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 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 perhaps 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, and rely on dynamic dependencies that change over time.

Tasks however, in functional language verbiage, are typically called *processes* due to the inherent isolation this term brings and the language paradigm calls for. Instead, message passing is a common alternative to, and sometimes abstraction of, shared memory. Message passing is akin to emailing a colleague a question. You operate asynchronously, and your colleague can check her mailbox and then respond at her leisure. 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. For example, consider a purely functional `pmap`, where all workers are given subsections of the list. Each worker thread will have no need to access another's subsection and thus no messages will need to be passed. However, what happens when the function being mapped on a particular subsection uses several processes? Each may access a shared resource via message

passing. We would see a close coupling in this case. This highlights the granularity of process coupling, in that the `pmap` workers exhibit course-grained coupling, which allows the scheduler greater flexibility to run them in parallel. The opposite is true for the processes which show close coupling, like the mapped function. We define this granularity of process coupling as **Process Cooperativity**.

There exists a large number of mechanisms that scheduling systems can use in an attempt to improve work-load across all processing units. Some of these mechanisms 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 improve operation.

Process Cooperativity is an interesting metric by which to grade a system. In biochemistry the term cooperativity is 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 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 especially true in functional languages as the language assumes shared-nothing by default. Also, just as compilers can optimize using language constraints, so can a run-time using the language implementation. We will therefore examine possible message passing designs and how their implementation might effect our schedulers.

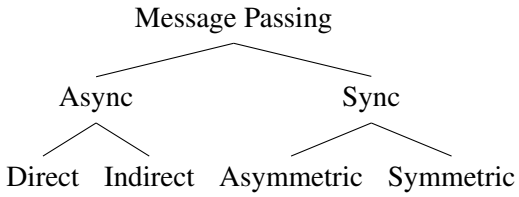


Figure 2.1: A High-Level Message-Passing Taxonomy

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 send the message directly to another process like in our example of emailing a coworker; these implementations are aptly named mailbox message passing. Another implementation sends the message indirectly via a rendezvous point, like a socket.

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 forces a block somewhere along in the operation, to push the message into some shared storage. Asynchronous operations require the additional capabilities

to both store and resend a message at a later time, so at some point, a process will need to synchronize; this will therefore, inevitably need to be behind the scenes.

In terms of scheduling, a language with asynchronous message passing will not hint much in regard to whether progress is being made. If a consumer process requires a value before continuing and therefore is repeatedly trying to receive from the channel, the schedule for the system would be better served by coming back to that process at a later time rather than repeatedly looping. However, asynchronous can benefit from process placement so as to take advantage of possible gains in cache affinity [1].

For example, the effects of the cache on direct message passing (*e.g.* a process mailbox) can be substantial if two processes on different cores share a location to store and therefore check for content. This shared location, if accessed from two cores will have to be updated in possibly multiple locations and validated for consistency at the cache level. However, if the two processes are local to the same cache there will be time saved in context switching and cache-line validation. In indirect message passing the task can be even worse as it's common that more than two processes may need access to the same space. Note, however, that recognizing cache effects can also lead to gains in synchronous message passing.

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 [2] or standard Π-Calculus [3], 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 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.

It's worth noting that asynchronous message-passing can be simulated using synchronous channels with a secondary buffer process. But by simulating it in this fashion we, as the scheduler, elevate the problem of cache locality to a problem of process locality. The same methods suggested to alleviate some of the lost efficiency due to cache locality [4,

$$\frac{P \overset{\sigma}{\Rightarrow} P_i \uplus P' \qquad P_i \Downarrow \sigma \to (P_i', \sigma') \qquad \{P_i'\} \uplus_{\sigma'} P' \to P''}{S(\sigma, P) \to S(\sigma', P'')}$$

Figure 2.2: A Runtime scheduler semantics.

5] are the same techniques which could be simulated for process locality; namely process batching and process affinity.

Note also, 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 core language we have chosen to base our semantics on symmetric synchronous message-passing.

2.2 Classic Runtime Scheduling

Operating Systems research have long been the leading front for scheduling topics. However, most of the early concern in scheduling was devoted to job scheduling over a group/shared system. As such, their concerns were largely devoted to fairness and job priority. They frequently had *a priori* knowledge regarding their jobs which gave them an opportunity to decide on a complete schedule beforehand.

Instead the systems for which we can schedule must be defined differently. A runtime scheduler may only have access to the processes which it observes, and can only speculate about their future based on their current state and any recorded historical information. These schedulers can be imagined to have a simple step like that of figure 2.2. In other words, for each time step, a scheduler must pick a process using what it knows about the world, and reduce it.

One possible selection mechanism could be to use a First-Come, First-Serve method, which means ordering the processes in a queue and running them as they spawn and enqueuing them as they block. However, if a particular process is computationally intensive,

processes involved with user-interaction for example would have to wait. This results in an obvious lag or hang in the system as the interactivity of the system stalls to finish computation.

