State History System

Alexandre Montplaisir
Michel Dagenais

May 5th, 2011
École Polytechnique, Montréal

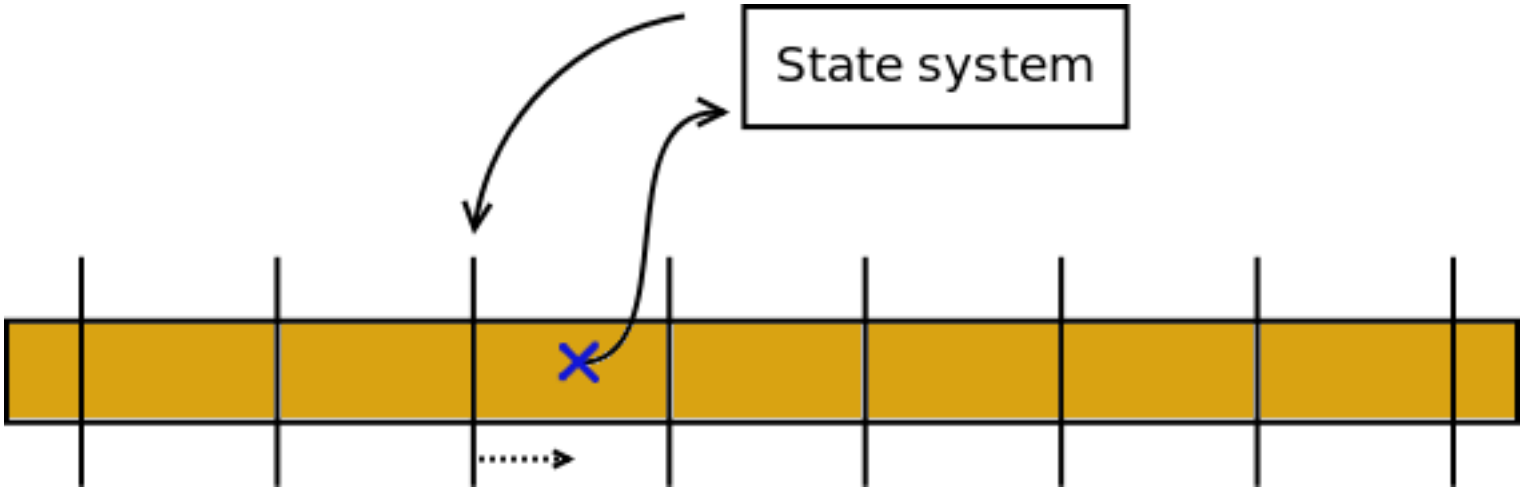
Contents

- Summary
  - Checkpoints vs. State History
  - History Tree
- State History System library
- Converting trace events to state changes
- Performance results
- Conclusion

Summary

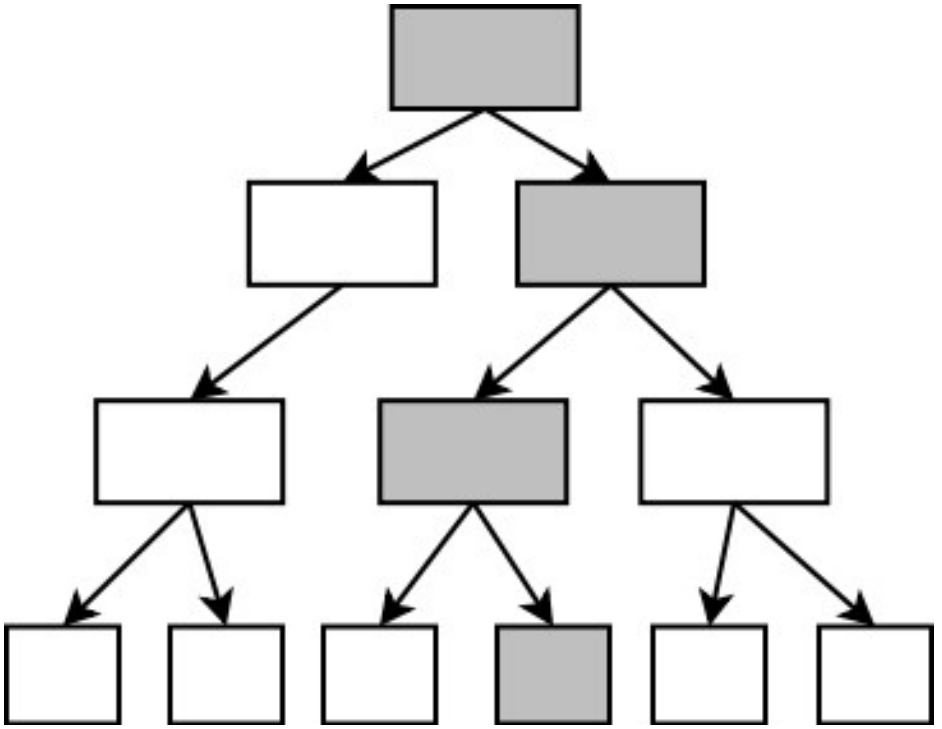
- Trace viewers need to be able to re-create the complete state the machine was in, at any given point in a trace.
- State information includes:
  - Running processes
  - Open file descriptors
  - State of CPUs, block devices, ...
  - etc.

