State History Storage in Disk-based Interval Trees

Alexandre Montplaisir

June 29, 2010

École Polytechnique de Montréal

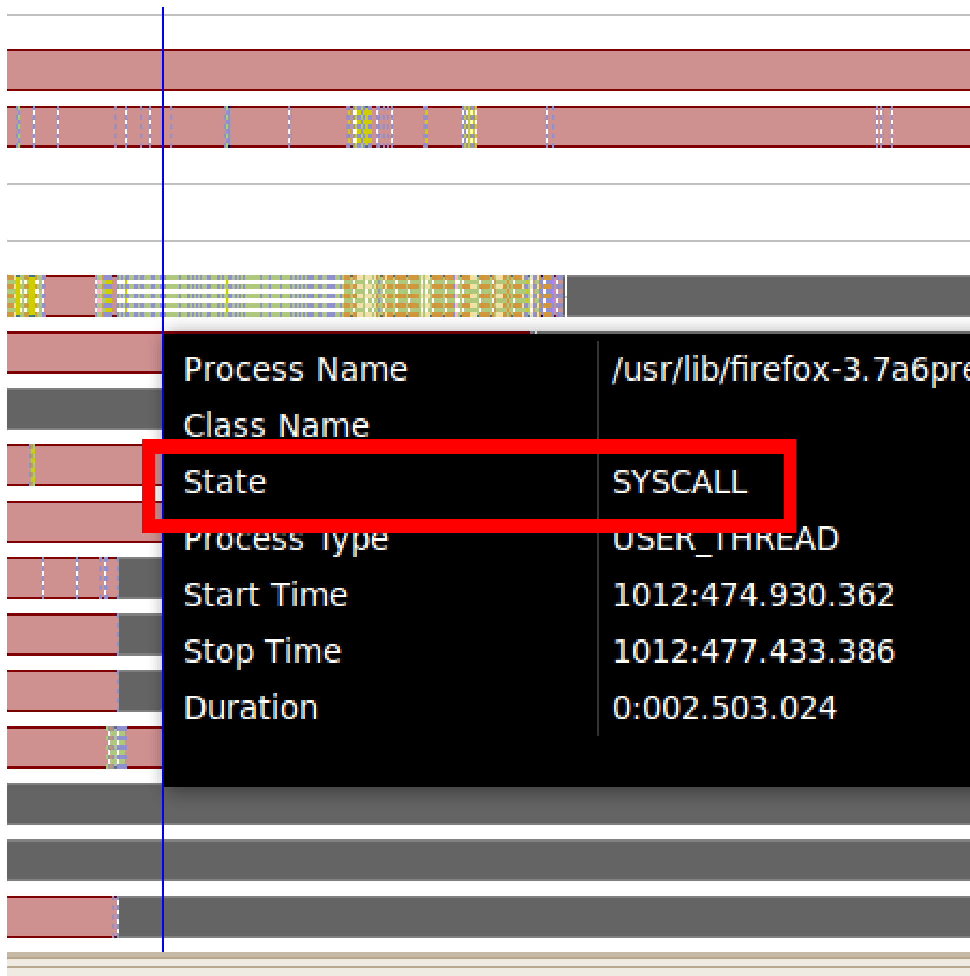
Content

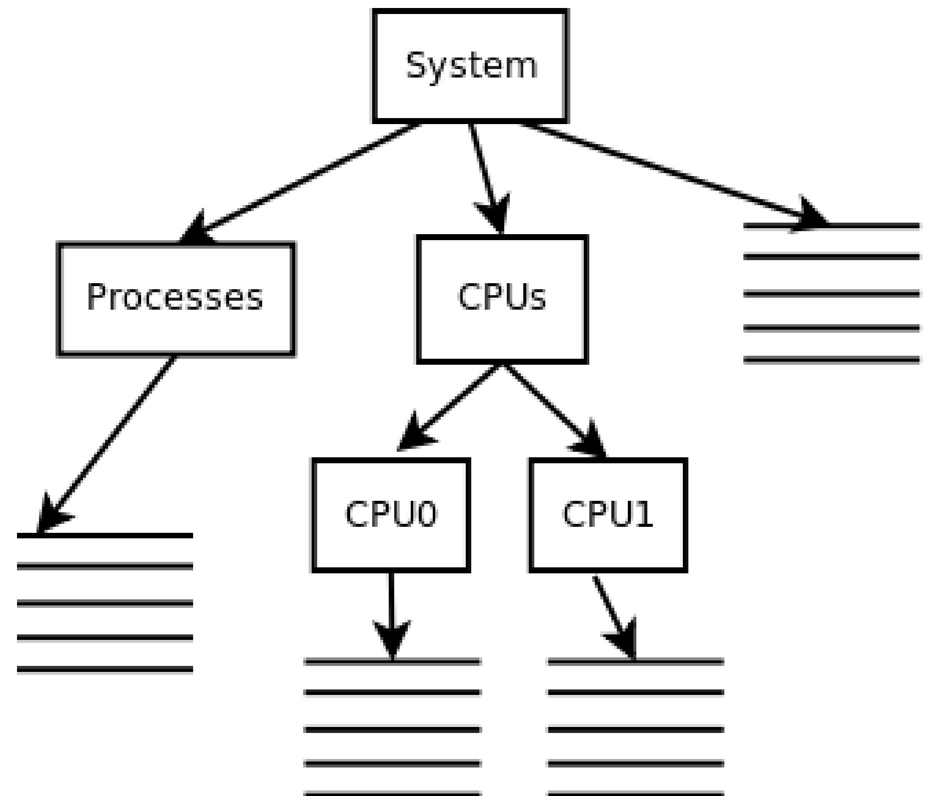
- Introduction : The concept of “State”
- The current method : Checkpoints
- The proposed State History Tree
  - Construction of the tree (insertions)
  - Queries
  - Integration with the viewers / trace reading library
  - Advantages / limitations
