Analyzing Performance of Multicore Applications in Erlang

Kim J.L. Nevelsteen
Abstract

Analyzing Performance of Multicore Applications in Erlang

Kim J.L. Nevelsteen

The advent of multicore architectures has bred complexity for developers. Variability in scheduling and resource management makes predicting runtime behavior substantially more complex. In order to cope with the added complexity, developers need tools that grant transparency in the system. Usually, a number of key characteristics heavily determine the performance of an application. By building a framework that exposes these key characteristics to the developer, we empower him/her to more closely monitor the system and gain important insight into its behavior.

Erlang is a concurrent functional language, especially tailored for distributed and fault tolerant applications. Because, multiprocessing and asynchronous message passing have long been supported as language constructs, a considerable trace library and multiprocessing system tools have also long been available. This makes Erlang well suited as a modeling platform to build the mentioned framework. We do this in three steps; by first learning the Erlang trace system and its virtual machine, building a framework and thereafter test the framework.
Acknowledgements
Sincere gratitude to my supervisor Bengt Jonsson for interesting discussions, his understanding during tough times and simply to be given the chance to be part of his team.

I must also extend thanks to Sverker Eriksson at Ericsson for his help during development in the hot summer months. Without his knowledge into the intricate details of the Erlang VM, I would not have been able to complete this thesis.
1 A move toward multicore platforms

1.1 Monitoring and key characteristics

In development and maintenance of software for multicore platforms, there are many degrees of freedom and many designs decisions to be made, e.g., concerning how the application is structured into computational units, how they are configured and mapped onto available hardware, how dynamic scheduling and resource management is handled, etc. This variability makes multicore system development a much less straightforward and more unpredictable activity than for single-core platforms. To assist developers, methods and tools should be developed, that allow them to make qualified decisions in system design, and to predict the effect of available design alternatives. Such methods and tools should be able to answer questions such as

- What will be the effect on average performance and guaranteed timing constraints when mapping computational units onto cores, and of different dynamic scheduling strategies?
- What speedup can be expected when replacing a previously sequential algorithm by a new parallel one, on a specific platform?
- How much new functionality can be allowed due to migrating to a more powerful multicore hardware platform?
- How can I adapt my design and implementation to achieve better performance. Where are possibilities for performance improvements, and how can performance bottlenecks be found?

An important basis for all the problems described in the previous section is to be able to measure and model, with some precision, various key characteristics that affect performance. These key characteristics include execution time, memory footprint, communication behavior, access to shared data structures, etc. A suitable modeling framework should reflect how performance, timing, computational efficiency, and resource utilization are related in a component, and how they affect other computational components in the application.

Erlang is a concurrent functional language, especially tailored for distributed and fault tolerant software, e.g., in telecommunication applications [Armstrong et al.][Arm07]. Prominent features of Erlang include support for light-weight processes, asynchronous message passing, and fault handling as integral parts of the language. These features make Erlang into an especially interesting programming language for multicore platforms. Support for performance monitoring the many light-weight Erlang processes is already well developed through the tracing library and system tools, but now these tools must also be expanded for recent multicore platforms. This means the current set of tracing information must be expanded to include which component the trace information originated from and the correlations with other components. Where the goal of Erlang was to intricately hide details of multiprocessing from programmer for ease of use, it is actually desirable to monitor exact scheduling and execution path relative to hardware cores.

Erlang lends well as the modeling platform we seek. This, in part, because of the native support for multicore processing, the tracing system to measure
key characteristics and also partly because our industrial partner, Ericsson, identified some of the described problems as present in the Erlang environment. Our approach is then to . . .

- Learn the Erlang trace system, assess how much support exists for multicore architectures and how we can tie into the Erlang virtual machine in order to extend it;
- Build a framework where we extract characteristic metrics from the Erlang system that can be used for monitoring and collect them into a structure accessible to the programmer. Setup trace handlers which will post-process the collected metrics with the help of a programmer supplied heuristic function into a gauge for system performance;
- And, utilize the framework to test a system under test (SUT), in an attempt to find a correlation between some of the characteristic metrics and performance.

2 Description of the Erlang runtime and tracing

2.1 Erlang runtime executed on multicore architectures

Erlang is implemented as a virtual machine (VM) programmed in the language C. Erlang was conceived as a concurrent paradigm with light-weight processes, which would be serialized on a single core hardware architecture. With the advent of multicore into the mainstream market, Erlang is being moved toward these new architectures by implementing multiple schedulers/run queues, migration logic, etc... [Erl08]

Because light-weight processes and message passing were already present in Erlang the transition to multiprocessing proves less complicated than that for traditional languages, for example, such as C. Erlang programmers were already in possession of the needed skills to parallelize their programs with easy built-in keywords and constructs. This passes most of the burden of adapting to multicore, such as dealing with caching and race conditions, to the makers of Erlang and the implementation of the Erlang VM.

To effectively parallelize code in Erlang a programmer calls one single built-in function (BIF) to create a spawned process and uses a message passing language construct to transfer results to other processes. Spawned processes are typically spawned on the same run queue as the parent, but can be swapped to other run queues by work stealing.

