Process Cooperativity as a Feedback Metric in Concurrent Message-Passing Languages

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Matthew Fluet, Supervisor

Dr. James Heliotis, Reader

Dr. Rajendra K. Raj, Observer

Process Cooperativity as a Feedback Metric in Concurrent Message-Passing Languages

by

Alexander Dean, B.S.

THESIS

Presented to the Faculty of the Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

Rochester Institute of Technology

August 2014

Abstract

Process Cooperativity as a Feedback Metric in Concurrent Message-Passing Languages

Alexander Dean, M.S.
Rochester Institute of Technology, 2014

Supervisor: Dr. Matthew Fluet

Runtime systems for concurrent languages have begun to utilize feedback mechanisms to influence their scheduling behavior as the application proceeds. These feedback mechanisms rely on metrics by which to grade any alterations made to the schedule of the multi-threaded application. As the application's phase shifts, the feedback mechanism is tasked with modifying the scheduler to reduce its overhead and increase the application's efficiency.

Cooperativity is a novel possible metric by which to grade a system. In biochemistry the term cooperativity is defined as the increase or decrease in the rate of interaction between a reactant and a protein as the reactant concentration increases. This definition translates well as an information theoretic definition as: the increase or decrease in the rate of interaction between a process and a communication method as the number of processes increase.

This work proposes several unique takes on feedback mechanisms and scheduling algorithms which take advantage of cooperative behavior. It further compares these algorithms to other common mechanisms via a custom extensible runtime system developed to support swappable scheduling mechanisms. A minimalistic language with interesting characteristics, which lend themselves to easier statistical metric accumulation and simulated application implementation, is also introduced.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Runtime systems can be broken up into multiple distinct parts: the garbage collector, dynamic type-checker, resource allocator, and much more. One sub-system of a language's run-time is the task-scheduler. The scheduler is responsible for order of task evaluation and the distribution of these tasks across the available processing units.

Tasks are typically spawned when there is a chance for parallelism, either explicitly through `spawn`, or `fork` commands or implicitly through calls to parallel built-in functions like `pmap`. In either case it is assumed that the job of a task is to perform some action concurrent to the parent task because it would be quicker if given the chance to be parallel.

It is up to the scheduler of these tasks to try and optimize for where there is opportunity for parallelism. However, it's not as simple as evenly distributing the tasks over the set of processing units. Sometimes, these tasks need particular resources which other tasks are currently using, or maybe some tasks are waiting for user input and don't have anything to do. Still worse, some tasks may be trying to work together to complete an objective, like in the `pmap` example above.

Tasks however, in functional language verbiage, are typically called *processes* due to the inherent isolation this term brings and the language paradigm calls for. So how do these processes share information or return a value back to their parent? Message passing is a common abstraction to shared memory. Message passing is akin to emailing a colleague a question. You operate independently, and your colleague can check her mailbox when they want to and respond when they want to. Meanwhile you are free to operate on an assumption until proven wrong, wait until she gets back to you, or even ask someone else.

While message passing is a good method for inter-process communication, it is also a nice mechanism for catching when two processes are working together. Let's, for example, consider a purely functional `pmap`, where all workers are copied subsections of the list. Each worker thread will have no need to cooperate and thus no messages will need to be passed amongst them. However, suppose the function being mapped uses several processes accessing a piece of shared memory to update it's particular cell in the list. These processes would do little good if treated like the course-grained parallelism the `pmap` workers create.

Bad example, should be more real world, and doesn't flow well into the next paragraph.

There are a large number of mechanics that scheduling systems can use in an attempt to improve work-load across all processing units. Some of these mechanics use what's called a feedback system. Namely, they observe the running behaviour of the application as a whole, (i.e. collect *metrics*), and modify themselves to operate better.

Process cooperativity is an interesting metric by which to grade a system. In biochemistry the term cooperativity can be defined as an increase or decrease in the rate of interaction between a reactant and a protein as the reactant concentration increases. We can translate this into an information theoretic definition:

Definition 1. *The* **degree of cooperativity** *of a system is the increase or decrease in the rate of interaction between processes and an inter-process communication method as the concentration of processes fluctuate.*

Thus, when a process attempts to pass a message to another we know it's trying to cooperate on some level. When this frequency of interaction is high, it may indicate a tight coupling of processes or fine-grained parallelism. If it is low, this could indicate course-grained parallelism. In either event, a scheduler able to recognize these clusters of cooperative and non-cooperative processes should have an edge over those that don't.

Chapter 2 will look first at the background of classical scheduling systems as well as the recent feedback-enabled approaches. Then we will also examine the types of message passing implementations and how these effect scheduling decisions, now that we are looking at process cooperativity. Chapter 3 introduces our work on a language and compiler, built to easily simulate system cooperativity and visualize the effects of scheduling mechanisms on these systems. We also discuss a few example mechanics which take advantage of cooperativity. Some example applications which demonstrate different degrees of cooperativity and phase changes are also explained. In Chapter 4 we run our cooperativity-enabled schedulers along with a few common non-feedback-enabled schedulers on the example applications and discuss the results. Finally, in Chapter 5 we give some concluding remarks and avenues of future research we believe would be fruitful.

Chapter 2

Background

2.1 A Note on Control Theory

Since the formalization of feedback driven systems and the advent of Cybernetics, multiple fields have attempted to mold these principles to their own models; and run-time schedulers are no exception. This is due, in part, from process scheduling in parallel systems being fundamentally an NP-Complete problem [1].