- Preliminary test results
- Conclusion : The next steps

The concept of “State”

- A state is a “snapshot” of a running system at one given time.
- We can represent state information at different levels, for example:
  - System level (running programs, open file descriptors)
  - CPU level (what is scheduled on this particular CPU)
  - Process level (PID, PPID, execution mode, ...)

The concept of “State”

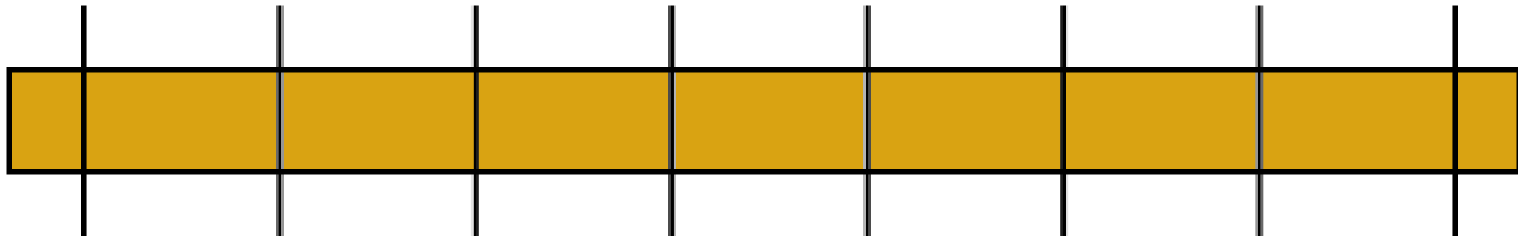




The concept of “State”