Summary
The checkpoint method



- Instead we wanted a system that:
  - is more generic
  - stores the data on disk (better scalability)

Summary
History Tree

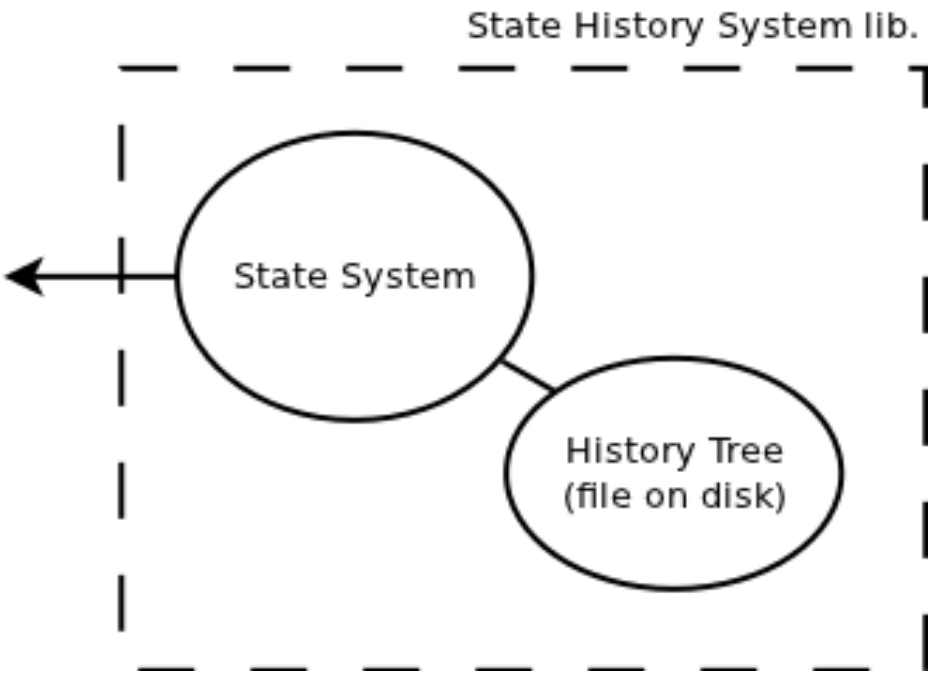


- Generic interval tree
- Optimized for disk
- Best if intervals are inserted in ascending order of end-times

https://projectwiki.dorsal.polymtl.ca/images/1/17/AMG_StateHistory_29062010.pdf

State History System library

- *State System*
  - Maintains the *Current state*
  - Optionally generates state intervals for the History Tree.
  - Can restore the current state for any time position.