Message passing in the Erlang system is asynchronous. To be entirely accurate, it must be noted that some synchronization occurs behind the scenes. Asynchronous is true because the receiving process handles messages in its mailbox according to its own specified priority. How synchronous the operation is, is highly dependent on the operating system. The hidden background synchronization is the operation of getting the message into the mailbox of the receiver, namely requesting the receiver to pause long enough to copy the message to the receiver's memory area.
By default, Erlang is a multi-tiered system. Without specific configuration, Erlang creates a run queue for each logical processing unit (or core) in the hardware which in turn is implemented using one POSIX pthread on the OS level. The Erlang VM schedules its processes to Erlang run queues while the OS is scheduling the pthreads to OS level cores. This means that not only are individual processes swapped out and rotated to the back of the run queue on the Erlang level, but also the entire run queue can also undergo a context switch or be swapped to another core on the OS level at any time. The Erlang VM would report the Erlang process as continuously running when in reality it has been swapped out by the OS. Monitoring performance of such a multi-tiered system therefore becomes the challenge of monitoring processing each level of the system. Erlang processes typically are scheduled on the same run queue as the parent process and are potentially swapped by a work stealing algorithm to equalize load. The affinity of POSIX pthreads to cores is entirely under influence of the scheduling algorithm of the selected OS executing the Erlang VM; in our case Solaris. (It is highly overambitious for this thesis to attempt to simultaneously monitor both Erlang and the underlying OS.)

An Erlang process is allowed to execute for a time slice of approximately 2000 reductions before a context switch occurs and the process is placed at the end of the run queue. Each reduction corresponds to the execution of one BIF in the Erlang VM. However, the length of a BIF is variable depending on its task i.e., calling a BIF executes a variable amount of machine level instructions depending on how complex the function is. This means that a single reduction is not a constant value. The measure is, however, used throughout process and system information reports produced by the system and therefore we adopt its use as well.

2.2 Details of the existing Erlang trace system

In order to activate the Erlang trace system, the programmer makes a call to the tracer module and passes as a parameter which processes should be traced, together with a number of flags which instruct the system of the desired behavior. The following is a brief list of some of the flags available [Eri10].

s (send) : Traces the messages the process sends.

r (receive) : Traces the messages the process receives.

m (messages) : Traces the messages the process receives and sends.

c (call) : Traces global function calls for the process according to the trace patterns set in the system.

p (procs) : Traces process related events to the process.

sos (set on spawn) : Lets all processes created by the traced process inherit the trace flags of the traced process.

sofs (set on first spawn) : This is the same as sos, but only for the first process spawned by the traced process.
The flags mentioned are important to determine the behavior of the tracing system. If one does not flag a set of trace messages, no trace messages will be produced for that set. In addition to the flags there is also a boolean flag which alters trace message output. It specifies whether timestamps should be appended to each of every trace message reflecting when the trace message was generated. We will most certainly set this boolean to \texttt{true} in this thesis.

In order to obtain tracing specific to certain specific function calls, we must pass the Erlang VM a function mask which is used to identify the functions we are interested in tracing. After specifying this mask, the Erlang VM will send a trace message for each function call/return matching the given mask.

Upon activation of the trace system, Erlang creates a port or process which is dedicated to receive trace messages from all of the traced processes. Trace messages are flushed to disk immediately to be processed later. All are read from disk after the completion of the trace and passed to the trace handler. The programmer must assign a function that will serve as this handler. Erlang function overloading can easily be employed to separate the assigned generic handler into a specific handler for each type of trace message. Each trace message contains a certain amount of characteristic information depending on its type, \textit{e.g.}, a trace message signifying a received message will contain the sender \texttt{PID} also.

It is important to note that the trace system assumes processing of trace data after a trace session. Erlang does not support the continuous tracing of a running system with constant processing, \textit{i.e.}, assigning a certain amount of processing time or cores to continually monitoring a running system.

## 3 Framework for tracing multicore executions

### 3.1 Obtaining metrics and gauging performance

We attempt to create a generic framework to extract as many different metrics from a SUT as possible, so as to promote the analysis of multicore execution. The resulting framework consists of an Erlang script and a modified Erlang VM. The Erlang script initializes the trace environment for a SUT and defines routines that post-process the trace output. It needs to be noted that the definitions for routines handling the trace output are modified versions of those specified in the trace system. The modifications to the routine definitions coincide with the modifications that were made to the Erlang VM.

If we are determining the execution behavior of a particular SUT and we must modify that SUT in order to obtain certain metrics, then we are potentially modifying the behavior of the SUT. For example, we can write scripts that alter the source code of the SUT to get additional metrics, but this does, however so slightly, alter the execution path of the SUT. So, ideally, we should measure a SUT with non-invasive techniques unless entirely unavoidable.

To be able to perform some type of analysis, we must be able to to gauge performance. This performance can be gauged in different ways, so we require the programmer to specify a heuristic which takes the supplied framework metrics, processes them and outputs a gauged result. Specifying an accurate heuristic is in itself not trivial. The domain of possible combinations of metrics which influence SUT behavior is very broad indeed; one must decide what is a correct
gauge of system performance in general i.e., overall execution time, throughput, et cetera. And, to speak of even more complexity, we also have to consider programs in which the execution path depends heavily on input data, making it far more difficult to perform predictions or find patterns of execution.

We’ve spoken repeatedly about performance, but now we must stop and deliberate on what performance, in the context of the Erlang environment, means and how we measure it. Reduction count is used throughout Erlang and this thesis, but we must question whether reduction count can serve as a unit of measure for performance. Unfortunately, we must conclude that the reduction count is too crude to be considered a unit of measure, specifically because each BIF consists of a different number of machine language instructions. We do, however, remain satisfied using reductions to measure performance and leave calculations at a more granular level as an extension to this thesis seeing as this requires intricate knowledge of the Erlang VM combined with that of the underlying OS.

3.2 Problem domain and limitations of Erlang tracing

Erlang’s tracing system is quite advanced, but it lacks support for monitoring a multicore system. We aimed to augment the existing trace system to incorporate the monitoring of a multicore system so that the exact execute behavior of the SUT would be clearer.

