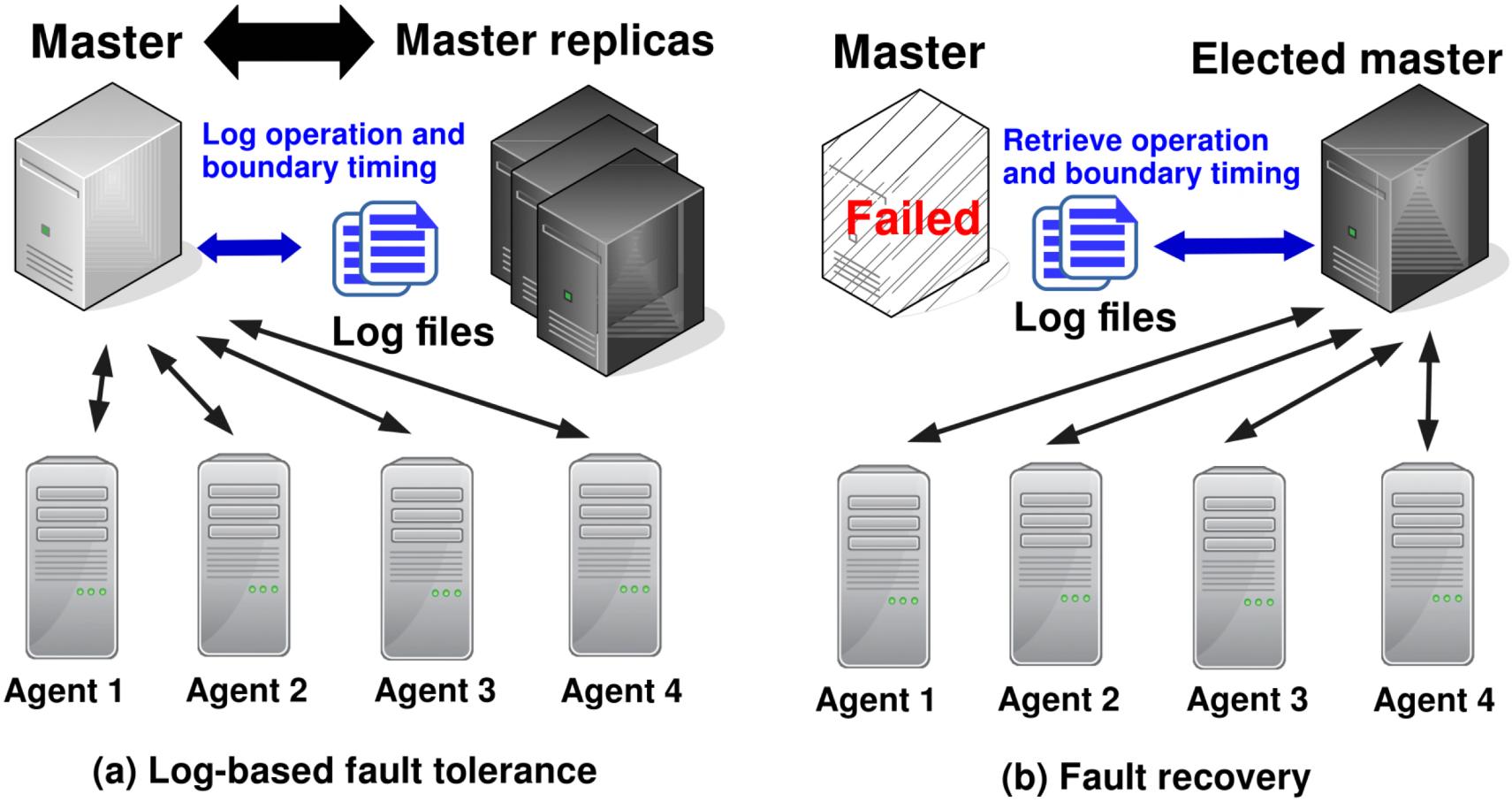
- Master maintain an levelized agent graph



- **Synchronization**
  - Between levels
  - MapReduce style
- **Runtime profile**
  - 75% communication
  - 25% computation
- **Load balancing**