Note that the simple case of runtime scheduling is called the Multiprocessor Scheduling Problem which states:

Definition 2. MULTIPROCESSOR SCHEDULING PROBLEM
Given a set of jobs $\mathscr{J} = (J_1, J_2, \ldots, J_n)$*, a directed acyclic graph (lattice)* $L = (\mathscr{J}, C)$ *(indicating job dependence, and thus precedence constraints), an integer* P *(the number of processors) and an integer* D *(the deadline), is there a function* S *(the schedule) mapping* $\mathscr{J} \to \{1, 2, \ldots D\}$ *such that:*

1. *For all* $t \leq D, |\{J_i : S(J_i) = t\}| \leq P$*. (# of jobs scheduled per time slice is no more than # of processors)*
2. *If* $(J_i, J_j) \in C$*, then* $S(J_i) < S(J_j)$*. (# jobs cannot be scheduled before their dependencies)*

In runtime scheduling, the deadline, D, is incremented for each timeslice we pass. As such, it is possible for $|\mathscr{J}|$ and thus C to fluctuate causing a need to re-find S. The continuous nature of this problem complicates the scheduling problem substantially. Instead, focus has been more fruitful when pursuing the optimization of various measurements using some particular objective function [2] to tune for particular edge cases. As such, scheduling based on such feedback metrics is not a new practice [3].

There is a big distinction though, which can be made between the effects of control theory in classical cybernetic applications versus that of run-time systems. This is primarily in the adaptation of the controller in the generic feedback loop (figure 2.1).

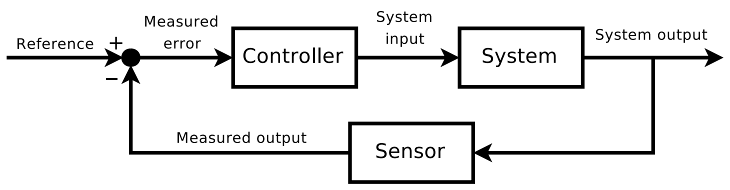


Figure 2.1: A classical feedback loop representation.

In typical physical feedback loops there are two scenarios which need to be avoided: resonance and rapid compensation. It can be seen that most controller models will attempt to damp the adjustments to reduce oscillation which could cause resonance or sharp spikes in behavior based on its output. This is due to the limitations of the physical space in which they are having to deal with.

However, in run-time scheduling systems we would very much like to do the opposite. We would prefer tight oscillations or consistent behavior of our runtime so as to achieve minimal overhead from our modifications. We can also compensate, to reach our reference signal, as quickly as we need to as there are no physical restrictions for our modifications.

As such these feedback systems are closely coupled with the design of the scheduling algorithm, rather than being an interchangable sensor, and controller modules. As such we make an effort to trace the feedback optimizations during our evaluation and explaination of the scheduler designs.

Another distinction must be made as far as the level of foresight the scheduling systems have, at least, within this paper. There is a spectrum of clairvoyance in classical job-scheduling, in that on one end, job-schedulers have full foresight over the jobs which will enter the queue and their order (*i.e.* the full $\mathscr{J}$ set will always be known). These schedulers have the opportunity to optimize for future events (by constructing a valid lattice L based on the current time $t \leq D$), which is a luxury the scheduling systems that this paper discusses do not have.

However, as it is a spectrum, there is a single point of knowledge this subrange of schedulers can assume. Namely, that the first job will always be the last, and all other jobs will spawn from it. Thus there will always be a single process in the queue at the beginning. This is true as the runtime will always require an initial primary process (*e.g.* the 'main' function), and once that function is completed, the system is terminated (despite the cases

Note: So main is required to terminate?

of unjoined children). Apart from this, all other insights will need to be gleaned from the evaluation of this initial process.

We mention this due to the implications on our scheduling problem (dynamicly evolving lattice which is altered based on evolving cooperation networks and the forking/joinging of child processes), but can we tie this back to the above definition and possibly cooperation?

2.2 Classical Runtime Scheduling

I would like to discuss current methods of distributing processes accross processors, as I am assuming later that readers know of terms like:
- work-stealing & sharing
- round-robin
- global/local process queues
- spawning and joining methods

Then somehow lead into process dependencies and thus process synchronization. This will tie it over to the message-passing section.

2.3 Feedback-Enabled Scheduling

This is the "related work" section, I would like to give the two big examples i've been considering: CML, and Occam-π.

2.3.1 Cooperativity as a Metric

This should give the primary definitions of a $cooperativeprocess$ and the differences between focusing on cooperativity rather than interactivity, etc. It would also be a good idea to talk about its relationship with application phasses here.

2.4 Message-Passing

In concurrent systems, there are a number of methods for inter-process communication. Arguably though, one of the more popular abstractions is the idea of message passing. This is expecially true in functional languages as the language assumes shared-nothing by default. Just as compilers can optimize using the language constraints, so can the run-time using the implementation.

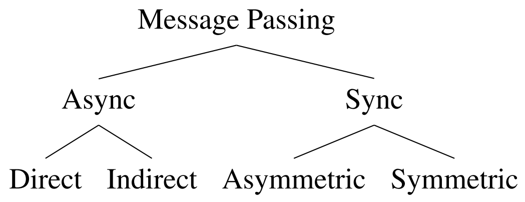


Figure 2.2: A High-Level Message-Passing Taxonomy

Message passing in general can be broken down into two types based on the language's implementation; asynchronous or synchronous. In asynchronous message passing a process can either be provided a rendezvous point or an identifier for another process. To send a message in either case requires pushing/copying the message into a shared or global memory space for another process to access (possibly) at a later time. This push/copy can be done in a lock free manner with some lower level atomic data structures such as a double-ended queue. But in either a locked or lock-free manner, the process performing the send still blocks on the operation to push the message.