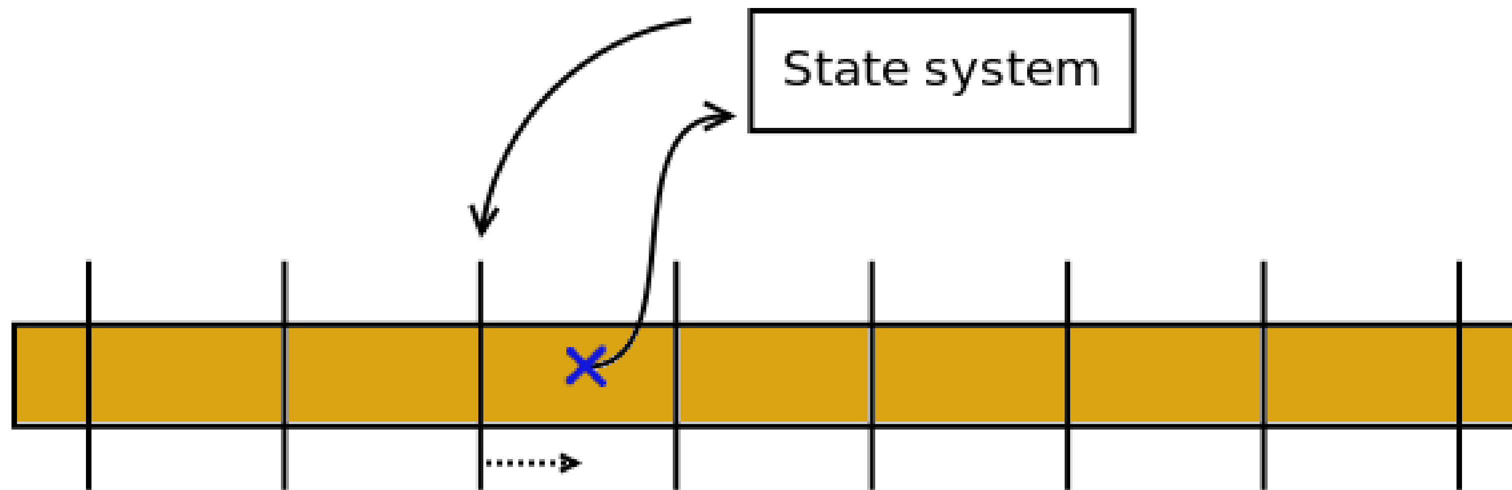
- The trace visualizers will query the state system and display relevant information.
- The traces contain series of *events*. Some of these events will modify the state:
  - Scheduling event
  - Process creation
  - File open
  - ...

The current method : Checkpoints



- From the initial state, play the events in order and update the state accordingly.
- Every *N* events, save the current state → Checkpoint (≈I-frame)

The current method : Checkpoints



- When queried about the state at time *t*, load the closest earlier checkpoint and replay the events up to *t*.

The current method : Checkpoints

- Shortcomings:
  - To keep the query time constant on average, we need to keep the number of events between checkpoints constant.
  - With larger traces (last longer, more events / second), we need to save more checkpoints.
  - With larger systems, each checkpoint contains more information.
  - There can be a lot of redundant information between checkpoints.
- The storage requirements may get out of control at some point.

The proposed State History Tree

- What if we used a temporal (interval-based) tree structure to store our state data?
- The storage requirements would stay within the same magnitude of the trace files.
- The query times would remain efficient (we should only explore one branch of the tree).
- The tree would be stored on disk.

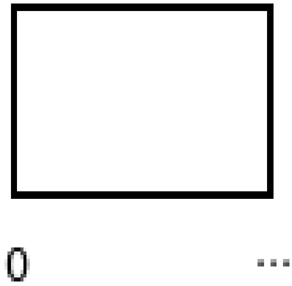
The proposed State History Tree

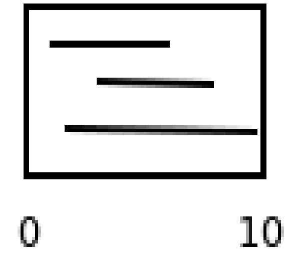
- Disk-based interval trees are not very common.
- “Pure” interval-based structures (segment-tree, interval-tree) are binary, thus more appropriate for memory storage.
- Most applications in the field use a database engine to store temporal data. (SQLite and Oracle both have R-Trees implementations.)
- In the literature, most interval-based trees work as query optimizers to sit on top of a database.[1]