To solve this problem a scheduler can $preempt$ a process after a certain amount of time has passed. This time slice is also called a time interval or a $quantum$ and has quite a literature involved with its selection. Too short, doesn't allow a process enough time to progress and the runtime system starts to spend more time context switching than computation. Too long and the preemptive-scheduler effectively becomes non-preemptive as all computation-bound processes hog the CPU from the interactive ones.

cite

After preempting a process, the scheduler has a choice as to where the process is placed. The common choice is to place it at the end of the queue, his behaviour is called Round-Robin. It is a common choice, not necessarily because of its simplicity, but due to its fairness. Each process in the queue is guaranteed an equal amount of time on the CPU and starvation of processes can therefore never happen. However, this isn't always the case as it's based on how process spawning is implemented. For example, if the newly spawned process was placed at the front of the queue or preempts the currently running process, a fork-bomb like process could hog the CPU and effectively shut out all other processes. Spawning to the end of the queue is the only effective way to avoid these scenarios in Round-Robin.

This, however, has all been using the assumption of a single process queue. While it is possible to implement a single global queue for all P processors, we will eventually get into an issue of contention where all the processors are attempting to take or add a process to the queue while another one is. However, in the event of multiple process queues there needs to be a mechanism in place for dispersing the processes across them all in an even or fair way.

There are two mechanisms for this, *work-sharing* and *work-stealing*. In work-sharing, the processor with more than enough work to do, will offload any new processes onto another (either randomly or by some heuristic). In work-stealing, it's the scheduler with the empty or small process queue that contacts another scheduler (either randomly or by some heuristic) so as to steal one or more. In the case of work-stealing the victim processor can be working on a process while another processor steals from it. This means that the cost of performing the process transfer is potentially hidden by the parallelism gained. However, in the case of work-sharing there is always an additional cost involved on

top of the time of execution as the overloaded processor must wait to work until after the transfer is complete.

This is why most schedulers which support multiple processing units utilize some work-stealing implementation. Of the implementations, there are two which we would like to highlight as they are provided by the ErLam toolkit: Shared-Queues and Interrupting-Steal. The Shared-Queues work-stealing scheduler allows other processes to directly access an end of their local process queue. This means, while a processor is potentially popping from one end of the queue, another could be stealing from the other end (assuming a lock-free doubly-ended queue like structure).

The alternative, Interrupting-Steal, has gone by several names like Work-Requesting, and Thief Processes. It's mechanism is to send a fake or dummy process to one or more other schedulers so when they run them they steal a process and send it back to its parent process. This reduces the overhead involved in synchronizing on the victim's process queue, but will instead stall it during the steal.

2.3 A Note on Control Theory

There has never been a scheduler which works optimally in all cases. Instead, focus has been the most fruitful when pursuing the optimization of various measurements using some particular objective function [6] to tune for particular edge cases. As such, scheduling based on such feedback metrics is not a new practice [7].

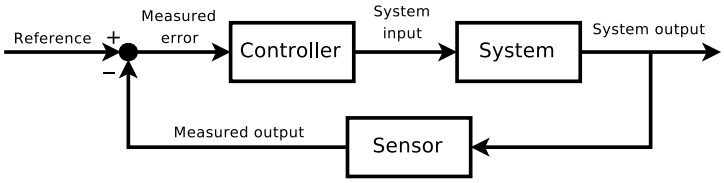


Figure 2.3: A classical feedback loop representation.

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 2.3).

The classic feedback loop starts by making reading the current state of the system and applying some operation to it (via the controller). The operation in some way affects the

system which can be observed by measuring particular metrics. These measurements can be fed back to the controller along with details about the system's output. If the controller wishes to modify the system further via the same or an opposite operation, it may do so. The canonical example is that of an automobile's cruise control. The controller can correct the speed of the vehicle by applying or releasing the throttle based on readings of the current speed.

In typical physical feedback loops there are two scenarios which need to be avoided: resonance and rapid compensation. Resonance in physical systems is when a spike in the amplitude of a system's oscillation can cause it to fail at a particular frequency. 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 work. But frequent or extreme damping or can stress physical systems to the point of failure as well.

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 interchangeable sensor, and controller modules. As such we make an effort to trace the feedback optimizations during our evaluation and explanation of the scheduler designs.

2.4 Feedback-Enabled Scheduling

Operating systems have also had motivation for designing intelligent feedback-enabled schedulers. As systems move away from perfect knowledge about the jobs it will be running, scheduling has needed to make guesses about the length of time jobs will need to run. A well-known example to this effect is called the Multi-Level Feedback Queue (MLFQ) scheduler, first described by Corbató *et al.* [8, 9].

The scheduler maintains N separate process queues, for N priority levels. All new processes would be spawned to the highest priority and would be subsequently demoted if they ended up running their whole designated quantum. However, a process may inadvertently game the system by running just up to the quantum before yielding. To fix this, after some time, S, the MLFQ is reset and all processes are boosted to the highest priority.

This helps with adapting to new system behaviour which may arise as well as coping with process starvation.