In order to extend the trace system with support for monitoring a multicore system and obtain certain metrics, we required information that was not readily available through the existing trace system. Most of the information we sought, however, was present in the Erlang VM. So, we needed to alter the VM to pass along this additional information through the trace system in an efficient manner. (It should be noted that learning the VM implementation was a large exercise in itself and outside the scope of this thesis.) One of the major additional features that was released in the Erlang VM for multicore support was the implementation of multiple run queues, one per core. And because of additional feature, the number of the run queue a process is running on is a major factor in determining the runtime behavior of a process. We altered the Erlang VM, having it pass the run queue number along with the normal in/out trace message payload. We did the like for reduction count, altering the Erlang VM implementation so that it passes the current reduction count along with each trace message pertaining to the in/out and call/return traces. Reduction count is an important metric in performance measuring, so this metric was needed at regular intervals.

The Erlang system has process_info and system_info function calls which a programmer can call to obtain a wide range of statistics about the running system. But, during trace this is problematic at best, because of the inability of the trace system to pass process or system information along with trace messages. This is a definite hurdle towards obtaining pertinent monitoring information. To exemplify, lets say a programmer would like obtain the current reduction count at the exact moment a particular function call returned. It is currently possible to tell the trace system to monitor the return of a function call and send a trace message on that event. Since, it is not possible to ask the trace system to send the current system statistics along with the trace message, the earliest moment the trace system would have to call the process_info and
system_info function calls would be as soon as the trace message arrived at the trace process or port. But, the statistics would not coincide with that particular event. It would be late. Making the obtained information disconnected and rather useless. The functionality that we are looking for is the option to set a flag on certain trace messages instructing the VM to pass process or system information statistics along with those types of trace messages. We would then obtain accurate statistics relative to the sent trace messages. An added benefit of obtaining system statistics in this manner would be that it is completely non-invasive to the SUT i.e., we would not have to change the code of the SUT to obtain the information, but just add instructions to the trace system.

We have previously mentioned a multi-tiered system comprising of Erlang and the underlying OS. Concretely, we can trace the Erlang level, but lots of magic is happening on the OS level and Erlang deliberately obscures access to hardware statistics in order to obtain platform independence. This make any assessment of the system as a whole rather impossible. In order to correctly assess the behavior of a running Erlang system, we must reduce this complexity by reducing the rescheduling behavior on the OS level to a minimum. The only technique we currently have is to statically map each pthread (representing a run queue) to a single hardware core. We are then guaranteed that pthreads will not be swapped to another core, but this, however, does not remove the possibility for the OS to swap out the pthread in order to service some other OS process temporarily. To correctly monitor the system entirely we would need to simultaneously monitor Erlang and the underlying OS behavior. With even just considering timing on the Erlang system paired with timing on the hardware level, we have enormous complexity. We then also must consider that Erlang can be run on several different operating systems.

For simplicity, Erlang hides from the programmer the scheduling of processes to run queues. Before Erlang had multiple run queues, the VM was responsible for scheduling the processes to the single run queue and swapping out the processes to provide time slices to each process. Since Erlang has multiple run queues now, the VM continues to take the burden of scheduling. The same language constructs free the programmer from having to schedule processes to run queues. But, no language constructs have been provided to give the programmer to manually control which run queue a process is scheduled on. If we are to test certain scheduling schemes for processes in an attempt to find greater efficiency, then we must be able to control where a process is scheduled and also command the process to switch run queues if needed.

As a last obstacle, we must mention that the auto-numbering of processes according to an auto-increment strategy is a very poor labeling system for processes, if one requires unique PIDs for system analysis. To be able to do a comparative analysis between two different executions of the same SUT, we must be able to uniquely identify processes spawned during execution. Currently, the first and consecutive processes of the second execution are numbered starting where the first execution stopped. This mechanism makes it extremely difficult to uniquely identify the same single process in two different executions. This is especially true for processes spawned during program execution. We could potentially restart the Erlang environment between runs, but there is still no guarantee that PIDs will coincide between executions.
3.3 Data collection and post-processing

After configuring the trace system, trace data is generated by the Erlang VM at appropriate times and passed to the process or port responsible for collecting the trace, where it is then dumped to storage. A programmer can assign a function handler to accept and process the trace data after the trace has ended. We did precisely this, but as previously mentioned, we altered the Erlang VM which resulted in the trace data being slightly different than that specified in the Erlang trace documentation. In the post-processing, we create a data structure to collect intermediate results which a programmer specified heuristic will later then further process towards the desired performance gauge.

To summarize, we present here a list of intermediate metrics that are calculated through the combination of the trace system and our extensions.

- The run queue a process is running on at each context session,
- reduction cost for each run time slice,
- total time for each run time slice and how long it was since the previous context switch,
- count of the number of context switches and traced function calls,
- tree of spawned process,
- number of messages received by a process and who the sender was, and,
- total overall number of reductions and time for a single run,

can all be calculated from . . .

- the run queue number, reduction count and timestamp at each context switch,
- the reduction count and timestamp at each traced function call and return,
- trace of spawned processes and messages received,
- and, end_of_trace signal.

