A NUMA-AWARE RUNTIME ENVIRONMENT FOR THE ACTOR MODEL

Emilio Francesquini
emilio@ime.usp.br  
Alfredo Goldman
gold@ime.usp.br
Jean-François Méhaut
jean-francois.mehaut@imag.fr

University of São Paulo
University of Grenoble
## Some Actor Libraries

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>Latest release</th>
<th>License</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>F# MailboxProcessor ⚙️</td>
<td>Active</td>
<td>same as F# (builtin core library)</td>
<td>Apache License</td>
<td>F#</td>
</tr>
<tr>
<td>Ateji PX ⚙️</td>
<td>Active</td>
<td>?</td>
<td>?</td>
<td>Java</td>
</tr>
<tr>
<td>Actor Framework ⚙️</td>
<td>Active</td>
<td>2013-09-02</td>
<td>Apache 2.0</td>
<td>.NET</td>
</tr>
<tr>
<td>Quasar ⚙️</td>
<td>Active</td>
<td>2013-07-19 [48]</td>
<td>LGPL/Eclipse</td>
<td>Java</td>
</tr>
<tr>
<td>Pulsar ⚙️</td>
<td>Active</td>
<td>2013-07-19 [44]</td>
<td>LGPL/Eclipse</td>
<td>Clojure</td>
</tr>
<tr>
<td>Akka ⚙️</td>
<td>Active</td>
<td>2013-07-09</td>
<td>Apache 2.0</td>
<td>Java and Scala</td>
</tr>
<tr>
<td>Pulsar ⚙️</td>
<td>Active</td>
<td>2013-06-30</td>
<td>New BSD</td>
<td>Python</td>
</tr>
<tr>
<td>Actor ⚙️</td>
<td>Active</td>
<td>2013-05-30</td>
<td>MIT</td>
<td>Java</td>
</tr>
<tr>
<td>JActor ⚙️</td>
<td>Active</td>
<td>2013-01-22</td>
<td>LGPL</td>
<td>Java</td>
</tr>
<tr>
<td>GPars ⚙️</td>
<td>Active</td>
<td>2012-12-19</td>
<td>Apache 2.0</td>
<td>Groovy</td>
</tr>
<tr>
<td>Pykka ⚙️</td>
<td>Active</td>
<td>2012-12-12 [45]</td>
<td>Apache 2.0</td>
<td>Python</td>
</tr>
<tr>
<td>Cloud Haskell ⚙️</td>
<td>Active</td>
<td>2012-11-07 [41]</td>
<td>BSD</td>
<td>Haskell</td>
</tr>
<tr>
<td>QP frameworks for real-time embedded systems</td>
<td>Active</td>
<td>2012-09-07 [54]</td>
<td>GPL 2.0 and commercial (dual licensing)</td>
<td>C and C++</td>
</tr>
<tr>
<td>libcpp ⚙️</td>
<td>Active</td>
<td>2012-08-22 [51]</td>
<td>LGPL 3.0</td>
<td>C++</td>
</tr>
<tr>
<td>Theron ⚙️</td>
<td>Active</td>
<td>2012-08-20 [46]</td>
<td>MIT [47]</td>
<td>C++</td>
</tr>
<tr>
<td>Celluloid ⚙️</td>
<td>Active</td>
<td>2012-07-17 [52]</td>
<td>MIT</td>
<td>Ruby</td>
</tr>
<tr>
<td>libprocess ⚙️</td>
<td>Active</td>
<td>2012-07-13</td>
<td>Apache 2.0</td>
<td>C++</td>
</tr>
<tr>
<td>Actor-CPP ⚙️</td>
<td>Active</td>
<td>2012-03-10 [49]</td>
<td>GPL 2.0</td>
<td>C++</td>
</tr>
<tr>
<td>LabVIEW Actor Framework ⚙️</td>
<td>Active</td>
<td>2012-03-01 [53]</td>
<td>?</td>
<td>LabVIEW</td>
</tr>
<tr>
<td>NAct ⚙️</td>
<td>Active</td>
<td>2012-02-28</td>
<td>LGPL 3.0</td>
<td>.NET</td>
</tr>
<tr>
<td>Jetlang ⚙️</td>
<td>Active</td>
<td>2012-02-14 [43]</td>
<td>New BSD</td>
<td>Java</td>
</tr>
<tr>
<td>S4 ⚙️</td>
<td>Active</td>
<td>2011-11-28 [50]</td>
<td>Apache 2.0</td>
<td>Java</td>
</tr>
<tr>
<td>Killm ⚙️ [38]</td>
<td>Active</td>
<td>2011-10-13 [39]</td>
<td>MIT</td>
<td>Java</td>
</tr>
<tr>
<td>ActorKit ⚙️</td>
<td>Active</td>
<td>2011-09-14 [40]</td>
<td>BSD</td>
<td>Objective-C</td>
</tr>
<tr>
<td>Korus ⚙️</td>
<td>Active</td>
<td>2010-02-01</td>
<td>GNU GPL 3</td>
<td>Java</td>
</tr>
<tr>
<td>Termite Scheme ⚙️</td>
<td>Active?</td>
<td>2009</td>
<td>LGPL</td>
<td>Scheme (Gambit implementation)</td>
</tr>
<tr>
<td>Libactor ⚙️</td>
<td>Active?</td>
<td>2009</td>
<td>GPL 2.0</td>
<td>C</td>
</tr>
<tr>
<td>ActorFoundry (based on Killm) ⚙️</td>
<td>Active?</td>
<td>2008-12-28</td>
<td>?</td>
<td>Java</td>
</tr>
<tr>
<td>Haskell-Actor ⚙️</td>
<td>Active?</td>
<td>2008</td>
<td>New BSD</td>
<td>Haskell</td>
</tr>
</tbody>
</table>