The goal of the MLFQ model is two-fold: to prefer interactive processes and to subsequently reduce the strain of computation bound processes on the overall system. This allows the system to prune the short-running processes out quickly and also maintain an adequate level of interactivity. A MLFQ implementer would also be able to heuristically set the quantum, N, and S based on the needs of the system as it's running, so as to introduce a second layer of feedback. For example, one could observe how much of a particular time period each priority queue is using. If a lower priority queue is being starved, it could trigger a reset [10].

The MLFQ idea in general is highly malleable and can be adapted to a number of situations. As such it transferred well into the level of runtime systems quite well. Concurrent ML (CML), uses this idea of a MLFQ to improve application interactivity.

CML is an extension to SML which adds the *spawn* function, and channel operations, among other things (such as asynchronous events) [11]. CML's scheduler defines a MLFQ where $N = 2$ and uses a single promotion algorithm instead of a reset. However, there is a key difference: CML uses process tagging to mark whether a process has communicated in the past.

As all newly spawned processes are appended to the primary queue, CML tells the difference between these newcomers and the short-running processes by tagging any process which makes a communication, or demoting it if not. A promotion can only happen if a previously marked process gets a demotion. However, the demotion process of a marked process is just a mark removal. Thus, the primary queue is essentially two queues in one.

CML's dual-queue system has the effect of reacting to new processes by testing them for longevity. It then makes an assumption about their behaviour immediately, but a process can change the scheduler's first impression of them through consistent behaviour to the contrary. A marked communication-bound process will, if it continues to use its entire quantum, eventually be demoted. A computation-bound process can eventually be promoted and marked as a communication-bound if it continues to communicate. Thus the system eventually adapts its behaviour to the new phase of the process.

However, recently an alternative mechanism has been utilized to adapt to system behaviour, that of process batching. The occam-π language, and specifically the Kent Retargetable occam Compiler (KRoC), allows processes which frequently communicate to be

batched and processed together [12]. This has two side effects: cache-affinity, and informed work-stealing.

The goal of the KRoC scheduler is primarily to take advantage of cache-locality when scheduling processes. It does so by reducing the chances for cache-misses by grouping processes which have a higher likelihood to communicate. The concept being, if two processes communicate, the data which is being shared will be in cache unless too many context-switches forces it out, thus place them close together in the queue. As a side effect of this, instead of stealing single processes, the KRoC schedulers will steal batches from each-other. This results in a quicker equilibrium in work-load saturation than stealing single processes.

Process migration between batches is done in two ways: 1. A channel synchronizes and causes the process to be de-scheduled from one scheduler and sent to the one which unblocks it. 2. A batch is split when more than one process in a batch is active, by popping the head of the batch into a new one. We explain this de-scheduling method in greater detail in Section 3.2.2, as we've implemented this mechanism for testing purposes. However, the mechanism absorbs a blocking process into the channel it's blocked on until another process unblocks it. At that time, the scheduler which unblocked it, now becomes its owner. Occam-π uses this mechanism as a method to build up batches of cooperating processes.

Ritson *et al.* mention however, that without a method to break up the batches, the system will eventually become one large batch. Therefore, whenever a new process joins a batch, the batch is allowed to split if there are more than one currently active processes within it (*e.g.* non-blocked or waiting processes). Thus, if a parent spawns a large number of processes (*i.e.* passed the batch size limit), the parent can start a new batch, while the batch of children can be stolen.

While KRoC's primary goal was cache-affinity, and CML's was optimizing interactivity, their feedback systems enabled a closer to optimal schedule than would have otherwise been possible with a classical scheduler focused on work-saturation. We now discuss another feedback metric, *Process Cooperativity*, which KRoC's scheduler, and our algorithms presented later, were able to benefit from.

2.4.1 Cooperativity as a Metric

Process Cooperativity stands out as a critical feedback metric in process-oriented programming. In fact the KRoC scheduler showed this through their performance gains.

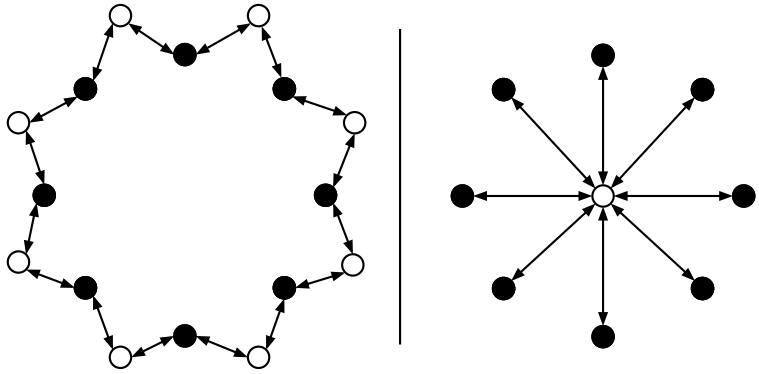


Figure 2.4: Two subcomponents formed by process cooperativity. Black dots represent processes, and the white dots represent a channel.