3.4 Trace handlers and intermediate data

Erlang tracing allows the programmer to assign a custom trace handler for each type of trace event. In Listing 1, the print/4 function is assigned as the trace handler delegating each trace event to one of the overloaded trhandler/3 functions. Line 6 of the listing shows modified function arguments to parallel the modified Erlang VM. The RunQ argument that has been added to this trace event and it holds the number of the run queue this context switch is being processed on. The other trace handlers, for namely out, call, unregister, link, spawn, send, receive, waiting and exit, are further below in the listing.
print(Fd, end_of_trace, TraceInfo, State) ->
    post_processing(State);
print(Fd, Trace, TraceInfo, State) ->
    trhandler(Fd,Trace,State).

trhandler(_Fd, {_,Pid,in,RunQ,Reds,Timestamp},{Times,ProcPropL}) ->
    % !!! using hacked ERLVM for RunQ/Reds !!!
    PSstate0 = get_pstate(Pid,ProcPropL),
    ...
    Times++[Timestamp], update_pstate(Pid,PSstate0#pstate{
        ...,
        ProcPropL}
    )
);

trhandler(_Fd, {_,Pid,out,RunQ,Reds2ds,Timestamp},{Times,ProcPropL}) ...
trhandler(_Fd, {_,Pid,Reds,call,Call,Timestamp},{Times,ProcPropL}) ...
trhandler(_Fd, {_,Pid,unregister,_Name,Timestamp},{Times,ProcPropL}) ...
trhandler(_Fd, {_,Pid,link,_LinkToPid,timestamp},{Times,ProcPropL}) ...
trhandler(_Fd, {_,Pid,spawn,SpawnPid,Call,Timestamp},{Times,ProcPropL}) ...
trhandler(_Fd, {_,Pid,send,Msg,ToPid,Timestamp},{Times,ProcPropL}) ...
trhandler(_Fd, {_,RcvPid,'receive',Msg,Timestamp},{Times,ProcPropL}) ...
trhandler(_Fd, {_,Pid,waiting,Msg,Timestamp},{Times,ProcPropL}) ...
trhandler(_Fd, {_,Pid,exit,Call,Timestamp},{Times,ProcPropL}) ...

Listing 1: print function and called trace handler declarations

It should be noted that the end_of_trace trace message tells us when the
tracing has ended. There is an important distinction here, since the beginning
and end of the trace may not be the same as the beginning and end of an overall
run.

All trace handlers process their trace event and save intermediate data to one
very long data structure, called ProcPropL. Such a structure would not suffice if
we were doing continuous monitoring, but it is sufficient for our purposes. The
structure is a list of tuples, each consisting of a PID and a pstate record. The
pstate record is defined as given in Listing 2. The different elements are used
throughout the trace handlers for the collection of the different metrics.

-record(pstate, {
    name, % human readable name for the process
    status, % status flag to keep a state of trace
    reds, % accumulated total number of reductions for the process
    top, % flag if this process is a high communicator
    mem, % memory usage by the process
    ctx, % array of context switches [{ctx,RunQ,Reds},...]
    calls, % array of traced calls [{call,Calls,Reds,Timestamp},...]
    rcvd, % array counting received msgs [{Key,Cnt},...]
    msgwait, % accumulated time spent waiting for msgs
    ctxwait, % accumulated time spent waiting for cpu time slice
    etswait, % (incomplete) time spent waiting for ets access
    spawn, % array of child processes [{Pid},...]
    locks % (incomplete) array of locks the run encountered
}).

Listing 2: pstate record for storing intermediate data

It is not really important which form of storage is used. We just need some
area to save intermediate data, because trace events are handled sequentially.
The structure we have described is also by no means a definite set. The data
structure you see in Listing 2 contains the fields specific to the use case described
in Section 4. Some of the fields are rudimentary metrics that will most likely
be common to many who require tracing. But, a future heuristic might need
different calculated intermediate metrics than what we have implemented.
As an example we can examine the handing of the in/out trace events. At each context switch, the in trace event is triggered before a process is switched in and given a cpu time slice. At that time, we temporarily store the reduction count and timestamp. When the process is switched out of the cpu, the out trace event is triggered. At this time we evaluate the reduction count and timestamp minus the values saved during the in trace event. We record the result in the pstate.ctx field, together with the number of the run queue the cpu time slice was run on.

### 3.5 Utilizing the trace data

If we only utilize the data recorded in the pstate.ctx field, we can actually already derive quite a lot of knowledge from this. Taking the very last timestamp and subtracting the very first timestamp from it, gives us a total run time of the trace which is more accurate than simply timing the trace. The reason for this being that the trace might not be equal to the entire run of the system and we are also able to exclude the time it takes to setup the environment and the trace system itself.

More so, we can sequentially list all processes that were run on a particular run queue in a single column listed downward. We can do this for each run queue, meaning we can create an entire mapping of columns, showing where, on which core, what order, all process were run and during which context switch.

Listing 3 is a small excerpt of a run depicted in columns using the metrics of the trace system. We should have four columns because this run was executed on a four core architecture and each column represents one run queue executed on a single core. But, we have omitted the fourth column for brevity and also accumulated subsequent process runs on a run queue into one line in the column.