The Actor Model

- Why?
  - No shared memory
  - Simple message passing interface
  - Integrated to some languages and runtime-environments
    - Erlang
    - Scala
  - Also available as library for many other languages

- Current runtime-environments
  - Highly optimized for SMP machines
  - However, none of the current mainstream runtime-environments provides support for NUMA platforms
Presentation Outline

• Briefly explain actor model-based application characteristics
• Outline the design choices available for a runtime-environment
• Case Study: The Erlang VM
• Experimental results
• Conclusion and future work
Communication Graph

- Actors communication graph during the execution of the YCSB against CouchDB on a 24-core machine
- Although not simple, there is a clearly defined structure of communication throughout the application's lifetime
  - MapReduce, Rings, Trees, Pipelines, ...
- Clearly defined hub-actors and affinity groups
- The communication graph is extremely dynamic
Design Choices

• Thread-based
  • Limits what can be observed and acted upon, the OS makes most of the decisions
  • High-level RE information is not visible to the OS scheduler

• Event-based
  • Single run-queues
  • Multiple run-queues
Single Run-Queue

• Bottleneck for the overall system performance
• No soft-affinity

Case Study: The Erlang VM

- Event-based
  - However, schedulers are not bound by default
- Migration Logic, which is composed of two strategies
  - Both based on the size of the queues
  - Load-balancing
  - Work-stealing
- Initial placement
- Compaction of load
A NUMA-Aware Approach

• NUMA platforms challenge not only actor model REs but also any concurrent application
• We want to keep, at the same time, the hardware isolation provided by the high-level runtime-environment and a good performance
• Developers usually have good insights into the behavior of the application, so they can provide hints on possible hubs in the code
• Two main aspects of the runtime environment need to be taken into consideration
  • Load-balancing policies
  • Actor affinity maintenance
Load-balancing vs. Actor Affinity

• Load-balancing tries to keep every available scheduler busy and at the same time provide each actor a fair share of the Pus

• Actor affinity maintenance tries to keep actors, and their affinity group, close together so that there is a fast communication between them

• Sometimes these goals may conflict
  • We are after good trade-offs in terms of performance
First Part: Initial Actor Placement

• The hub heuristic gives us the insight that hubs are among the biggest spawners of the application
  • Therefore, it makes sense to spread them throughout the schedulers, or NUMA nodes, and not place them together

• On the other hand, regular actors are likely to communicate within their affinity group
  • And by the same heuristic, this group is typically composed of the hub and the actors it spawned

• We propose two distinct initial placement policies, depending if the actor is a hub
Second Part: Hierarchical Load-Balancing and Work-Stealing