In terms of scheduling, a language with asynchronous message passing can ignore the effects of these blocking operations, but will need to look closer at process placement to take advantage of possible gains in cache affinity [4]. For example, the effects of the cache on direct message passing (*e.g.* a process mailbox) can be substantial as two processes on different cores will need to copy stack frames to memory and back, rather than just a pointer to the message in the originator's stack frame. In indirect message passing the task is even worse as multiple processes may need access to the same data.

In synchronous message passing, a process must meet another at a provided rendezvous point but can either be symmetrical or asymmetrical. Note that the rendezvous point is not a requirement in the sense that direct synchronous messaging isn't possible. Instead we think of a rendezvous point in synchronous communication to be time bound rather than location bound (*i.e.* two processes are blocked until communication occurs, the implementation of this passing is irrelevant to this classification).

Asymmetrical message passing is synonymous with Milner's Calculus of Communicating Systems [5] or standard Π-Calculus [6], in that you have a sender, and then a receiver which will both block on their respective functions around an anonymous channel until the pass has been completed. This differs from a symmetrical message passing in that the only operation on the channel is a blocking function which swaps values with the process at the other end.

Note, it is possible to simulate symmetrical message passing on asymmetrical message channels, but in terms of scheduling of synchronizing processes, order is now a factor that needs to be considered. On top of this, directionality can also be a factor which complicates the channel implementation. Namely, the internal queuing of senders or receivers may not percolate hints up to the scheduler regarding their queue position.

For the alternative, symmetrical message passing or swap channels, the order is directly handled by the scheduling of the system (*i.e.* the order at which the channels evaluate the $swap$ command can be directly governed). And it is for this purpose along with simplifying our base language we have chosen to base our semantics on symmetric synchronous message-passing.

I feel like this section would benefit from some more figures describing the language primitives involved and their semantics. The key point to make is how much is available to the scheduler in each instance:
- Async: Potentially nothing, but if so:
    - Direct: The particular process which it communicated with,
    - Indirect: The channel/rendezvous it communicated with,
- Sync: If it blocks, a communication happened, potentially can know order in queue
    -Asymm: Directionality.

But noting that for cooperativity purposes, directionality is not meaningful.

Should i talk more about possible implementations and their considerations? I could go into the process absorption that SML and Occam-π does?

Chapter 3

Methodology

3.1 Overview

To examine the effects of cooperativity conscious schedulers we needed to have a method for comparing several scheduler implementations without needing to modify the underlying implementation of processes, channels, or application source code. It would be also beneficial if our solution were able to visualize these differences similar to Haskell's ThreadScope [7].

Our solution, $ErLam$, is a compiler for an experimental version of Lambda Calculus with Swap Channels and a runtime system which allows for swappable scheduler mechanisms and an optional logging system which can be fed into a custom report generator. We break up our solution description into three parts; Section 3.2 will duscuss our language syntax and semantics. It will also demonstrate our Runtime Scheduler API by breaking down the CML Interactivity scheduler. Section 3.3 will go more into depth about our testing environment which involves our logging system, the report generator, and the set of example applications we used to represent different cooperativity levels. Finally, Section 3.4 will go over our example schedulers we wrote which demonstrate cooperative-conscious behavior. These will be the schedulers we provide our results against.

3.2 ErLam

The ErLam toolkit is itself broken down into three parts, the language and its semantics, the Runtime System, and the Scheduler API. We will first lay out the language and it's basic semantics, as the finer-details are reliant on the exact selected scheduling solution as well as the chosen swap-channel implementation. We will then examine the possible channel implementations and how they effect the given semantics. Finally we will discuss the Scheduler API using an example scheduler implementation.

```
<Expression> ::= <Variable>
               |  <Integer>
               |  `newchan'
               |  `(' <Expression> `)'
               |  <Expression> <Expression>
               |  `if' <Expression> <Expression> <Expression>
               |  `swap' <Channel> <Expression>
               |  `spawn' <Expression>
               |  `fun' <Variable> `.' <Expression>
```

Figure 3.1: The ErLam language grammar, without syntax sugar or types.

3.2.1 The ErLam Language

The ErLam Language is based on Lambda Calculus, with first-class single variable functions, but deviates somewhat in that it provides other first-class entities. It deviates from Church representation to provide Integers, this is purely for ease of use. It also provides a symmetric synchronous Channel type for interprocess communication. As a note, this language can also be classified as a Simply-Typed Lambda Calculus.

ErLam also makes a number of ease-of-use decisions like providing a default branch operator and has some useful syntactic sugar such as SML style let expressions and multi-variable function definitions. There is also a set of built in functions for numeric operations, type checking, and standard functional behaviors (*e.g.* combinators, *etc.*) which are ignored in this document.

Figure 3.1 expresses ErLam in its simplified BN-Form. The semantics for the language is fairly straight forward, but it's operational semantics are layed out in appendix 1. All expressions reduce to one of the terminal types: Integer, Channel, or Function. To spawn for instance, if any terminal is passed other than a function, it returns a 0 (*e.g.* false). When the function is passed, it is applied with nil to initialize the internal expression.

ErLam extends this grammar only a little to add SML style let expressions and multiple variable functions which are curried from left to right (see figure 3.2 for syntactic transformation). We will be using this syntactic sugar throughout this document to make our source easier to review.

Also, note the possible steps **swap** can take: either returning the null set or another expression and a set of functions. In the former's case, the channel has blocked and the only course of action is to get another expression to work on from the scheduler. In the later's

$$\textbf{let}\; x \;=\; e_1 \;\textbf{in}\; e_2 \Rightarrow (\textbf{fun}\; x.e_2\; e_1)$$

$$\textbf{fun}\; x,y,z.e \Rightarrow \textbf{fun}\; x.(\textbf{fun}\; y.(\textbf{fun}\; z.e))$$

Figure 3.2: Syntactic sugar parse transformations.

case, we have an expression to work on, but we also may have unblocked other processes in the process so we need to reschedule them.