<table>
<thead>
<tr>
<th>RunQueue,Count*&lt;PID&gt;@Reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 27* &lt;0.91.0&gt; @52667</td>
</tr>
<tr>
<td>1, 7* &lt;0.84.0&gt; @13805</td>
</tr>
<tr>
<td>1, 1* &lt;0.86.0&gt; @1983</td>
</tr>
<tr>
<td>1, 1* &lt;0.87.0&gt; @1</td>
</tr>
<tr>
<td>1, 1* &lt;0.87.0&gt; @1</td>
</tr>
<tr>
<td>1, 11* &lt;0.84.0&gt; @14057</td>
</tr>
<tr>
<td>1, 4* &lt;0.87.0&gt; @2050</td>
</tr>
<tr>
<td>1, 3* &lt;0.88.0&gt; @6025</td>
</tr>
<tr>
<td>1, 8* &lt;0.89.0&gt; @6039</td>
</tr>
<tr>
<td>36* &lt;0.xx.0&gt; @68456</td>
</tr>
</tbody>
</table>

{tt,1.577066,sec}

Listing 3: (abbreviated) column view of run queues of processes

Columns are separated by the symbol “|” and in each column we have a running list of the processes run on that run queue. The fields in the column, |RunQueue,Count*<PID>@Reductions|, are the run queue of the column, how many time slices that particular PID was run on the run queue, the PID of the process and lastly the total number of reductions for those processes. For example, if |1,27*<0.91.0> @52667| is displayed, this means process with PID <0.91.0> was run for 27 time slices in run queue number 1 for a total of 52667 reductions. The very last line of each these long columns, line 12 in our example, is a sum of the entire column with time slice count and total reductions.
depicted. This representation allows the programmer to visually control that each process has been mapped correctly to its respective run queue, that the overall throughput on each core is acceptable and that there are no erratic patterns are apparent. The very last line of the listing, line 15, shows the total execution time of the trace (including its unit of measure sec).

4 Use case: highest communicating process pairs

With the described framework we set out to optimize process scheduling according to highest communicating pairs of processes. This would be centered around a sender/receiver dependency similar to producer/consumer dependencies inspired by Bertels, Heirman and Stroobandt [BHS08]. We were interested in knowing whether placing high communicating pairs of processes (two processes which are sending and receiving many messages between one another) on different cores would give an increase in performance.

The exchange of messages from one process to another involves some synchronization pertaining to memory and also possibly some dependencies if the sender is waiting (blocking) on a response from the receiver. By scheduling each individual of a communicating pair on different cores, we allow both processes to execute simultaneously instead of possibly being forced to run sequentially on the same core, thereby hopefully alleviating some of the synchronization overhead.

Concretely, this meant monitoring a running Erlang system and recording all message communication between pairs of processes, to see which were passing the most messages. The total execution time is recorded and we select a number of processes pairs equal to the number cores in the system divided by 2. We then execute the system again, but this time scheduling each individual of a high communicating pair to a different core. Again we obtain a total execution time and we can compare the times of each run to see if there was a speedup. Of course, there are fluctuations in any architecture, so the whole test sequence must be run multiple times to obtain averages with corresponding standard deviations.

The final solution was implemented as a two pass test, which executed the SUT (with some pre- and post- processing) once without any modifications to system configuration and a second time using the results from the first in an attempt to optimize the second. Due to the limited nature of this thesis, we concentrated on performance monitoring based on overall execution time. We have pre- and post- processing to obtain system information at the start and end of execution. Although the system information we obtain this way is not entirely accurate relative to the trace messages, it does give us an indication of the beginning and end status of the processes that were run e.g., a basic list of the allocated PIDs as a minimum.
4.1 Set up tracing framework for the SUT

Two architectures available for analysis

Erlang version R13B02-1, Sept 23, 2009, was analyzed on...

- UPPMAX’s “Os” (Solaris), 10 nodes (IBM X3455 dual Opteron 2220SE, 2.8GHz nodes, each with 4 cores), 8GB RAM, Gigabit interconnect;
- and a Macbook Pro (MacOSX), 2.2Ghz Intel Core 2 Duo, 4GB RAM.

Highest level execution script

The highest level script that controls the execution of our test case is presented in Listing 4. In the listing we can see the declaration of the heuristic on line 2, declaration of the SUT initialization script on line 6 and the calling of the runloop on line 9, followed by some post-run calculations. Both the heuristic and SUT initialization scripts are assigned to the anonymous variables, Heuristic and Fun0 respectively, which we thereafter pass to the run system to be executed twice via runloop/2.

```
run7(It,Count,NrProducers,NrConsumers,CostProducer,CostConsumer) ->
    Heuristic = fun(ProcPropL) ->
        ...
        end,
    put(heuristic,Heuristic),
    Fun0 = fun({SchdP,SchdC}) ->
        ...
        end,
    {TTAcc0,TTAcc1} = runloop(Fun0,It),
    Time0 = calc_time(TTAcc0,stdv),
    Time1 = calc_time(TTAcc1,stdv),
    io:format("p'n'p'n'([Time0,Time1]),ok.
```

Listing 4: highest level script of run sequence

Pseudo code for the SUT

The SUT that we used for the test case was a simple producer/consumer model, but implemented in Erlang fashion using BIFs for spawning processes and language constructs for message passing. Figure 1 is a diagram showing the two cycles of producer, on the left, and consumer, on the right. The multiple arrows in the middle represent that many communications are happening because there are many producers and many consumers operating simultaneously. The corresponding pseudocode for the SUT can be found in Listing 5. It is trivial and bare-bone, because it simply serves as a placeholder for any code that we wish to run in test.
Figure 1: producer/consumer execution diagram

```
1 producer(0, _PidList, _Len, _Padding) -> ...;
2 producer(N, _PidList, _Len, _Padding) ->
   calc(_Len),
   Chance = rnd(100),
   case Chance < 50 of % 50% of uniform distrib
   true ->
      Consumer = rnd(length(_PidList)),
      Consumer ! {self(), rpc, _Len, _Padding},
      producer(N-1, _PidList, _Len, _Padding);
   false ->
      producer(N-1, _PidList, _Len, _Padding)
   end.
3 consumer_rcv(_Len) ->
   receive
      {_From, rpc, N, _Padding} ->
      calc(N),
      consumer_rcv(_Len) after 100 ->
      consumer_wait(_Len)
   end.
4 consumer_wait(_Len) ->
   receive
      {_From, rpc, N, _Padding} ->
      calc(N),
      consumer_rcv(_Len);
      {'EXIT', _From, _Reason} ->
      ok
   end.
```
Declaration of the Heuristic