State History System library Attribute Tree

- *Attribute*

  Atomic unit of state (scalar)
- The tree nodes are added as we insert state values.
- Each attribute can be accessed by:
  - Relative or absolute path (*"Threads", "1", "Status"*)
  - String or pre-compiled ID for path components

```
<hostname>
 \_ CPUs
 |   \_ CPU0
 |   |   \_ Current_thread
 |   |   \_ IRQ_stack
 |   |
 |   \_ CPU1
 |       ...
 |
 \_ Threads
     \_ PID1
     |   \_ PPID
     |   \_ TGID
     |   \_ Status
     |   \_ Executable_name
     |   \_ Exec_mode_stack
     |
     \_ PID2
        ...
```

State History System library
The API

- Building the History

  modify(timestamp, value, attribute)

  remove(ts, attribute)

  push(ts, value, attribute)

  pop(ts, attribute)

  increment(ts, attribute)

State History System library
The API

- Queries
  - When there is no History (streaming, ...)

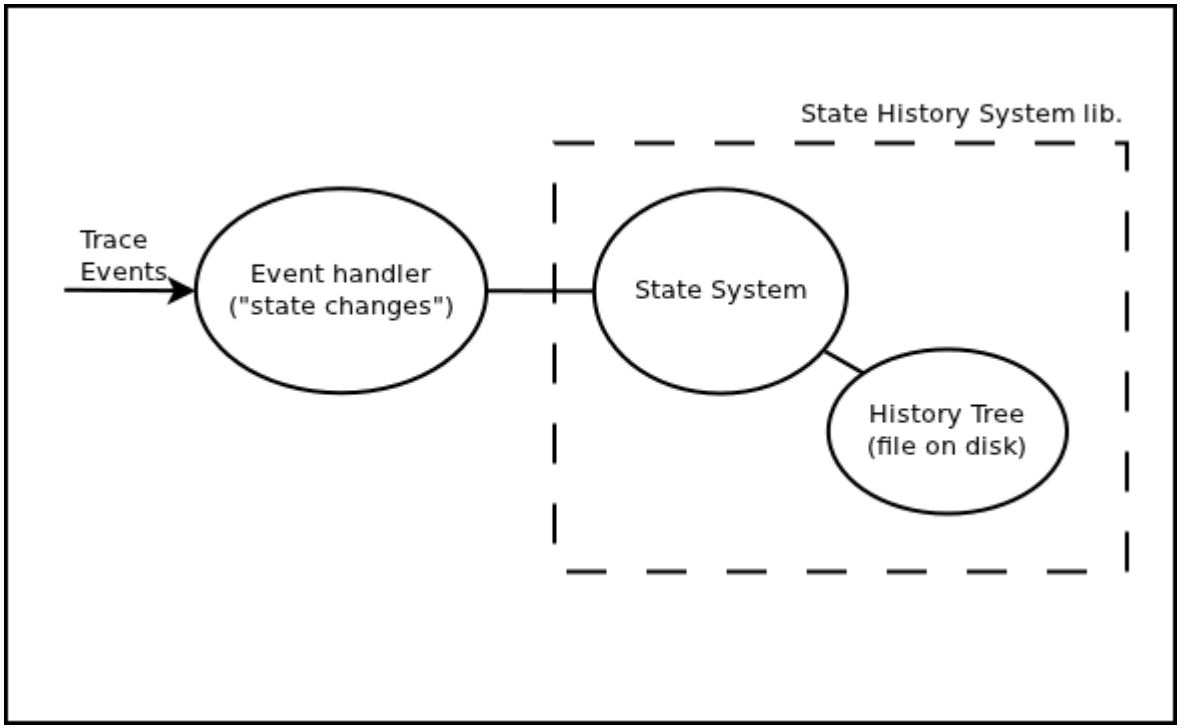
    getCurrentStateValue(attribute)
  - Updating the whole Current State

    loadStateAtTime(timestamp)

    getStateValue(attribute)
  - Single values, without updating C.S.

    getSingleStateValue(timestamp, attribute)

Converting trace events to state changes



- Next step: add an *Event Handler*, in which we define *state changes* for given event types.