Note however, that return set may be null and the expression returned may be another attempt at swapping (*i.e.* $e = (\textbf{swap} cv)$). This would let the scheduler choose whether to retry immediately or reschedule it for a later time and work on something in the mean time.

3.2.2 Channel Implementations

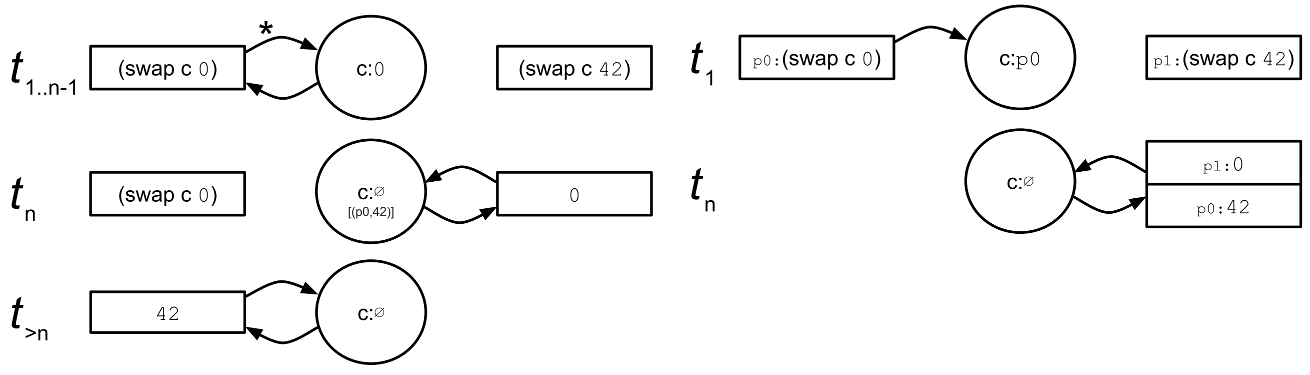
ErLam provides a selection of channel implementations so as to allow for interchangable comparisons to be made with synchronization methods and the schedulers themselves. ErLam comes with two channel implementations the *Blocking* Swap, and the *Absorbing* Swap. We will now look at an example application and it's execution using both methods for comparison.

Figure 3.3 gives an example ErLam application. It first creates a new channel for processes to communicate on. It then creates a null-function to spawn, who's sole purpose is to swap on the channel the number 42 and then die. Finally, it swaps on the channel the number 0 and returns the result of the whole evaluation, which in this case will be the value passed from the other end of the swap, 42.

```
let c = newchan in
let f = (fun _.(swap c 42)) in
let _ = (spawn f) in
in (swap c 0)
```

Figure 3.3: A simple ErLam application which swaps on a channel before returning.

As ErLam is innately concurrent, we do not know which process will ask to swap first. It may even be possible that 0 asks to swap several times before 42 even tries. In fact, the *Blocking* channel allows multiple swap attempts, We can see an illustration of this in figure 3.4(a); note the illustration makes no mention of the scheduler or it's functionality. It may be the case that the two processes are on different processing units and are in different



(a) Process Blocking Swap (b) Process Absorption Swap

Figure 3.4: Channel operation over time. Note abitrary time-slice t_1 is when the first swap operation is evaluated.

process queues. Or it may be the case that they both exist in the same queue and upon a block, the scheduler chooses the next one, which will immediately unblock the channel.

The *Blocking* channel effectively simulates a common spin-lock over a shared piece of memory. These channels represent a worst-case, albeit common, application implemenatation for concurrent software. Even so, they do allow for some hints to the scheduler, which can be taken advantage of. Alternatives on the spin-lock could be added to the ErLam toolkit though, such as a push-notifying semaphore, however we provide a simpler and functionally more common alternative: the process absorption channel.

In the *Absorption* channel (figure 3.4(b)), the first process to get to the channel will get absorbed by it. The scheduler in question will be out a process, and but the scheduler which unblocks the channel by arriving second will get back two processes (the one performing the swap, and the absorbed one). In terms of scheduling efficiency this type of message passing channel has provideded enormous improvements for runtimes which do not wish to introduce channel inspection into their scheduler.

3.2.3 The Scheduler API

ErLam was written in Erlang, and as such, it takes advantage of Erlang's callback behaviour specifications. An *erlam_scheduler* behaviour has been defined which requires a minimalistic 5 callback functions (figure 3.5).

```
-callback layout( erlang:cpu_topology(), scheduler_opts() ) ->
                scheduler_layout().

-callback init( scheduler_opts() ) ->
                {ok, scheduler_state()} |
                {error, log_msg()}      |
                {warn, log_msg(), scheduler_state()}.

-callback cleanup( scheduler_state() ) ->
                ok | {error, log_msg()} | {warn, log_msg()}.

-callback tick( scheduler_status(), scheduler_state() ) ->
                {ok, scheduler_status(), scheduler_state()} |
                {return, term()} | {stop, scheduler_state()}.

-callback spawn_process( erlam_process(), scheduler_state() ) ->
                {ok, scheduler_state()} | {error, log_msg()}.
```

Figure 3.5: The ErLam Scheduler API

Upon instantiation of the runtime the runtime system will call the *layout/2* function with the NUMA layout of the system the application is running on, along with any parameters the user specified at runtime. The result of this function is to be the scheduler layout.