In order for the framework to be able to gauge performance, the programmer must specify a heuristic function as a script in the form of an Erlang anonymous function assigned to a variable. This function variable is then passed to the framework, as can be seen in 6, which automatically executes it in the post-processing phase of the trace. All intermediate metrics stored in the structure, ProcPropL, is accessible by the heuristic function.

```
1 post_processing([{Times,ProcPropL0}] when is_list(Times) ->
    First = hd(Times),
    Last = lists:last(Times),
    TotalTime = timer:now_diff(Last,First),
    Heuristic = get(heuristic),
    ProcPropL1 = Heuristic(ProcPropL0)
    ...
```

Listing 6: execution of the heuristic in post-processing

For this use case, we must provide a heuristic that calculates a number of highest communicating pairs of processes. In order to calculate these pairs, we must have a count for each type of message that is received by a particular process, who sent it and estimate the amount of processing the message invoked. Therefore, a record is kept of all messages sent and received between two communicating processes. These are recorded in the per process list, ProcPropL, and post-processed by the heuristic. The end result for this use case is a scheduling decision separating the processes of a highest communicating pair to different hardware cores. It is difficult to determine exactly how much processing a message requires to be sent or invokes on receipt, due to the fact that this requires extensive static code analysis and might be highly dependent on input data. To make it possible to test the effects of varying the cost for producing and consuming messages, a parameter for each has been added which can be set at execution time.

```
1 Heuristic = fun (ProcPropL) ->
    TxnList = lists:foldl(
        fun ({Pid,PState},Acc) ->
            Rcvd = PState#pstate.rcvd,
            Calls = PState#pstate.calls,
            SendRcv =
                [ {FromPid,MsgID,Pid}, Cnt, reduction_cost(Calls,MsgID)}
                || {FromPid,MsgID}, Cnt <- Rcvd
                ],
                lists:append(Acc,SendRcv)
            end,[],ProcPropL),
    WorkList = [ {Txn,trunc(Cnt*Reds*Weight)}
            || {Txn,Cnt,(Reds,Weight)} <- TxnList],
    N = erlang:system_info(schedulers_online) div 2,
    TopN = lists:reverse(
        lists:nlstail(length(WorkList)-N),lists:keysort(2,WorkList)))
    end,
```

Listing 7: simple cost calculation implementing the heuristic

The heuristic function for this use case is quite elementary and is shown in Listing 7. It is simply the number of times the message was received by a process multiplied by the accumulated reduction count for all calls executed directly because of the message. Line 12 of Listing 7 shows the simple math for
the heuristic with an optionally weight which we can use to adjust the value. To reiterate, we simulate the cost for producing and consuming the messages using one function call for each cost with a parameter for the number of reductions to execute.

Guaranteeing determinism between executions

It would be ideal to be able to monitor the multitiered Erlang/OS system by reading out the identifier of the current Erlang process and simultaneously that of the hardware core the run queue is running on, but we opt to remove one tier of the hierarchy instead. We do this by statically assigning one run queue to each hardware core, dedicating it to running processes in that run queue (as long as no OS processes intervene). This feature is currently only supported by Erlang on Solaris architectures. The commands for this are shown in Listing 8.

```
1 Erlang R13B02 (erts-5.7.3) [64-bit] [smp:4:4] [rq:4] [async-threads:0]
Eshell V5.7.3 (abort with ^G)
1> erlang:system_info(scheduler_bindings).
{unbound,unbound,unbound,unbound}
2> erlang:system_info(cpu_topology).
[{{processor,{{core,(logical,0)}},
{core,(logical,1)}),
{core,(logical,2)}},
{core,(logical,3))}]]
2> erlang:system_flag(scheduler_bind_type, no_spread).
unbound
3> erlang:system_info(scheduler_bindings).
{0,1,2,3}
```

Listing 8: commands to bind run queues to specific cores

There is one other source of non-determinism between runs that we handle. Since the SUT we are using makes use of some random numbers, we also remove this independent variable in order to get a more stable test. To achieve this, we take advantage of the fact that repeatedly using the same seed with a random number generator generates the same random number sequence. By initializing processes requiring random numbers with the same initial seed before each of the two runs, this ensures the two independent runs of the SUT execute as similar as possible.

4.2 Comparative analysis between two passes

We execute the framework two times in total. In the first pass, we gather metrics, process the data via the heuristic to obtain a scheduling scheme relative to highest communicating pairs. In the second pass we utilize the scheduling scheme to map process pairs to cores and do the analysis again. A comparative analysis is done on the performance data from both runs. In listing 9 we see an excerpt of the output data from running the two passes and we continue by explaining it in detail.

```
1 {1,30,3,8,2107,2117}
Top [2,{{<0.93.0>},{<0.84.0>},16008},{{<0.92.0>},rpc,{<0.86.0>},12006}]
{<0.92.0>,producer_init,3}
5 {<0.93.0>,producer_init,1}
{<0.84.0>,consumer_init,2}
```
Listing 9: resulting highest communicators on four core architecture

Line 1 is only printed once and it is a regurgitation of the input parameters that were passed to the tracing system. The parameters are 
{Iterations,Count,NrProducers,NrConsumers,ProducerCost,ConsumerCost}
The number of iterations allows the programmer to control how many times the trace is executed. Since one trace consists of two passes, this means both are run the number of iterations specified. More iterations allows for a more accurate timing and a lower standard deviation. The rest of the parameters are specific to the SUT we chose. Count is the number of packets sent from all producers to any receiver. And, the next two pairs of parameters are the number of Producers/Consumers together with their respective calculation cost for producing/consuming a message.