They showed that when a scheduler can recognize when two or more processes form a subcomponent, treating them that way improves cache utilization, reduces context switching time, and makes for smarter work-stealing. From this, we can take that recognizing cooperativity gives a good mechanism for determining a potential for fine-grained parallelism.

However, we would like to revisit the concern Ritson *et al.* expressed regarding when a component becomes too large. They introduced the mechanism of splitting batches based on an arbitrary max size of a batch, without regard to the substructure of the component expressed by the processes cooperation.

To illustrate this problem, figure 2.4 visualizes two possible components which may occur naturally. On the left we get a ring like structure, where the dependency of one process is the one to its right. We can abstractly envision a data-flow like application which acts like a token ring network. On the right we have a cluster of processes, all communicating with a random other on the same channel. We can envision this as an abstraction over a single shared resource.

Both the ring and cluster subcomponents would gradually become grouped into a KRoC batch. This is optimal for the ring component, no matter how large the ring may be. There is nothing to be gained from splitting it into multiple batches, and by doing so, we may actually hinder it. However, the cluster component may improve if given the chance for more parallelism, this would depend entirely on the longevity of the processes. KRoC attempts to account for this by recognizing if there are multiple active processes in a batch, and splitting an arbitrary one into a new batch (which ever happens to be at the head of the

process queue).

While this may not be avoidable based on observing structure alone, we may now run into an issue. Suppose all processes are active, but run for a length of time under their designated time quantum. At every preemption, when the size of the batch forces a split, we will create a singleton batch which must be reabsorbed after a single run. Ignoring the overhead, fairness properties also start to percolate. Namely, the processes within the batch after being trimmed will get preferential treatment to the processes in the singleton batches. This is exacerbated in the case of a single processor, as all singleton batches would need to wait for the large batch to run. Inevitably, the worst-case scenario for KRoC is below that of the work-stealing scheduler.

From this we can take that the longevity of a process can affect its cooperativity. In fact, looking at definition 1, we can apply it to a single process too. Now, a *process' degree of cooperativity* can be defined as its *frequency* of interaction with a set of channels. Thus, a cooperativity-conscious scheduler should also want to consider both the longevity of a process, and which channels it communicates with. This would give a much more complete picture of cooperativity.

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 [13].

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 discuss 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. Next, we will discuss the Scheduler API using an example scheduler implementation. We conclude this chapter with a summary of each of the classic schedulers that are included in the ErLam toolkit.

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

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 A. 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 a function is passed, it is applied with nil to initialize the internal expression in another ErLam process, and evaluates to 1 (*e.g.* true) in the parent process.

ErLam also makes a number of ease-of-use decisions like providing a default branch operator and a library system for providing a set of built in functions such as numeric operations, type checking, and standard functional behaviors (*e.g.* combinators, *etc.*). However, these built-ins will be largely ignored in this document but explained when neccessary.

ErLam also extends this base grammar with some useful syntactic sugar (see figure 3.2 for syntactic transformations) such as SML style let expressions and multi-variable function definitions (which are curried from left to right). We will use the syntactic sugar throughout this document to make our source easier to review.

Also, note the possible steps *swap* can take: either returning a block or another expression and a set of functions. On a semantic level, either event is transparent and results in blocking behaviour until a successful swap. However, in the former case, the channel has blocked and the only course of action for the scheduler is to get another expression to work

$$\textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2 \Rightarrow ((\textbf{fun}\ x.e_2)\ e_1)$$
$$\textbf{fun}\ x,y,z.e \Rightarrow \textbf{fun}\ x.(\textbf{fun}\ y.(\textbf{fun}\ z.e))$$

Figure 3.2: Syntactic sugar parse transformations.

on. In the later case, we have an expression to work on, but we also may have unblocked other processes by doing so, so we need to reschedule them. Note that in this case the function set may be null and the expression returned may be another attempt at swapping (*i.e.* $e = (\textbf{swap}\ c\ v)$). 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. Thus, there are several possible channel implementations we could provide while still adhering to the above semantics.