For example, let's assume we are running our application on a Intel Core i7 which has 4 logical cores which support hyperthreading. The *layout/2* function will be given the following structure:

```
[{processor,[{core,[{thread,{logical,0}},{thread,{logical,1}}]},
             {core,[{thread,{logical,2}},{thread,{logical,3}}]},
             {core,[{thread,{logical,4}},{thread,{logical,5}}]},
             {core,[{thread,{logical,6}},{thread,{logical,7}}]}]}].
```

This indicates to the scheduler implementation that it at max, can spawn 8 instances of itself which would be bound to each logical processing unit (LPU). We could of course have a scheduler which acts differently based on how the architecture. However, the schedulers we have limited ourselves to are either single or fully multi-core (*i.e.* uses all available LPUs).

To spin up an instance of the scheduler on the particular core, the *init/1* function is called which should return the scheduler's state. As Erlang is a functional language, we use this state object as a means to maintain some global state for each scheduler process

```
spawn_process( Process, State ) ->
    enqueueAndSwitchCurThread( Process, State ).

enqueueAndSwitchCurThread( Process, #state{curThread=T}=State ) ->
    case T of
        nil ->
            setCurThread( Process, State );
        _   ->
            % New process takes over
            {ok, NewState} = enqueue1( T, State ),
            setCurThread( Process, NewState )
    end.
```

Figure 3.6: CML Process Spawning.

by threading it through all subsequent callback calls. Upon shutdown, the oposite function *cleanup/1* is called.

The last two functions are the most interesting as they pertain to the core of what each new scheduler provides, namely how to evaluate the world in a given timeslice (*tick/2*) and how a new process should be handled (*spawn_process/2*); but an explaination of these callbacks is best done through example.

3.2.4 Example: The CML Scheduler

Concurrent ML is an extention to SML which adds the *spawn* function, and channel operations. CML's scheduler utilizes a dual-queue structure rather than a simple unary-process-queue. The scheduler attempts to differentiate between *communication* and *computation*-bound processes so as to reduce the effects of highly computationally intensive processes from choking the system. The scheduling system thus improves on application interactivity by demoting *computation*-bound processes to the secondary queue (which isn't accessed until another process is demoted).

Spawning a process in the CML scheduler (figure 3.6) does not go onto the primary queue, instead we enqueue the current process and start evaluating the new process. This is a fairly simplistic example, but it shows how one would go about updating the state between ticks.

In the original CML scheduler, it defined a quantum which it would let the current process run for, and then it would preempt it if it attempted to run for longer. The ErLam

```
tick( _Status, #state{ curReduct=0 }=State ) ->
    {ok, NState} =  pick_next( State ),
    reduce( NState );
tick( _Status, State ) -> reduce( State ).

pick_next( State ) ->
    {ok, NewState} = preempt( State ),       % Place cur thread onto queue
    {ok, Top, Next} = dequeue1( NewState ), % Pop next off
    setCurThread( Top, Next ).              % Set as cur and return state
```

Figure 3.7: CML Process evaluation.

runtime avoids the use of time based quantum as logging and other factors directly effect the usefulness of this. Instead it uses a 'tick', which is suppose to emulate one step forward in the execution of the application. Thus to simulate a quantum we instead keep track of the number of reductions performed on the current process and decrement the counter until we reach 0.

The *tick/2* function (figure 3.7) performs one of two things based on what the state of the system is. If the current reduction count is 0, then we can pick a new process from the queue, otherwise we can perform a reduction.

Note for our scheduler simulation we ignore the first parameter to the *tick/2* function for either case. The first parameter was the status of the scheduler returned from the previous tick (*e.g.* running, waiting, *etc.*). This would be useful if the CML scheduler utilized work-stealing to get work to do from other LPUs when in *waiting* mode.

3.2.5 Provided Schedulers

Along with the Single-Threaded Dual Queue CML scheduler *(STDQ)*, ErLam comes with several basic scheduling mechanics. We utilize these as bases cases on which to compare the behaviour of all subsequent feedback-enabled schedulers.

- **The Single-Threaded Round-Robin Scheduler** *(STRR)*
  This scheduler uses a single FIFO queue which all processes are spawned to. There is no rearrangement of order, and the single-thread scheduler will just round-robin the queue performing a set number of reductions per process before enqueuing and poping the next one.

- **The Multi-Threaded Round-Robin Global-Queue Scheduler** *(MTRRGQ)*
  A multi-core version of the previous scheduler. This uses a single global process queue which all schedulers share and attempt to work from.
- **The Multi-Threaded Round-Robin Work-Stealing Scheduler** *(MTRRWS)*
  An improvement on the previous scheduler. Instead of a global process queue, each scheduler maintains their own. A waiting scheduler will randomly sleep-and-steal until it finds a process to work on from another scheduler. The provided implementation gives two example stealing mechanisms:
  - **Shared-Queue** *(MTRRWS-SQ)*
    Stealing a process involved performing an atomic dequeue from the bottom (rather than the top) of another scheduler's process queue. This will only block the other scheduler from performing a dequeue for a very short window of time, but involves accessing "remote" memory.
  - **Interrupting-Steal** *(MTRRWS-IS)*
    Simulates sending an evil theif-process over to another scheduler. When the victim scheduler preempts or yields their current process and selects the next one from the queue, they will instead get a theif process which will syphon a process away to the theif's home scheduler.

ErLam also comes with three cooperativity-conscious schedulers: the Longevity-Based Batching Scheduler (section 3.4.2), the Channel Pinning Scheduler (section 3.4.3), and the Bipartite Graph Aided Shuffling Scheduler (section 3.4.4). The first two build on the same shared queue module as provided by $MTRRWS$, while the third utilizes it's own implementation.