• On a NUMA platform
  • Migration costs depend on the origin and the destination
  • The chosen destination alters the performance of the system (NUMA factor)
    • Remote heap
    • Message exchange costs
    • I/O – Each device is connected to only one node

• We introduce the notion of home node
  • It is the NUMA node in which the heap of the actor is¹
  • Both load-balancing and work-stealing will try to
    • Bring actors back home
    • Only if that is not enough considers remaining actors²

¹ CCEM'12 - P. Saxena and V. Srinivasan, Optimizing virtual machine resource placement on multi-socket platforms.
² ICPP'12 - L.L. Pilla et al., A hierarchical approach for load balancing on parallel multi-core systems.
Experimental Results

• Modified Erlang VM based on version R15B02
  • C LoC - 300K+ C
  • Erlang LoC - 65K+
• Linux specific
• Most important VM modifications
  • Hub flagging
  • Home node tracking
  • Memory (actor heap) allocation/migration
  • Initial placement
  • Actor migration
  • Tracing
• Benchmarks from the BenchErl benchmark suite
  • CPU bound and Erlang API specific benchmarks where removed
    • We are interested in benchmarks where communication and placement of actors play an important role
  • We slightly modified the benchmarks to hint the possible hubs
## Evaluated Platform

<table>
<thead>
<tr>
<th></th>
<th>NUMA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMA Nodes</td>
<td>4</td>
</tr>
<tr>
<td>NUMA Topology</td>
<td>Full Graph</td>
</tr>
<tr>
<td>Cores</td>
<td>32</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.27 GHz</td>
</tr>
<tr>
<td>Total RAM</td>
<td>64 GiB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>24 MiB</td>
</tr>
<tr>
<td>NUMA Factor</td>
<td>1.2 to 3.6</td>
</tr>
<tr>
<td>Linux Kernel</td>
<td>3.5.7</td>
</tr>
<tr>
<td>GCC</td>
<td>4.7.2</td>
</tr>
</tbody>
</table>
Experimental Results: NUMA32

- Normalized execution time of the benchmarks for two different data input sizes on the NUMA32 platform

- Better initial placement and reduced costs of communication are the responsible for the observed performance gains

- Single actors bottlenecks (only one hub) or no hubs are the responsible for the observed performance losses, since we end up introducing overheads we cannot compensate
Experimental Results: NUMA32 – Vanilla VM

- Normalized execution time of the BenchErl benchmarks using the best tuning for the vanilla and modified VMs on the NUMA32 platform.

- Execution times were normalized by the vanilla VM execution time with no optional parameters.

- Benchmarks were executed, at least 30 times each, using intermediate and short configurations.

Migrations

- Average number of migrations for the intermediate data set
- The reduced number of migrations is due, mainly, to a better initial placement of the actors
Policy Impact on Avg. Execution Time

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>bang</th>
<th>big</th>
<th>ehb</th>
<th>orbit_int</th>
<th>serialmsg</th>
<th>timer_wheel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler Binding</td>
<td>35.1%</td>
<td>28.7%</td>
<td>10.5%</td>
<td>53.3%</td>
<td>-13.4%</td>
<td>-81.4%</td>
</tr>
<tr>
<td>Compaction of Load</td>
<td>-0.3%</td>
<td>0.2%</td>
<td>-0.6%</td>
<td>0.8%</td>
<td>0.1%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>Initial Placement</td>
<td>20.5%</td>
<td>38.9%</td>
<td>20.5%</td>
<td>64.1%</td>
<td>-8.8%</td>
<td>-73.3%</td>
</tr>
<tr>
<td>OnlyHubs</td>
<td>16.9%</td>
<td>21.4%</td>
<td>5.8%</td>
<td>43.3%</td>
<td>2.9%</td>
<td>-22.0%</td>
</tr>
<tr>
<td>Hierarchical Work-Stealing</td>
<td>9.6%</td>
<td>18.1%</td>
<td>9.8%</td>
<td>43.2%</td>
<td>-5.4%</td>
<td>-37.6%</td>
</tr>
</tbody>
</table>
Conclusion and Future Work

• We presented how some common application characteristics can be used to impact the performance of applications on NUMA platforms

• We still rely too much on hints to determine the hubs and on a weak heuristic to define an actor’s affinity group

• Better mapping of the underlying platform
Thank you!

Emilio Francesquini
emilio@ime.usp.br