Efficient methods for kernel trace analysis parallelization

Fabien Reumont-Locke
Under the supervision of Prof. Michel Dagenais
I. Introduction and research objectives
II. Adapting the tools to parallel processing
III. Parallelization methods
IV. Preliminary results
V. The road ahead and conclusion
Parallel computing

- Modern architectures allow higher and higher levels of parallelism
- Already used in a lot of areas: physics, mechanical engineering, rendering

For example, finite element analysis

Can we apply it to trace analysis?
Does the use of parallel computing methods allow for an acceleration of the analysis of kernel traces, which is both efficient and scalable?

The goal is to develop trace analysis parallelization methods that will:

a. Work for most existing analyses
b. Be efficient (provide considerable speedup over sequential methods)
c. Be scalable (improved performance as number of parallel units increases)
Challenges of parallelization

Load balancing

Data partitioning

Locking and synchronisation

Data dependencies

Source: https://computing.llnl.gov/tutorials/parallel_comp/
Choice of tools

- Trace Compass (formerly TMF) is the Eclipse-based tool for trace analyses and visualizations
  - Very complete framework, lots of infrastructure for reading traces, analysing them and displaying the results
  - Unfortunately, this also means lots of complexity, making it very hard to experiment with parallelization
- babeltrace is a C library that allows reading CTF traces
  - “Only” provides reading events from traces
  - The simpler design lends itself better to parallelization
Adapting babeltrace to parallel analysis

- Babeltrace allows only one iterator per trace
  - Temporary solution: create one context per thread, add trace to each context
  - Very long startup cost, since we have to parse metadata, create structures, etc., for each thread
  - Not viable when working with up to 64 cores
- Added support for multiple iterators per trace by cloning file streams inside each iterator
- Then there was also the problem of performance...
Use parallel processing to slow your application down!

Performance gradually worsens as we add more threads

Something is definitely wrong...

<table>
<thead>
<tr>
<th>Trace size</th>
<th>5,114,625 events</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU on analysis</td>
<td>AMD FX 9370</td>
</tr>
<tr>
<td>machine</td>
<td>Eight-Core Processor</td>
</tr>
</tbody>
</table>
What is happening?

```
In glib/gquark.c ->
```

```
perf record --call-graph
```

```
GQuark

G_quark_from_string (const gchar *string)
{
  GQuark quark;
  if (!string)
    return 0;
  G_LOCK (quark_global);
  quark = quark_from_string (string, TRUE);
  G_UNLOCK (quark_global);
  return quark;
}
```
After patching things up

- Added a thread-local quark cache to prevent global lock contention in the glib
- Each thread only queries the global glib quark hash table if it does not have the quark in its cache
- Duplicates data, but the tradeoff is worth it
Per-stream data partitioning

- One stream per CPU on the **analyzed** machine
- Each CPU on the **analyzing** machine treats one stream

What about if there are less streams than CPUs?
- Synchronization problems when causality between events (e.g. migrated process)

We need to split “vertically”, i.e. by time
Per-time range partitioning

- Split the trace evenly across streams by **timestamp**
- Each CPU analyzes all the events between two timestamps
- Load balancing: uneven event density
- Data dependency: some events are dependent on prior events
  - E.g. which system call just exited?

These two threads slow down the analysis

![Diagram showing time partitioning and CPU analysis](image-url)
Using CTF packet index to balance load

- CTF traces have a packet index that we can use to balance the load
- We assume that packet size is proportional to the number of events
- Other advantages:
  - Seeks on packet frontiers are cheaper than within packets
- Disadvantages:
  - Works best only if trace has a lot of packets (not always the case)
  - At the moment only works on single-stream traces
Breaking data dependencies

- Most analyses keep a running “current state” containing all the necessary data.
- This current state is also queried to know, for example, which system call was running.
- But what if we don’t know some of the current state?
- We rely on the fact that the unknown state lasts only until the next event is read:
  - `sys_* -> syscall`
  - `exit_syscall -> user`
State propagation

- Values dependent on unknown state are kept in each chunk’s context
  - e.g. unknown syscall, or syscall in unknown current thread
- State is propagated forward in time at the merge phase
- In terms of implementation, this simply means handling unknown state + adding an additional *merge* method to allow merging the contexts
Test analyses

Implemented some of the Python analyses made by Julien Desfossez
Results - Event counting

<table>
<thead>
<tr>
<th>Trace size</th>
<th>17,358,022 events</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU on analysis machine</td>
<td>4 x AMD Opteron 6272 Sixteen-Core Processor</td>
</tr>
<tr>
<td>Serial time</td>
<td>36.7 s</td>
</tr>
<tr>
<td>Parallel time (best)</td>
<td>4.5 s w/ 26 threads</td>
</tr>
</tbody>
</table>

Count the number of events in a trace

Maximal acceleration: ~8.2x

Worst-case scenario: Lots of I/O for little CPU work
Results - CPU analysis

<table>
<thead>
<tr>
<th>Trace size</th>
<th>44,001,071 events</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU on analysis machine</td>
<td>4 x AMD Opteron 6272 Sixteen-Core Processor</td>
</tr>
<tr>
<td>Serial time</td>
<td>66.3 s</td>
</tr>
<tr>
<td>Parallel time (best)</td>
<td><strong>7.4 s</strong> w/ 32 cores</td>
</tr>
</tbody>
</table>

Measure the proportion of CPU active vs idle + CPU usage per thread

Maximal acceleration: ~9x

Only uses scheduling events (*sched_switch*)

Very simple analysis
Results - IO analysis

<table>
<thead>
<tr>
<th>Trace size</th>
<th>17,358,022 events</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU on analysis machine</td>
<td>4 x AMD Opteron 6272 Sixteen-Core Processor</td>
</tr>
<tr>
<td>Serial time</td>
<td>80.3 s</td>
</tr>
<tr>
<td>Parallel time (best)</td>
<td>6.0 s</td>
</tr>
</tbody>
</table>

Measure syscall read/write I/O per thread

Maximal acceleration: ~13.3x

Uses I/O syscalls (sys_read, sys_write, sys_splice, etc.)

Slight increase in complexity brings much better scaling
The road ahead

- **Short-term goals**
  - Apply to more types of analyses, such as current state, memory
  - Better load balancing through a hybrid per-stream/per-time range partitioning
- **Medium-term goals**
  - Add support for parallelizing the XML state system analysis
  - Output into State History Tree
- **Long-term goals**
  - Distributed analysis
  - Live tracing analysis
Conclusion

Research question:
*Does the use of parallel computing methods allow for an acceleration of the analysis of kernel traces, which is both efficient and scalable?*

The preliminary results seem to indicate that parallel processing is a viable way to achieve better, more scalable performances for the analysis of large traces.
One more thing...
Thank you!
Questions?