3.3 Simulation & Visualization

The second primary goal of the ErLam toolkit was the ability to visualize how the scheduler proceeded to evaluate a simulated application. Thus we needed a way to log all events over time, including unique per-scheduler events, such as the size of both the primary and secondary queues in the CML scheduler.

As a secondary goal, we need a sample set of application simulations to run against our set of schedulers. These simmulations would need to be concise and demonstrate various levels of cooperativity and phase changes. We would like to also have the ability to mix and match test cases together to better create realistic work-sets for the schedulers to react to.

3.3.1 Runtime Log Reports

Logging in Erlang is a fairly simple matter. We utilize a simplistic data logging module based on syslog. The output of running an application could look like this:

```
timestamp,lpu,event,value
...
983847.935268,3,sched_state,running
983847.935333,0,queue_length,59
983847.935677,24,channel_blocked,6102
983847.935683,6,yield,""
983847.936003,0,queue_length,59
983847.936430,3,tick,""
983847.936439,3,reduction,""
...
```

The timestamp given is a concatenation of the second and microsecond that the event happened in. The lpu is the scheduler which caused the event, unless it's a channel based event, such as a *channel_blocked* event, in which case it's the channel ID.

Our logging API is fairly simplistic as we only need to capture two types of metrics from our events: quantity and frequency. With frequency, we want to know the amount of events which happened in a time range, but with quantity we would like things like length of the scheduler's process queue over time or the amount of time spent in the running or waiting state.

Note time is not consistent per LPU, it may be the case that another OS application is getting time instead of one of the ErLam schedulers. This could result in one or more of the LPUs getting far less *"tick"* events. Worse yet, there may be a large gap of time missing from one scheduler to the next. For our purposes though we would like to compare the state of the scheduler while it is executing and would be fine with averaging over the largest gap. These from experimentation have not been found to be very frequent or large on an otherwise unoccupied processor.

Anecdotal! I need to prove this

To explain this averaging technique we'll now discuss the report generation method. The ErLam toolkit comes with a secondary R script which can be given a generated log file for processing. This script dynamically loads chart creation scripts based on the types of events it sees in the log file. The toolkit comes with five charting scripts which should work for all schedulers: Channel Usage (Communication Density) over time, Channel State (blocked vs. unblocked) over time, Process Queue Length per LPU over time, Reductions (Computation Density) over time, and Scheduler State (running vs waiting) over time.

Communication Density for example (see figure 3.8, creates a heatmap based on the frequency of *yield* events which occur whenever a process attempts a $swap$. Each cell of the heatmap is a color intensity based on the number of *yield* events seen in a given timeslice for a given LPU. This timeslice is where the averages come into play. R heatmaps have a max number of colors of 9, so any range we select must be modulo 9. However, the constant multiplicand is based on the mean amount of time N ticks take place accross each LPU. We can obviously tune the accuracy of these averages on a per-LPU basis by modifying N. Anecdotally, this turned out to be adventagous on several occasions when debugging scheduler implementations. As decreasing the number of ticks to average for, pushed the number of samples (and thus accuracy) higher.

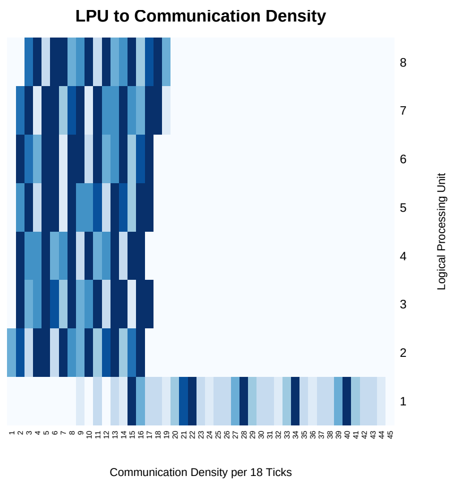


Figure 3.8: Example of Communication Density graph for the Work-Stealing scheduler on a Core i7 running the $PRing$ application.

3.3.2 Cooperativity Testing

As part of the thought experiment, we needed to implement a decent set of test cases which would give us a good coverage of the range of cooperative behaviour in our applications.

On one hand we have an axis depicting the amount of parallelism possible in our application. A system which is completely parallel, would be one where all processes spawned have no dependence on any of the others. For our toolkit, we called this behaviour $ChugMachine_N$ (figure 3.9(d)) ; where N depicts the number of parallel processes. On the other side of the axis, we would have a system which had absolutely no parallelism possible. We called this behaviour $PRing_N$ (figure 3.9(b)), as it would spawn N processes in a ring formation and pass a token in one direction. Each process has a channel to its left and right and would synchronize to the right until it recieves a token to continue.

$PRing_N$ also gives an example of full-system cooperation, except we would instead like some degree of parallelism possible. To expirement with that, we would have to throttle the degree of cooperativity. This behaviour is called $ClusterComm_{(N,M)}$ (figure 3.9(c)) as it spawns N processes and M channels which can be synchronized with by any process. Note for this system to work with swap channels we limit M to be at most $\lfloor N/2 \rfloor$ for all tests.

$ClusterComm_{(N,M)}$ is also an example of full-system cooperation, we also want to have a possible case for partial-system cooperation. We begin this range of experiments with a behaviour which acts like a bunch of $ClusterComm_{(N,1)}$ running in parallel. We call this special case behaviour $PTree_W$ (figure 3.9(a)); where W is the number of work groups to run in parallel. This is the cleanest case of partial-system cooperation. We would expect to see obvious clustering of processes by work-group affiliation if the scheduler was cooperativity-conscious.