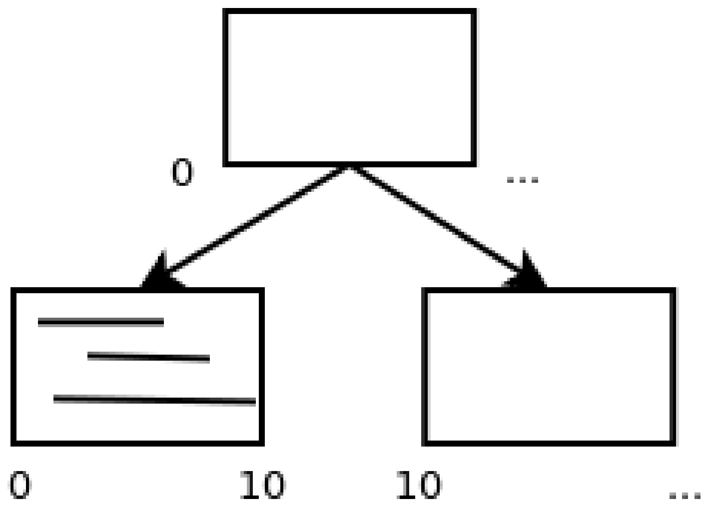
The proposed State History Tree

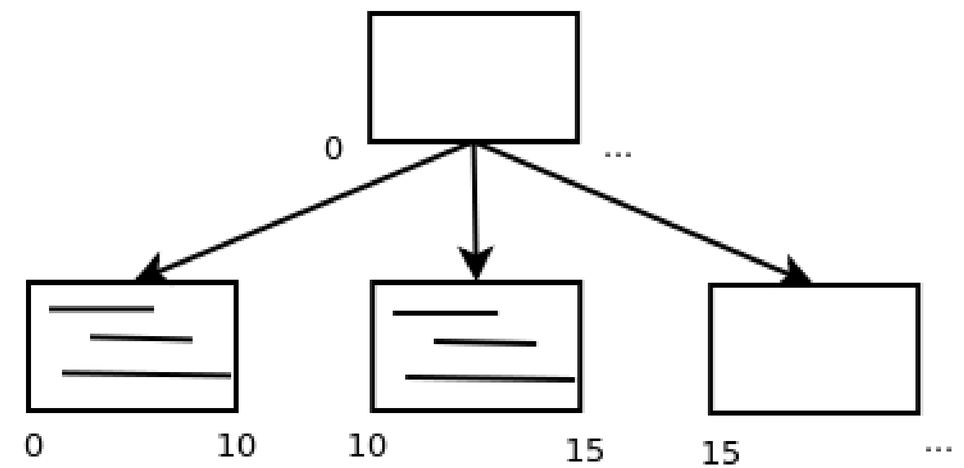
- The tree will deal with *intervals*, which are defined by a start time, an end time, as well as a key and a value of variable size.
- Each *node* is stored as a block in the tree-file, contains a header and a payload section.
- Two main parameters:
  - size of each block (multiple of 4 KB)
  - max. number of children per node