Converting trace events to state changes

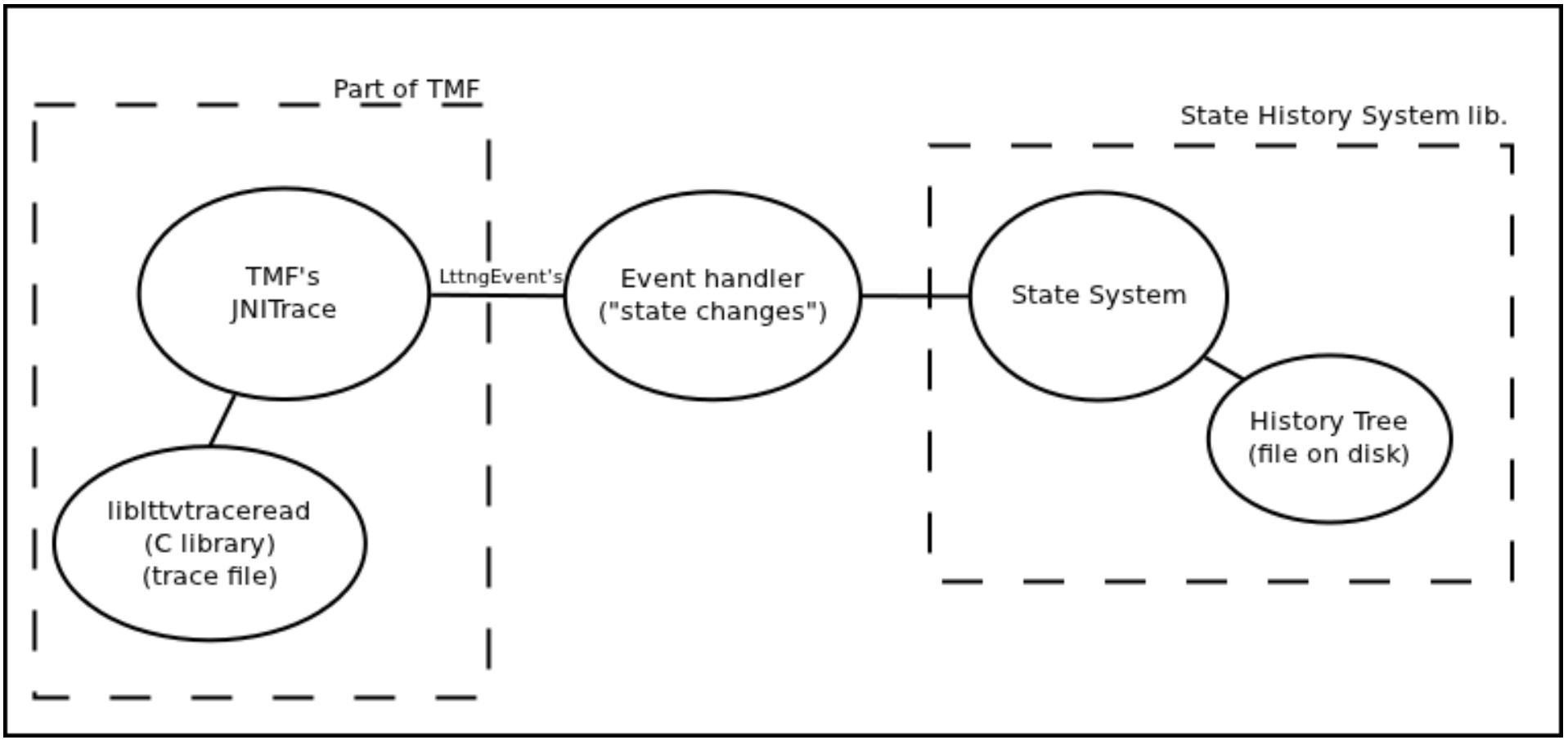
- Event Handler prototype for LTTng kernel traces:

```
switch ( event.getType() ) {

case LTT_EVENT_SYSCALL_ENTRY:
    ss.pushAttribute(ts,
                     LTTV_STATE_SYSCALL,
                     ["Threads", eventPID.toString(), "Exec_mode_stack"]);
    break;


case LTT_EVENT_SYSCALL_EXIT:
    ss.popAttribute(ts,
                    ["Threads", eventPID.toString(), "Exec_mode_stack"]);
    break;
```