3.2.2 Channel Implementations

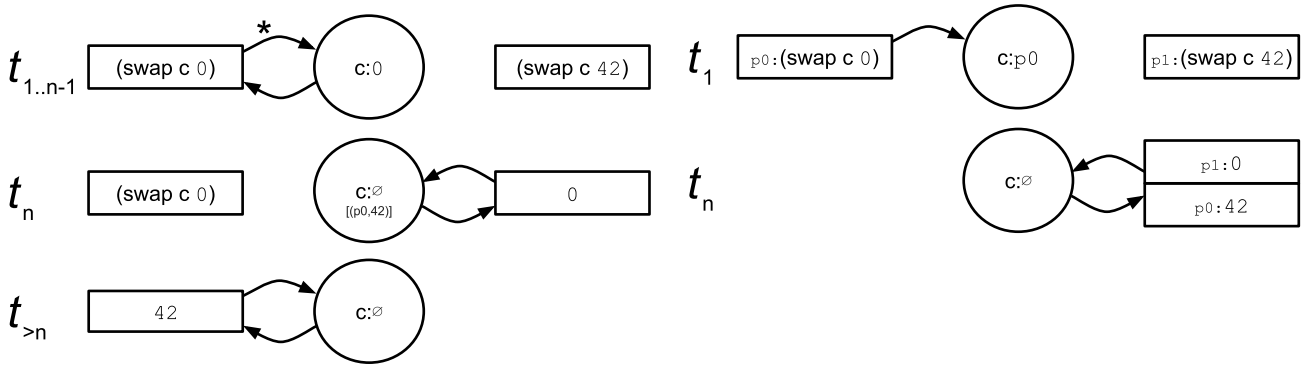
ErLam provides a selection of channel implementations to allow for interchangeable scheduler comparisons with different synchronization methods. We chose two channel implementations the *Blocking* Swap, and the *Absorbing* Swap as they highlight key differences for the runtime. We will now look at an example application and its 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, whose sole purpose is to swap on the channel the number 42 and quit. 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 (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 this behaviour of multiple swap attempts. We can see an illustration of this in figure 3.4(a). The first row shows the arbitrary time-slice t_1 where



(a) Process Blocking Swap

(b) Process Absorption Swap

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

the process swapping 0, $p0$, first contacts the channel with its value. The process may be scheduled again, repeatedly, to check the channel up to some arbitrary time-slice t_{n-1}. At t_n however, the process swapping 42 requests a swap and immediately gets a value back and logs that the process which it swapped with can get its value when it returns. Thus on the third line, for any arbitrary time-slice in the future $t_{>n}$, the process $p0$ can ask for a swap and get the value $p1$ stored.

Note the illustration makes no explicit mention of the scheduler or its functionality. It may be the case that the two processes are on different processing units and are in different 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 implementation 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 which evaluated the swap 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

```
-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

this type of message passing channel has provided enormous improvements for run-times 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, can take advantage of Erlang's callback behaviour specifications. An *erlam_scheduler* behaviour was defined which requires a minimum of 5 callback functions (figure 3.5).

Upon instantiation the runtime system will call the *layout/2* function with the Non-uniform Memory Access (NUMA) layout of the system that 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 hyper-threading. 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). Although we could of course have a scheduler which acts differently based on 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 by threading it through all subsequent callback calls. Upon shutdown, the opposite 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 time-slice (*tick/2*) and how a new process should be handled (*spawn_process/2*). An explanation of these callbacks is best done through example.

3.2.4 Example Usage: The CML Scheduler

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. Note also, that the *spawn_process/2* call happens on the same scheduler instance which evaluated the $spawn$. While this is not of consequence for this scheduler, a multi-core scheduler could be confident in appending a new process to its local queue without interfering with another LPU's scheduler.

In the original CML scheduler, it defined a quantum which it would let the current process run for, it would preempt it if it attempted to run for longer. The ErLam 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 emulates one step forward in the execution of the

```
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.

```
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.

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 involves 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 a thief-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 thief process which will syphon a process away to spawn on the thief's home scheduler. This blocks the process for a longer period of time, but does not involve accessing remote memory.

ErLam also comes with three cooperativity-conscious schedulers: the Longevity-Based Batching Scheduler (section 3.4.1), the Channel Pinning Scheduler (section 3.4.2), and the Bipartite Graph Aided Shuffling Scheduler (section 3.4.3). The first two build on

the same shared queue module as provided by $MTRRWS-*$, while the third utilizes it's own implementation.

For any compiled ErLam script, the runtime installs a command line option for selecting the scheduler used (among several other options). We are able to specify that we wish to run $pfib$, for example, with $MTRRGQ$ with the following command:

```
./pfib -s erlam_sched_global_queue
```

Any new schedulers can be added to the ErLam toolkit without needing to recompile the scripts as they are dynamically fetched and loaded at runtime.

3.3 Simulation & Visualization

The second primary goal of the ErLam toolkit was the ability to visualize how a scheduler proceeded to evaluate an ErLam application. We therefore 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. It would also be advantageous to be as finely grained as possible and leave it up to the visualization mechanism to dial the accuracy.

We also needed a sample set of application simulations to run our set of schedulers against. These simulations needed to be minimal to reduce extraneous data but still demonstrate various levels of cooperativity and phase changes. We would like to also have the ability to compose 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,""
983847.936003,4,queue_length,50
```

```
983847.936430,3,tick,""
983847.936439,3,reduction,""
...
```

The time-stamp 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 (see section 4.1.2 for details).