Construction of the tree

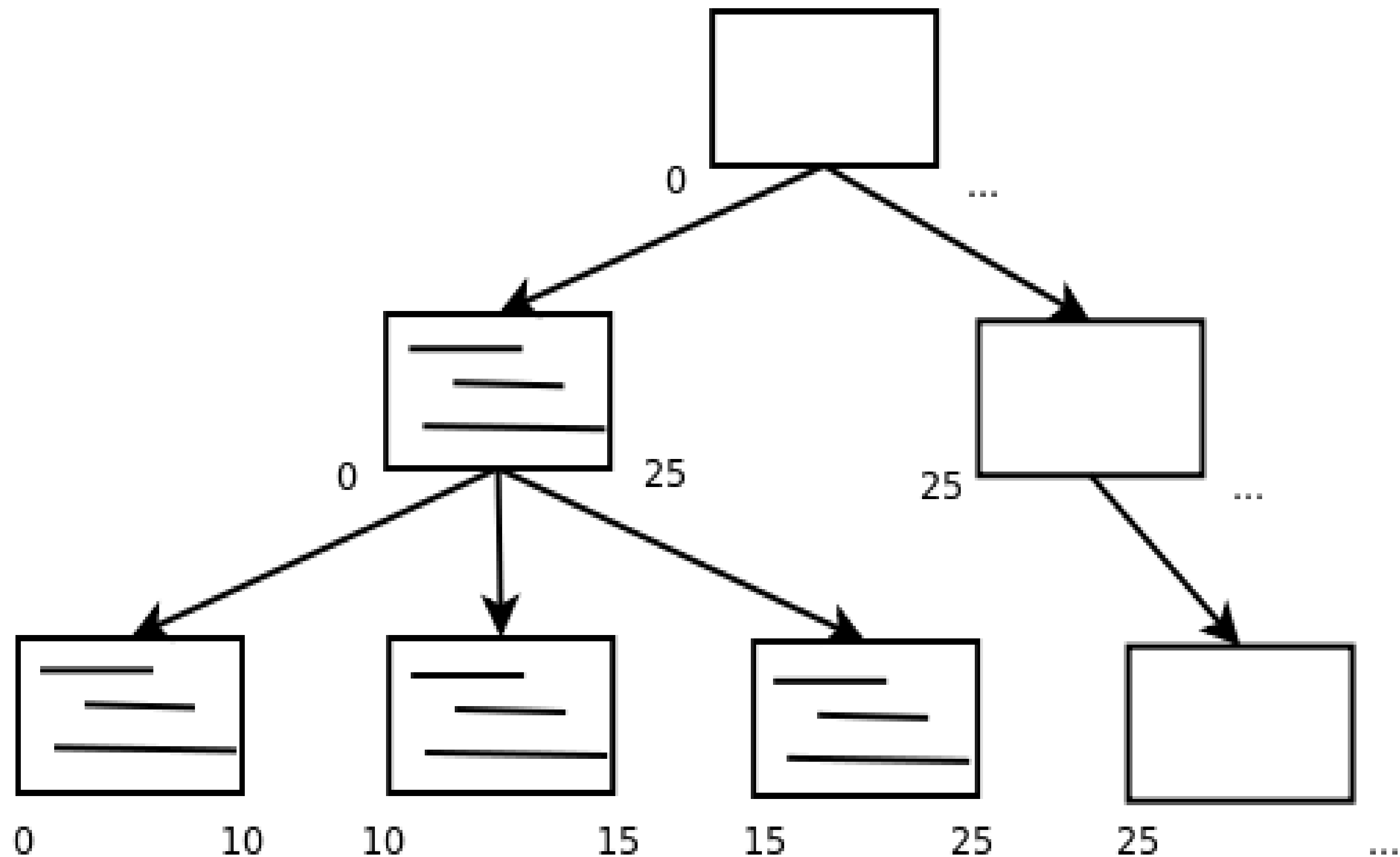




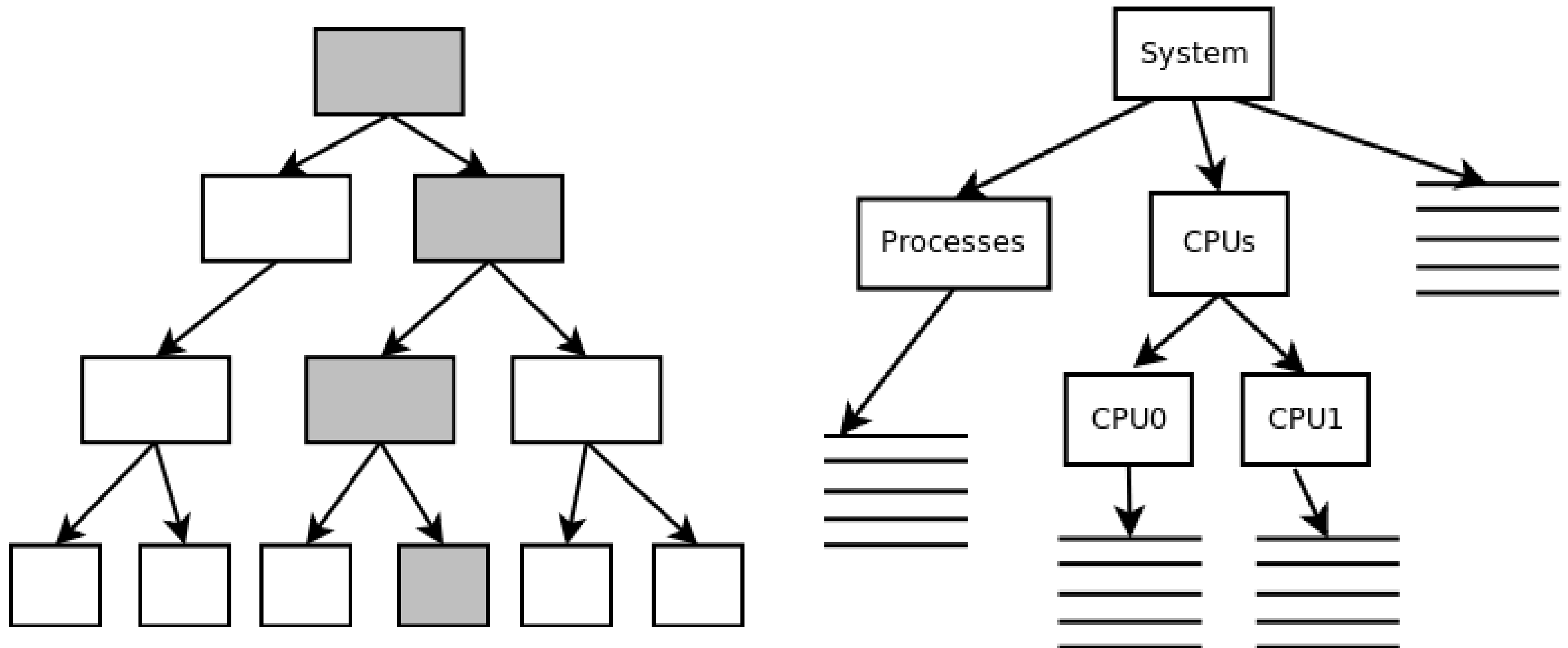




Construction of the tree

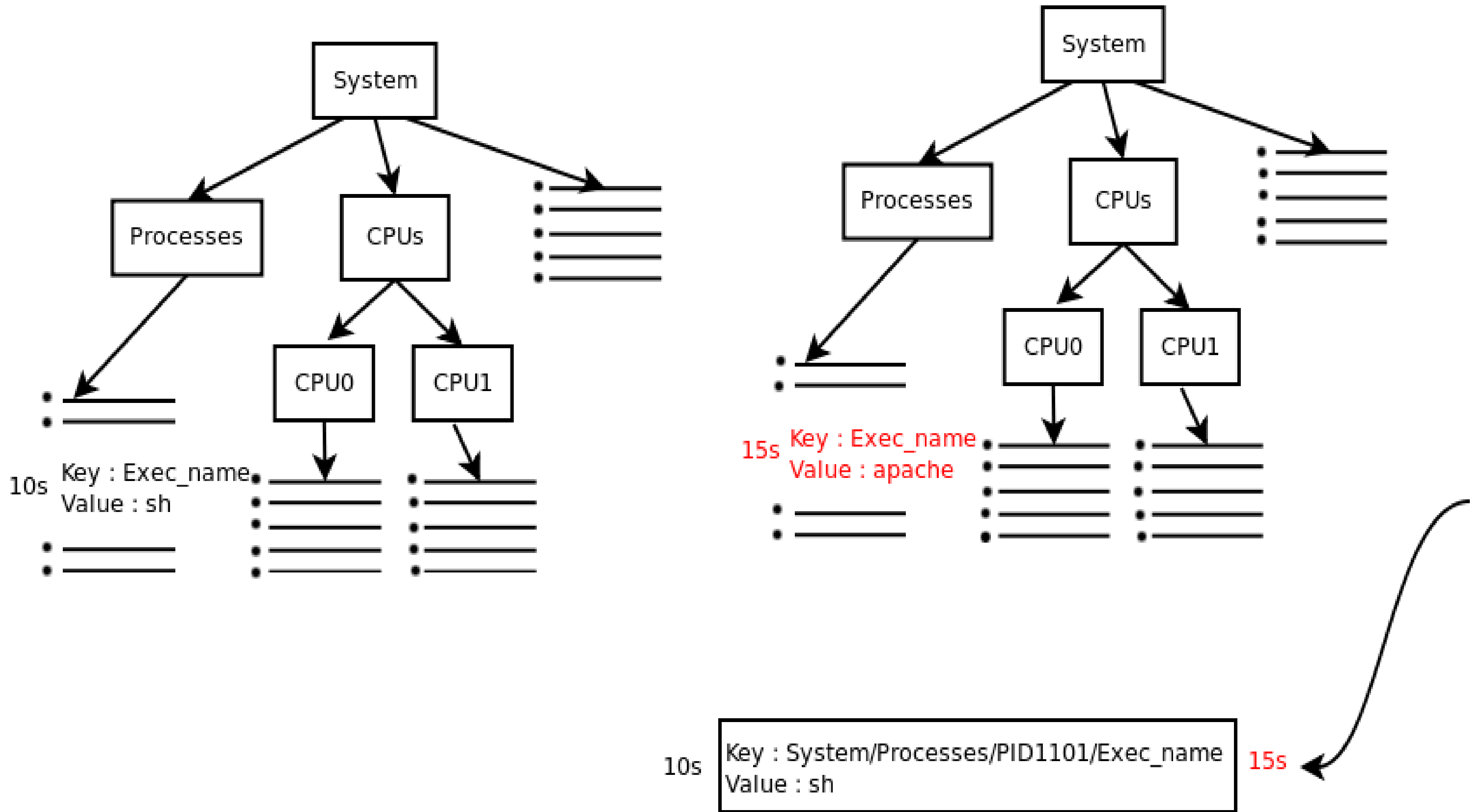


Queries



Integration

(how to build the tree from the events in the trace file)



Strong points

- The tree-file only needs to be built once, it can be re-used afterwards with the same trace.
- Built incrementally
  - Can be used as it's being built
  - Can be built in one go or in parts (limited resources)
  - Streaming