However, to expand on the concept of partial-system cooperativity, we would also like to experiment with lop-sided behaviours where a work-group exists along with other processes which may not be affiliated with one another. An application like this would be the combination of $ClusterComm_{(N,M)}$, $ChugMachine_N$, and $PRing_N$ running in parallel.

We are missing two important behaviour simulations: application phase changes, and hanging processes (typical of I/O bound processes). In the case of the latter, a simple built-in command *hang* was provided which would simulate hanging for a random amount of time before allowing the process to proceed with evaluation. If a scheduler attempted to reduce the process before the *hang* time was completed, it would be immediately preempted. The behaviour implemented we call $UserInput_{(T,C)}$ (figure 3.9(e)); where T is the max time in seconds the process would hang before continuing, and C is the number of times it would simulate "waiting for user input".

For the former missing behaviour, phase changes, we decided to make a variation of $PTree_W$ called $JumpShip_{(W,P)}$ (figure 3.9(f)) which would act like $PTree_W$ but would "change phase" P times before completion. The act of "changing phase" would be the successive relocation of all the processes from one work-group to another, effectively having all processes from work-group X "jump-ship" to $X + 1 \textbf{ mod } N$.

We would have liked to possibly experiment with variations on the "jump-ship" behaviour so as to inject phase changes into $PRing_N$ (by perhaps reversing direction) or $ClusterComm_{(N,M)}$ (by switching to $ChugMachine_N$ for a brief period before returning to $ClusterComm_{(N,M)}$).

3.4 Cooperativity Mechanics

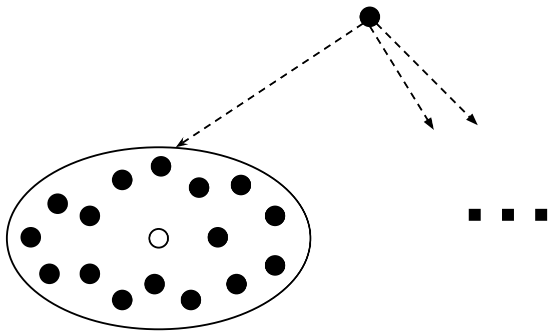
As mentioned previously, ErLam provides a number of feedback-enabled cooperativity-conscious schedulers for comparison purposes. However, before a detailed description of their function, we will discuss an overview of practical cooperativity recognition as well as our approach to designing mechanisms to take advantage of it.

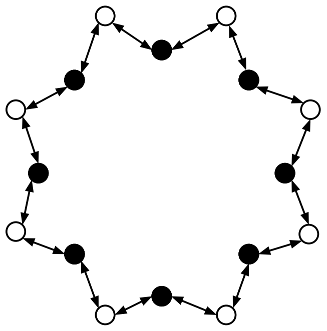
3.4.1 Overview & Approach

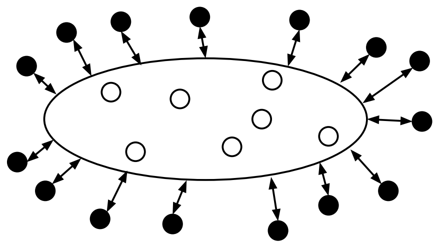
3.4.2 Longevity-Based Batching

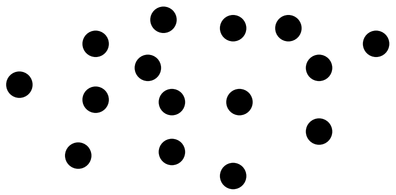
3.4.3 Channel Pinning

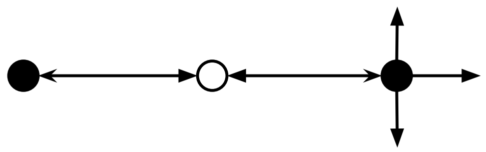
3.4.4 Bipartite Graph Aided Shuffling


(a) Graphical representation of $PTree$, N Parallel work groups.


(b) Graphical representation of $PRing$, full system predictable cooperation.


(c) Graphical representation of $ClusterComm$, N processes to M channels for unpredictable full system cooperation.


(d) Graphical representation of $ChugMachine$, N worker processes without cooperation.


(e) Graphical representation of $UserInput$, single randomly hanging process.

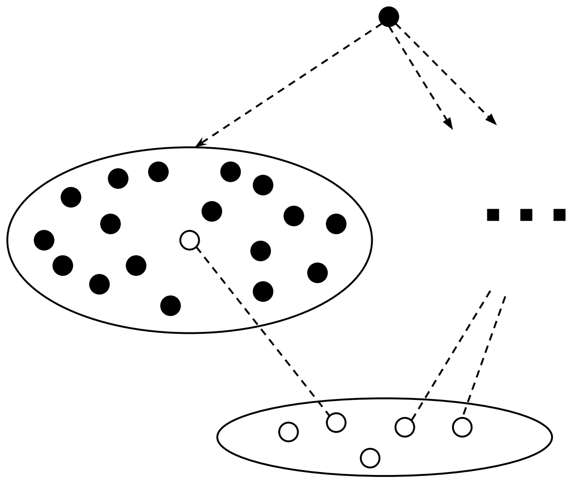

(f) Graphical representation of $JumpShip$, N Parallel phase shifting work groups.

Figure 3.9: Simulated behaviour examples.