We are searching for the highest communicating pairs of processes. The output line 2 is the direct result of the heuristic function and can be read as a list of two elements, [2, {...}, {...}], with one element being [{senderPID, rpc, receiverPID}, reductions], namely the sender’s PID, the receiver’s PID and the estimated cost of all communication between the two processes. In the listing, we have obtained two pairs of processes that according to the heuristic function have been communicating the most. Using the output from line 2, the heuristic generates a scheduling scheme that will be put to use in the second run. The scheduling scheme is the result of assigning each process in the order they appear in line 2 to a different core; the output of which can be seen in lines 4 to 7.

The number of lines that are output after line 4 corresponds to the number of processes selected as being highest communicating, i.e., two pairs means four processes as seen here. The fields {PID,NiceName,RunQueue}, represent the PID of the process, a friendly name for the process (for now this is actually the name of the initial function called to start the process) and the number of the run queue that the process should be statically mapped to. This last number is the ultimate result of the heuristic. As seen in lines 4 to 7, the framework suggests PID <0.92,0> to be mapped to run queue 3, PID <0.93,0> to be mapped to run queue 1, and so forth. This static mapping will be applied in the next run of the SUT by the framework.

The final output of the gauged performance is displayed on the last two lines, lines 10 and 11, one line for each execution of the framework. The parameters {avg,Iterations,Time,sec,Stdv} are the constant avg marker to signify that the average was taken of all the different iterations, the number of iterations, the computed execution time of the all the run iterations (including its unit of measure sec) and the standard deviation on the computed time. The standard deviation, {stdv,Time,sec}, is denoted as a constant stdv to signify the error statistic that was used, the time error itself and its unit of measure. From these lines, it is easy to compare the two calculated total run times to see if the scheduling scheme provided speedup.
Jimmy rigging process indexing and scheduling

A number of problems were mentioned in Section 3.2. We address two of them here. The first being the poor labeling of (PIDs) for processes. In order to identify the same process in two consecutive runs, we relied on the determinism of the Erlang VM being executed on isolated cores of a Solaris system. More precisely, our execution script spawns a number of processes in series and we trust that the creation of the processes happens in the same order on both consecutive executions. We index the processes on creation and in the second run assume processes with the same index are actually the same process, regardless of their PIDs.

The second issue we dealt with revolves around being able to schedule processes to specific cores, which is still hidden from the programmer. This becomes a problem, however, if one wants to test certain process scheduling schemes, such as we do. In order to access the schedulers to perform the mapping of processes to specific run queues, we had to rely on an undocumented feature supplied by Ericsson to manually assign processes to a specific run queue, which involved an enhanced version of spawn BIF.

4.3 Scaling and varying parameters

With the framework we can test a SUT repeatedly with variable parameters, in an attempt to find a correlation between our scheduling scheme for highest communicating processes and overall execution time. In Listings 10, 11, 12, we have the output of multiple runs of the framework. Each set of three consecutive lines of one run for the framework are identical in format to lines 1, 10 and 11 of Listing 9 above. In each of the listings we selected different parameters to scale.

```
1 {3,300,3,8,2107,2117}
{avg,3,12.2328903332333333,sec,{stdv,0.1879466599488564,sec}}
{avg,3,11.7633133333333333,sec,{stdv,1.0331893368103998,sec}}
{3,600,3,8,2107,2117}
{avg,3,23.2905466666666668,sec,{stdv,1.3153021672468352,sec}}
{avg,3,3412086666666667,sec,{stdv,3.765162123298677,sec}}
{3,900,3,8,2107,2117}
{avg,3,33.562758,sec,{stdv,1.7553641714339505,sec}}
{avg,3,41.339893333333333,sec,{stdv,0.9394367880463746,sec}}
{3,1200,3,8,2107,2117}
{avg,3,45.039804333333336,sec,{stdv,1.85081548746029,sec}}
{avg,3,56.168724,sec,{stdv,0.1958153548746029,sec}}
{3,1500,3,8,2107,2117}
{avg,3,57.034495,sec,{stdv,1.385503409531953,sec}}
{avg,3,68.884584333333333,sec,{stdv,1.559805439435674,sec}}
{3,1800,3,8,2107,2117}
{avg,3,66.7344526666666667,sec,{stdv,5.68786498713117,sec}}
{3,2100,3,8,2107,2117}
{avg,3,80.6029496666666667,sec,{stdv,6.730182746411886,sec}}
{avg,3,84.47031533333333,sec,{stdv,5.91882128410527,sec}}
Listing 10: scaling number of message sent

1 {3,600,3,8,2107,2117}
{avg,3,21.063572,sec,{stdv,0.4344078007962871,sec}}
{avg,3,27.4824403333333333,sec,{stdv,0.0701615936510516,sec}}
{3,600,9,24,2107,2117}
{avg,3,59.09763333333333,sec,{stdv,0.6940555081568686,sec}}
{avg,3,61.7017066666666667,sec,{stdv,0.6940555081568686,sec}}
```
Listing 11: scaling number of producers/consumers

Listing 12: scaling amount of work to produce/consume

4.4 Evaluation

Given the results, no obvious pattern is discernible. Some signs of improved efficiency can be found in small tests. But, when the amount of processes or work is scaled upward we seem to lose the advantage of statically scheduling the highest communicating processes to cores.