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 time-slice for a given LPU. This time-slice is where the averages come into play. R heatmaps have a maximum of 9 colors, so any range we select must be scaled to 9. However, the constant multiplicand is based on the mean amount of time N ticks take place across 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 advantageous on several occasions when de-

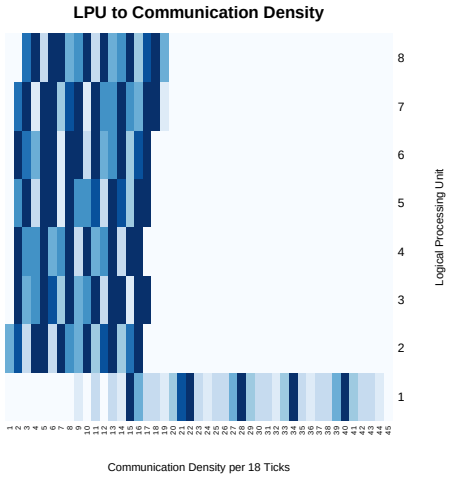


Figure 3.8: Example of Communication Density graph for the Work-Stealing scheduler on a Core i7 running the $PRing$ (defined in section 3.3.2) application.

bugging scheduler implementations. As decreasing the number of ticks to average together, increased the number of samples and thus accuracy.

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 common applications.

On one hand we have an axis depicting the amount of parallelism possible in an 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 receives 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 experiment with that, we would have to throttle the degree of cooperativity. This behaviour is called $ClusterComm_{(N,M)}$ (fig-

ure 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.

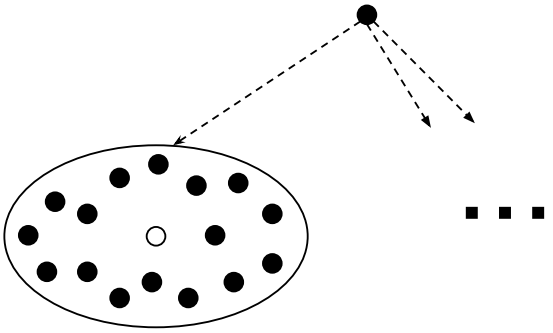
$ClusterComm_{(N,M)}$ is also an example of full-system cooperation, we also want to have a possible case for partial-system cooperation. We begin this range of experiments with a behaviour which acts like a bunch of $ClusterComm_{(N,1)}$ running in parallel. We call this special case behaviour $PTree_{(W,N)}$ (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/or $PRing_N$ running in parallel. For this reason, we made our behaviours composable.

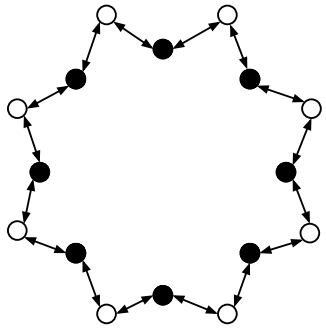
We are missing two important behaviour simulations: application phase changes, and hanging processes (typical of I/O bound processes). In the case of the latter, a simple built-in command $hang$ was provided which would simulate hanging for a random amount of time before allowing the process to proceed with evaluation. If a scheduler attempted to reduce the process before the $hang$ time was completed, it would be immediately pre-empted. The behaviour which implements this is called $UserInput_{(T,C)}$ (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". This simple behaviour would also compose with the others.

For the former missing behaviour, phase changes, we decided to make a variation of $PTree_{(W,N)}$ called $JumpShip_{(W,P)}$ (figure 3.9(f)) which would act like $PTree_{(W,N)}$ 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 $X + 1$ **mod** N.

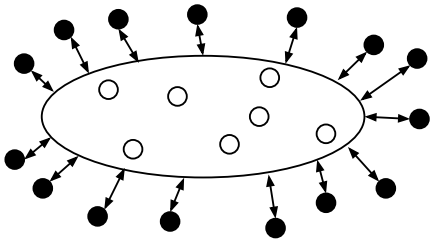
We would have liked to possibly experiment with variations on the "jump-ship" behaviour so as to inject phase changes into $PRing_N$ (by perhaps reversing direction) or $ClusterComm_{(N,M)}$ (by switching to $ChugMachine_N$ for a brief period before returning to $ClusterComm_{(N,M)}$). Yet time constraints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



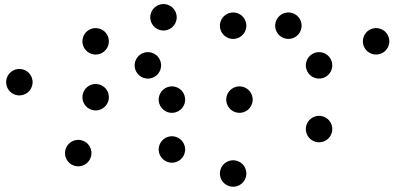
(a) Graphical representation of $PTree$, N Parallel work groups.



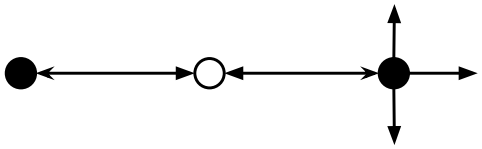
(b) Graphical representation of $PRing$, full system predictable cooperation.



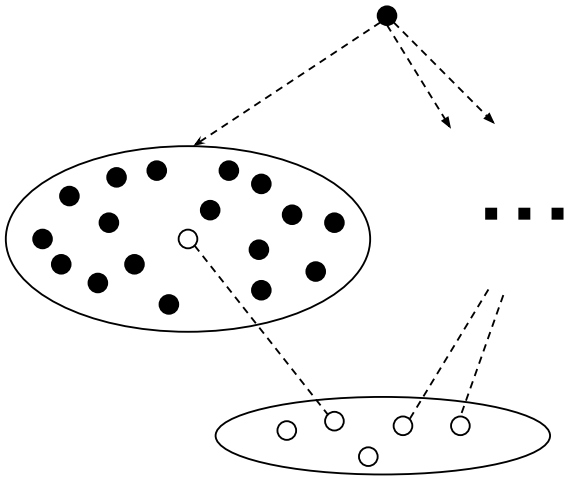
(c) Graphical representation of $ClusterComm$, N processes to M channels for unpredictable full system cooperation.