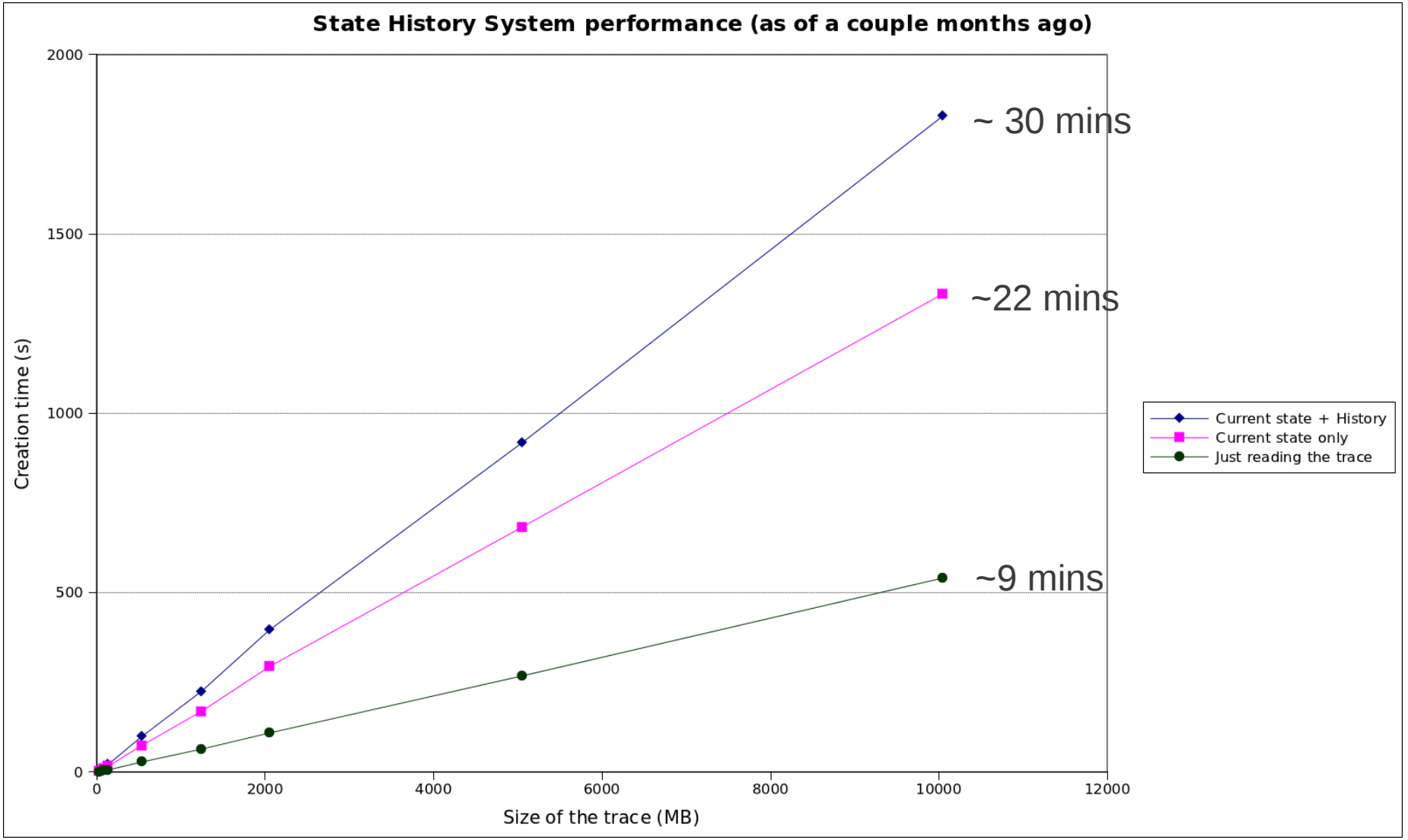
Converting trace events to state changes

```
case LTT_EVENT_SCHED_SCHEDULE:
    /* Read information from the event payload */
    nextPid = (Long) event.getContent().getField(0).getValue();
    prevPid = (Long) event.getContent().getField(1).getValue();
    stateOut = (Long) event.getContent().getField(2).getValue();

    /* Set the status of the new scheduled process */
    ss.modifyAttribute(ts,
                       LTTV_STATE_RUN,
                       ["Threads", nextPid.toString(), "Status"]);

    /* Set the status of the process that got scheduled out */
    ss.modifyAttribute(ts,
                       stateOut.intValue(),
                       ["Threads", prevPid.toString(), "Status"]);

    /* Set the current scheduled process on the relevant CPU */
    ss.modifyAttribute(ts,
                       nextPid.intValue(),
                       ["CPUs", event.getCPU().toString(), "Current_thread"]);
    break;
...
}
```