After having seen the run diagram output, the only speculation that we can make according to why performance gain is lost upon scaling upwards, would be due to the fact that static scheduling is too far rigorous. It would be interesting to perform continuous monitoring, thereby continually knowing which processes are highest communicating and scheduling those processes to the lowest loaded cores. Finally then, we might outperform the work stealing algorithm of Erlang.

5 Conclusion

After extensive research and implementation, it can be said that the major conclusion here is not the intended framework. The framework successfully implemented multicore monitoring for a small controlled SUT, but the framework is far from it’s intended design goal; the monitoring of full scale Erlang multicore applications. Even so, what we have obtained is a list of major issues that are impeding the framework from obtaining it’s goal. I will review them here. It might have been possible to solve some of these issues, but not in the time allotted for this thesis.
Obtaining metrics relative to trace messages timestamps

One of the design ideologies that was kept in mind during the creation of the framework, was to not have to alter the SUT source code. Yes, it is possible to do parse transform of the SUT source to obtain certain extra metrics, but this technique is not applicable for all metrics.

If the tracing system is to be of real use for monitoring, then the tracing system needs a way to get access to statistical information that resides in the Erlang VM. As an added criteria, it is not sufficient to simply have access to the information through a simple Erlang function call. The information needs to be related to particular trace messages that are being generated and passed to the tracing process/port. By example, if I have a trace message generated at each context switch (with corresponding timestamp), then I should be able to ask for the current state of the Erlang VM relative to that exact timestamp as well. It is impossible to do this with a simple function call. The information needs to be gathered by the Erlang VM and passed along with the trace message, labeled with a timestamp.

A good example of statistical information that might be needed is anything returned by the `process/system_info` function calls. But, this concept can be brought even one step farther.

Custom OS/architecture metrics in trace

Erlang is running on top of another OS, creating a multi-tiered system. If statistical information is wanted about the underlying OS, then how can this information be obtained without again altering the Erlang VM? If Erlang is to continue to run on many different architectures and each has its own extensions for monitoring, then we need to have a way to execute these custom extensions. Again, outside the trace this is not a problem, but getting this information relative to the generated trace messages is not possible with the current system. Could the trace system allow a stub of code to be configured with particular trace messages, which gets called each time the trace message is generated?

Programatic run queue assignment of processes

In order to be able to do effective monitoring, we need programatic control of process assignment, to be able to programatically start a process in a particular run queue and be able to move a process between run queues during program execution. The assignment should be configurable to be static or not, with respect to allowing “working stealing” to automatically move the process. An implementation as such would allow for dynamic programmer defined scheduling. Two examples of use, could be assigning two processes with a dependency between them to two different cores or perhaps isolating a process on a core dedicating processing power to it.

Solving the above issues will pave the way for the creation of a tracing system that is flexible enough to cater to the needs of multicore monitoring.

---

1All performance gauges throughout this entire thesis are based on the measure of Reductions. Because one Reduction is equal to a variable number of machine level instructions, the measure is not exacting enough. A more fine grained unit of measure relative to the Erlang level and the underlying OS level would be an improvement.
5.1 Possible Extensions

There are still a number of extensions to this thesis which we discovered, that were far beyond its scope.

Continuous monitoring, adaptive system

As we have seen in the data collection section, we use one large structure to collect the different trace metrics. The reason for this choice is because the handling of the trace data is post trace anyhow, so processing one large structure after the fact will not effect the result. But, what if we want to make an adaptive system? We would continuously want to monitor a running system and make alterations to the system at timely intervals. Such a system is not possible with the current implementation. Instead of having trace handlers that process the data after the fact, perhaps they should have a chance to process the data before it is being dumped to storage. This of course would incur processing time and we can question whether it is necessary to save the data to permanent storage at all.

What is the cost of monitoring a running system?

As mentioned, continuous monitoring would incur processing time. What is hidden from this thesis as well, is the amount of processing time the trace incurred. We have utilized the built in Erlang trace system, which sends trace messages to a dedicated process. But, the operation of sending the trace messages and saving of data to storage also requires processing time, because the trace process must run on a particular core as well. If we envision a many many core machine, how many of these cores are we willing to sacrifice for monitoring? How many are required to sustain monitoring? Or shall we simply continue with the current implementation style and wildly schedule tracing across the running system?

Monitor Erlang and OS tier simultaneously

This brings us to another issue. It is impossible at this point to dedicate some of the cores to tracing a running Erlang system. Even if we could dedicate four run queues to tracing, we have no way to guarantee that those run queues have cores dedicated to them. In this thesis, we did a binding of run queues to cores, but the cores were not dedicated in any way. The operating system was still free to run other system processes on any core effectively swapping out the entire Erlang run queue.

We have already mentioned that attempting to monitor Erlang and simultaneously monitor the underlying OS is a highly complex task. However, whether implemented on the OS level and communicated to the Erlang level externally or implemented directly in Erlang, it would be beneficial to know how the OS processes representing the Erlang run queues behave. Especially since other OS processes might be hindering or effecting the execution of the Erlang VM.
Graphical user interface for visualization

One of the most straight forward and gratifying extensions would be to create a proper graphical visually pleasing output for the trace of a program executed on a multicore architecture. Since multicore architectures are only now taking serious hold, tools for developing multicore programs are still severely lacking. A graphical user interface could allow for a visual analysis of program flow which is far more efficient in displaying simultaneous execution than simple line output. But, creating such an analysis tool is a daunting task. In a fraction of time, we have a plethora of information. And, it is not well established exactly what information is of use to a programmer developing multicore applications. Resource dependencies or inter-dependencies is a possible candidate. What are the others?
References