(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 and test primitives.

3.4 Cooperativity Mechanics

As mentioned previously, ErLam provides a number of feedback-enabled cooperativity-conscious schedulers for comparison purposes. Through our exploration of cooperative behaviour we noticed that cooperativity has, on some level, been captured in previous scheduling systems. Through techniques keeping in mind cache locality, channel efficiency, and work-stealing efficiency, we see a common theme of keeping processes which communicate together, in close proximity.

Ultimately cooperativity gives us a mechanism for recognizing when processes may be moving along the spectrum of application parallelism. We would therefore like to look at systems which recognise particular behaviours which utilize this.

3.4.1 Longevity-Based Batching

As we've mentioned before, batching processes together has been a common mechanism in capturing some of the lost efficiency of cache locality. To take advantage of the cache a message passing language is required to be considerate of their channel implementation to allow for it. For example, occam-π makes sure their channel fits into a single 32 bit cache line (technically two, as they have asymmetrical channels and split it based on send and receive).

However, ErLam elevates the concerns of cache locality into an issue of process locality by relying only on symmetrical message passing. Note that a channel is loaded into memory at the location (*i.e.* LPU) of whichever process first blocks it and then again at whichever process unblocks it. Thus to reduce the number of times the channel needs to be loaded, we can still use the batching mechanism. Namely, if two processes are communicating frequently, the channel never needs to be migrated if they exist in the same batch and thus on the same LPU. This mechanism would be applicable to any other language which uses message passing too.

However, this batching mechanism is great when the application is going through a phase of frequent communication and thus fine-grained parallelism. But in the event the phase changes or the system is course-grained, then a batch actually harms the ability of the scheduler to parallelize to the fullest. This issue however is one we've seen before, the struggle of computation and communication bound processes.

The CML Interactivity based scheduler breaks up the processes into groups of long running (computation-bound) and short running (communication-bound) processes. We

can take advantage of this in our case by kicking processes which are too long-running out of a batch (into a singleton batch for example). To merge communicating processes back together in the same batch would be as easy as turning on Process Absorption for our channels. A process would then be batched with processes that have all begun communicating on a particular set of channels.

The mechanism we just described has been implemented as the Longevity-Based Batching Scheduler. A process belongs to a batch as long as it does not exceed it's quantum (reduction count). If it is preempted, it is thusly kicked out from the batch it's a part of (unless it's a singleton). There are of course alternatives to this (*e.g.* allow for N quantum to pass before kicking a process from a batch), but we, for testing purposes, can just extend the size of the quantum for the same effect.

To gain entry back into a batch, all a process needs to do is perform a swap with Process Absorption turned on. Note as Process Absorption must be toggled on for this to work, we can experiment with max-parallelism possible in the application. Our assumption prior to experimentation was that, the longevity-batching scheduler would drop-down into a common work-stealing scheduler in the worst case (*i.e.* all processes are singleton batches). We talk more on these assumptions and their validity in Chapter 4.

3.4.2 Channel Pinning

An alternative approach to batching, which moves the channels to the processes, is to set an affinity to a core based on which channels you cooperate on. We call this mechanism Channel Pinning, as we bind a channel upon creation to a particular LPU and force processes to that location to perform a swap. There are a large number of interesting mechanics for this behaviour. But we will look at three: 1. How to spread the channel pinnings? 2. How should processes react when attempting a swap on a remotely bound channel? 3. And based on the decisions made in the previous, how should a scheduler steal/spawn a process?

When choosing a channel spread algorithm there are two things which need to be compared, cost of creating a channel and channel usage of the running application. In stark contrast to the previous scheduler, channel pinning would either need to be an expensive heuristic or a programmer-aided decision based on the application itself. For example, if our channel pinning algorithm was a sane even spread across all processors we would have ignored the possibility that a subset of the channels could be used more frequently.

This could therefore cause more harm than good. If we chose to do an expensive check across all processors to compute the saturation each time we create a channel we would be harming the application which uses a map-reduce style, and uses a lot of temporary one-use channels.

On top of this complication, this also ignores the possibility of phase changes. It may be the case that during a start up phase, the frequency of particular channel usage may offset the saturation to such a degree that any further checks will suggest alternate processors despite possibly being inaccurate. However, these issues are beyond the scope of this discussion and we will instead focus on scheduling around channel pinning. As such we only provide the following channel pinning implementations: 1. *same*, which pins the channel to the same processor it is created on (*i.e.* the processor which evaluates $newchan$). 2. *even*, which pins the channel in a round-robin fashion starting at LPU 0.