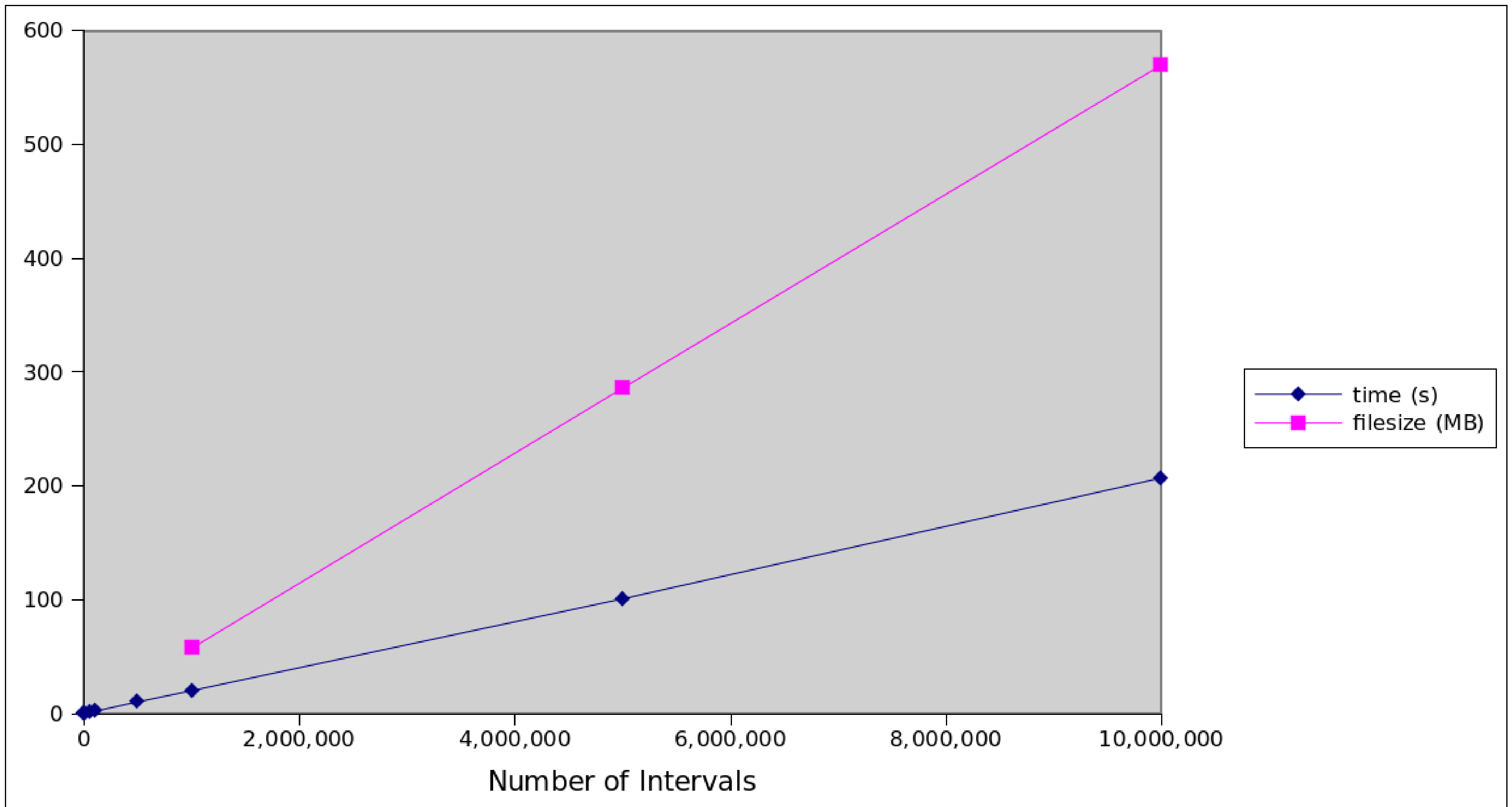
Limitations

- The average length of the intervals must be short compared to the total time range of the tree.
  (The tree could still handle it, but would degenerate into a simple list.)
- For optimal results, the intervals must be inserted in ascending order of their end times.
  (Could be worked around, ie a special insertion method that starts at the root node and works its way down.)

Preliminary test results

- Using the reference implementation in C and randomly-generated interval data.
- Building a tree with 10,000,000 intervals:
  - ~3 minutes
  - ~600 MiB
- An example trace of 10 millions events: ~1.8 GiB

Preliminary test results



The next steps

- Refocus the integration on TMF:
  - Re-implement the tree creation and query methods in Java.
  - Implement the interval-generating logic (the “glue” between the SHT and the state system) as an alternative code path.
- Test the tree behaviour thoroughly with data coming from real traces. Compare the performance with the checkpoints method.
- It is still quite early in the project. We could optimize or rework the design if deemed necessary (try new things, etc.)

References

[1] Kriegel H.P., Pötke M. and Seidl T., **Managing Intervals Efficiently in Object-Relational Databases**, Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

[2] Ang C.-H. and Tan K.-P., **The interval *B*-tree**, Information Processing Letters 53, pp. 85-89, 1995

[3] Arge L. and Vitter J.S., **Optimal External Memory Interval Management**, SIAM J. Comput Vol. 32 No. 6, pp. 1488-1508, 2003

Questions ?