Converting trace events to state changes



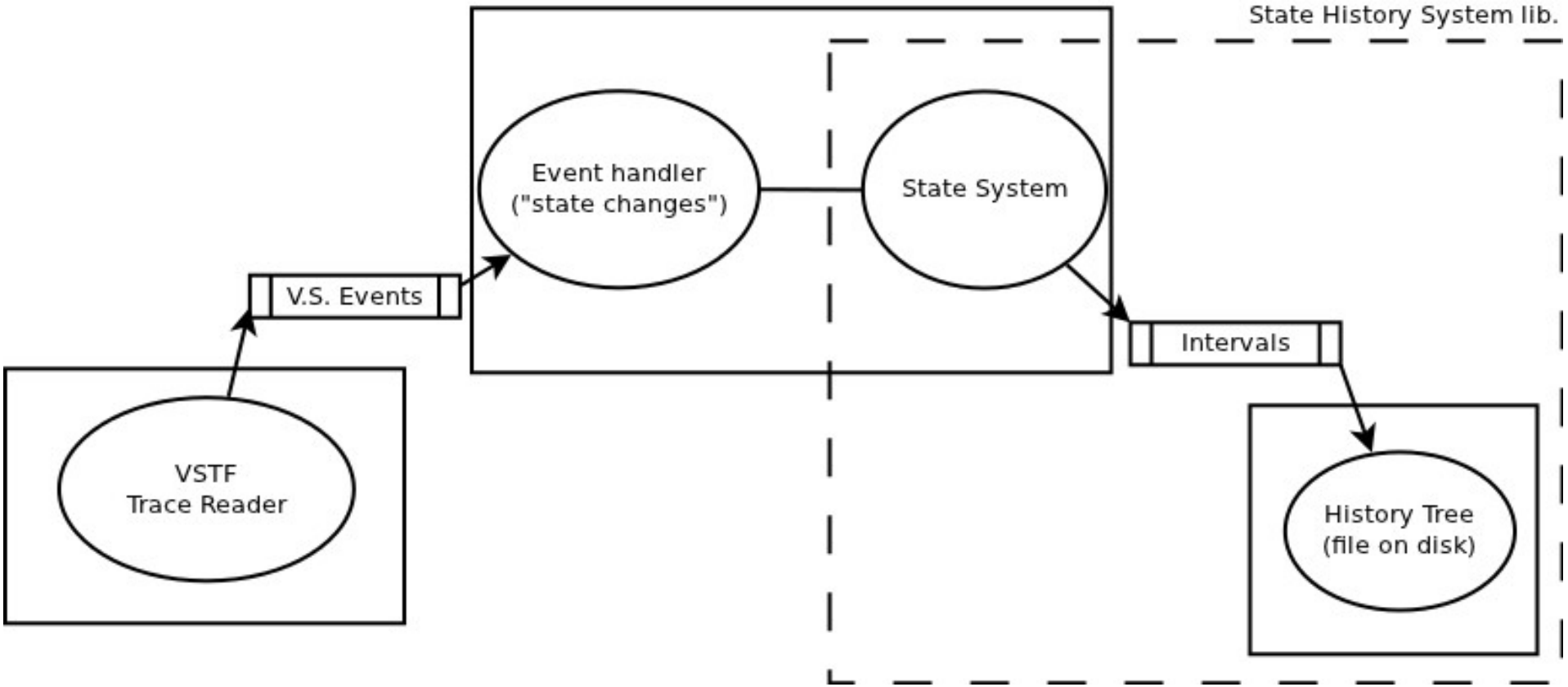
Performance results



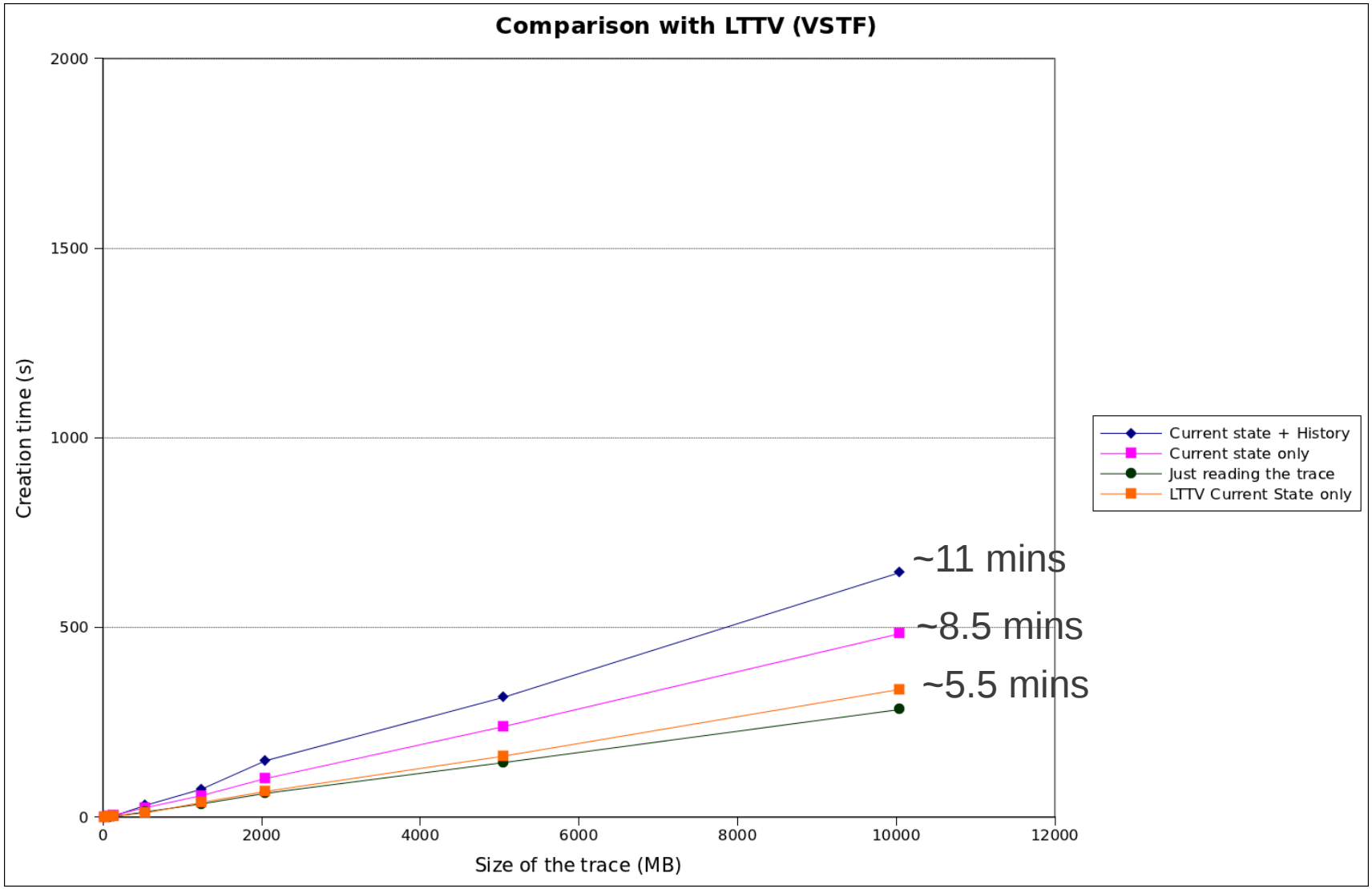
Performance results

- Recent optimizations:
  - Avoid re-walking the Attribute Tree whenever possible (hashing strings, etc.)
  - Keep handles to the Attribute Tree nodes across events
  - Have the processing done in a thread separate from those accessing the disk
  - Read traces directly from Java (bypass JNI)

Performance results



Performance results



Future work

- Further performance analysis
  - Measure the time decomposition for each operation
  - Compare alternative tree topology and parameters for the History Tree on disk (e.g. R-Trees)
- Revise and complement the API
- Propose and adapt for upstream TMF

Questions?

Thank you!