Chapter 4

Results and Discussion

Chapter 5

Conclusion and Future Work

Appendix

Appendix 1

ErLam Operational Semantics

$$\text{Variable}\ \frac{E(x) \Rightarrow v}{S, C, E : x \to S, C, E : v} \qquad \text{Integer}\ \frac{}{S, C, E : n \to S, C, E : n}$$

$$\text{Fun}\ \frac{}{S, C, E : \textbf{fun}\ x\textbf{.}e \to S, C, E : \textbf{fun}\ x\textbf{.}e} \qquad \text{Unwrap}\ \frac{}{S, C, E : \textbf{(}e\textbf{)} \to S, C, E : e}$$

$$\text{NewChan}\ \frac{|C| + 1 = n \qquad C \downarrow n \Rightarrow chan_n}{S, C, E : \textbf{newchan} \to S, C; \{chan_n\}, E : chan_n}$$

$$\text{App(1)}\ \frac{S, C, E : e_1 \to S', C', E' : e_1'}{S, C, E : e_1 e_2 \to S', C', E' : e_1' e_2}$$

$$\text{App(2)}\ \frac{S, C, E : e_2 \to S', C', E' : e_2'}{S, C, E : \textbf{fun}\ x\textbf{.}e_1 e_2 \to S', C', E' : \textbf{fun}\ x\textbf{.}e_1 e_2'}$$

$$\text{App(3)}\ \frac{}{S, C, E : \textbf{fun}\ x\textbf{.}e_1 v \to S, C, E; (x, v) : e_1}$$

$$\text{If(1)}\ \frac{S, C, E : e_1 \to S', C', E' : e_1'}{S, C, E : \textbf{if}\ e_1\ e_2\ e_3 \to S', C', E' : \textbf{if}\ e_1'\ e_2\ e_3}$$

$$\text{If(2)}\ \frac{v \geq 1}{S, C, E : \textbf{if}\ v\ e_2\ e_3 \to S, C, E : e_2} \qquad \text{If(3)}\ \frac{v \leq 0}{S, C, E : \textbf{if}\ v\ e_2\ e_3 \to S, C, E : e_3}$$

$$\text{Swap(1)}\ \frac{S, C, E : e_1 \to S', C', E' : e_1'}{S, C, E : \textbf{swap}\ e_1 e_2 \to S', C', E' : \textbf{swap}\ e_1' e_2}$$

$$\text{Swap(2)}\ \frac{S, C, E : e_2 \to S', C', E' : e_2'}{S, C, E : \textbf{swap}\ c e_2 \to S', C', E' : \textbf{swap}\ c e_2'}$$

$$\text{Swap(3)}\ \frac{C(c, v) \Rightarrow \emptyset \qquad S \downarrow (S', e)}{S, C, E : \textbf{swap}\ cv \to S', C, E : e}$$

$$\text{Swap(4)}\ \frac{C(c, v) \Rightarrow (e, F) \qquad \{S \uparrow f \Rightarrow S' : \forall f \in F\}}{S, C, E : \textbf{swap}\ cv \to S', C, E : e}$$

$$\text{Spawn(1)}\ \frac{S, C, E : e \to S', C', E' : e'}{S, C, E : \textbf{spawn}e \to S', C', E' : \textbf{spawn}e'}$$

$$\text{Spawn(2)}\ \frac{S \uparrow f \Rightarrow S'}{S, C, E : \textbf{spawn}\ f \to S', C, E : 1} \qquad \text{Spawn(3)}\ \frac{}{S, C, E : \textbf{spawn}\ v \to S, C, E : 0}$$

Bibliography

[1] John L Bruno et al. *Computer and job-shop scheduling theory*. Wiley, 1976.

[2] Michael R Garey, Ronald L Graham, and DS Johnson. "Performance guarantees for scheduling algorithms." In: *Operations Research* 26.1 (1978), pp. 3–21.

[3] Richard D Dietz et al. "The use of feedback in scheduling parallel computations." In: *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings., Second Aizu International Symposium*. IEEE. 1997, pp. 124–132.

[4] Kurt Debattista, Kevin Vella, and Joseph Cordina. "Cache-affinity scheduling for fine grain multithreading." In: *Communicating Process Architectures* 2002 (2002), pp. 135–146.

[5] Robin Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

[6] Catuscia Palamidessi. "Comparing the expressive power of the synchronous and the asynchronous Π-calculus." In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1997, pp. 256–265.

[7] Don Jones Jr, Simon Marlow, and Satnam Singh. "Parallel performance tuning for Haskell." In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. ACM. 2009, pp. 81–92.

[8] John H Reppy. "Concurrent ML: Design, application and semantics." In: *Functional Programming, Concurrency, Simulation and Automated Reasoning*. Springer. 1993, pp. 165–198.

[9] David R White et al. "Automated heap sizing in the poly/ML runtime." In: *Trends in Functional Programming* (2012).

[10] Kunal Agrawal et al. "Adaptive work-stealing with parallelism feedback." In: *ACM Transactions on Computer Systems (TOCS)* 26.3 (2008), p. 7.

[11] Yuxiong He, Wen-Jing Hsu, and Charles E Leiserson. "Provably efficient online nonclairvoyant adaptive scheduling." In: *Parallel and Distributed Systems, IEEE Transactions on* 19.9 (2008), pp. 1263–1279.

[12] Carl G Ritson, Adam T Sampson, and Frederick RM Barnes. "Multicore scheduling for lightweight communicating processes." In: *Science of Computer Programming* 77.6 (2012), pp. 727–740.

[13] Thomas L. Casavant and Jon G. Kuhl. "A taxonomy of scheduling in general-purpose distributed computing systems." In: *Software Engineering, IEEE Transactions on* 14.2 (1988), pp. 141–154.

[14] Hagai Abeliovich. “An empirical extremum principle for the hill coefficient in ligand-protein interactions showing negative cooperativity.” In: *Biophysical journal* 89.1 (2005), pp. 76–79.