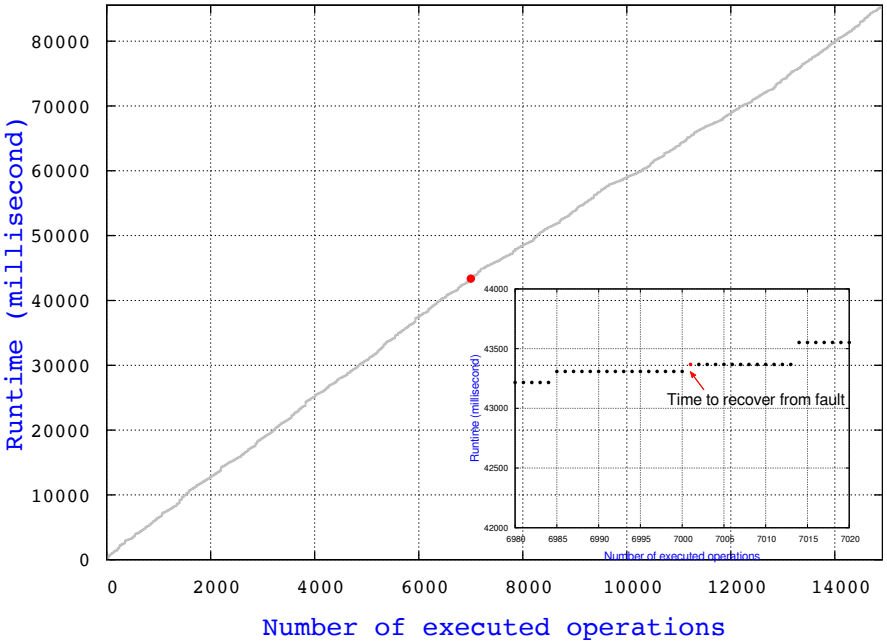
Feature 2: Fault Tolerance

- Master replica and consistent logging on operations

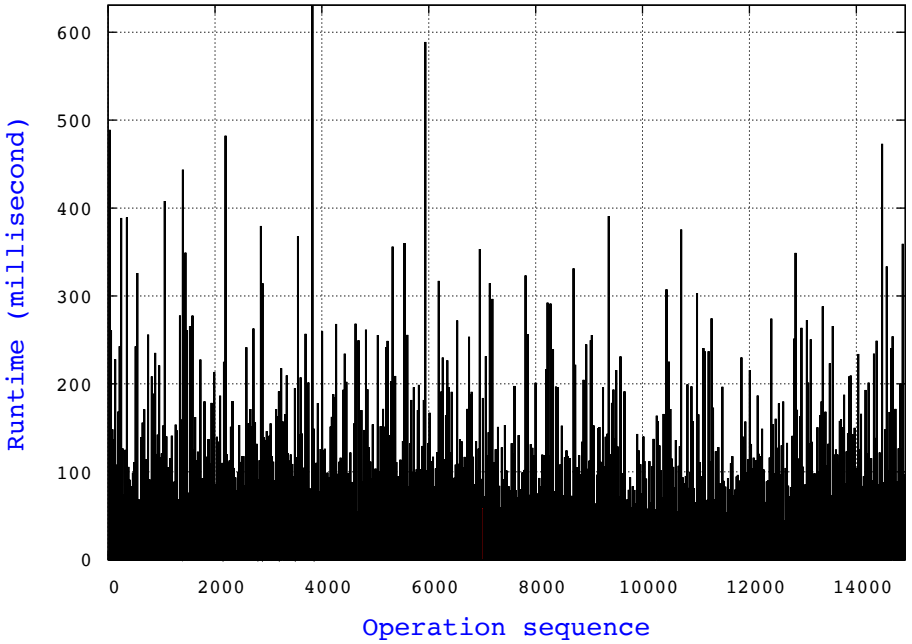


Experimental Results: Runtime and Fault Recovery

- **Benchmarks from TAU 2015 timing analysis contest**
  - 21M nodes and 78M edges
  - 14000 operations (design modifiers and timing queries)



Accumulated runtime vs operations



Runtime histogram of operations

Conclusion

- **Distributed timing analysis**
  - Server-client model
  - Non-blocking socket IO (overlap communication and computation)
  - Event-driven loop (autonomous programming)
  - Efficient messaging interface (serialization and deserialization)

- **Take-home message**
  - Rethink a distributed system for computation-intensive applications
  - Asynchronous data flow and resource control are important
  - Transparency, fault tolerance, and scalability

- **Acknowledgment**
  - UIUC CAD group
  - EinsTimer team (Debjit, Kerim, Natesan, Hemlata, Adil, Jin, Michel, etc.)

THANK YOU!