Note *same* pinning, would not be ideal for an application which creates all channels in one process and then hands them out to it's children like in $JumpShip_{(W,P)}$, or $ClusterComm_{(N,M)}$. However, *even* pinning will be perfect for them. We expect experimentation with composed application will have interesting effects in this scheduler.

We have a couple of possible mechanisms for how a process should be handled when it comes time to communicate. In the event it is wanting to communicate on a channel that is not local, we propose the following mechanism: let them anyway but if they block, spawn them to the LPU which owns the channel.

The theory for this is, in the event of a block, the local LPU won't gain anything from having it in its queue (if process absorption is turned on it would loose it anyway). However, if the process completes a swap, both the remote LPU and the local can continue in parallel.

Due to this selective spawn feature, we have the opportunity to look at a selective steal opportunity. Namely, when stealing we can attempt to grab from a random scheduler, one or more processes which have communicated with a randomly selected channel which the thief owns. This is akin to the children's card game "Go Fish" where in our case a scheduler may ask if another "has any channel 3's".

However, with this mechanism it may be the case that there is never any work for a particular scheduler. In this case, we have added the post condition that if a process has never communicated, or if the scheduler is not the owner of a channel, then they act as wild

cards and can match anything. This is both a simplistic algorithm, but it also makes sure the scheduler falls back to a standard work-stealing algorithm by default.

We now note that the primary interest in comparing this scheduler is to look at this work-stealing mechanism. We would like to see how this type of mechanism fairs against both a best and worst case scenario. We first thought the worst case scenario would be a $ChugMachine_N$ due to the reliance on wild-cards, however after more consideration a single $ClusterComm_{N,1}$ may end up having a worse behaviour due to the constant re-spawning of processes back to the owner of the primary channel. A best case in this scenario would be a $PTree_{(W,N)}$ when W is greater than or equal to the number of processing units available.

3.4.3 Bipartite Graph Aided Shuffling

Both of the previous schedulers ignore the effects ordering can have on execution behaviour. We hypothesize that order of process execution could have a drastic consequence on highly cooperative processes by pairing channels such that if a process where to block, the unblock would happen as soon as possible (*i.e.* the scheduler would not choose a process which had no probability of unblocking it).

Granted the extreme of this type of scheduler would be excruciatingly unfair, and as such we still maintain a round-robin like behaviour, except now we rely on a sorted process queue. Sorting the process queue would allow us the chance to increase the likelihood of immediately unblocking a blocked channel, while still maintaining execution fairness. Sorting the process queue would also have the aided side effect of making work-stealing potentially more efficient when stealing more than one from a victim queue. Namely, it would be much more likely that the group of processes stolen are cooperating.

Due to this, we hope to show an interesting case where process absorption may not be the preferred channel implementation. The blocking channel, one which will immediately return, will allow for the process to stay sorted and allow for the next process following it to unblock it for it's next turn. Thus relying on work-stealing alone to migrate processes in a potentially smarter and more-grouped way.

There are three metrics which could effect the order of processes, and which would subsequently trigger a resorting of the process queue. A process yielding, returning, or spawned could all mean that a particular process or set of processes should to be relocated.

However, frequent sorting may cause the scheduler's fairness to suffer. We therefore set a variable, Γ which defines how many "events" can happen before causing a resort.

To provide a mechanism for sorting, we structure our process queue as a bipartite graph, where one side is our process queue, and the other is the set of channels. We generate a "pseudo-priority" based on recency of the communications over the number of channels it's communicated with. Namely, we sort the process queue by $\Delta(P_i) = \Sigma E_i/|C_i|$ where E_i is the edge set of timestamps, and C_i is the set of channels P_i communicated with. The default priority function, Δ, is intended to give our processes which communicate frequently, a head start, and all other processes can be pushed to the end (and more likely stolen).

If our process queue is long, we maintain preferential treatment for the set of interactive processes. If our process queue is short, then the effects of sorting are negligible. As such, we should only be concerned with sorting a process queue after a particular size. This is another chance for heuristic analysis. Due to this though, we would expect to see poorer behaviour on most simple simulated applications, but would steadily gain in advantage when subjected to composed applications.

Chapter 4

Results and Discussion

4.1 Evaluation

We begin our discussion by first stepping back and performing a meta-validation of the ErLam toolkit itself, before comparing the scheduling mechanisms amongst themselves. We ask:

- How complex must our implementations be to create our test primitives?
- For each scheduler, does our implementation work as expected, despite a minimalistic scheduler API?
- Does the channel implementation lend itself to scheduling improvements? If so, in what case?

These questions will evaluate ErLam, both as a language and runtime, but also as a simulator for scheduler experimentation, comparison, and evaluation. We attend to these questions in order; Section 4.1.1 critiques the language as a medium for simulation design through the development of our test primitives. Section 4.1.2 discusses our evaluation of the scheduler API using several of the testing primitives on the aforementioned classical schedulers. Then in section 4.1.3, we discuss our findings regarding channel implementation differences and their subsequent effects on scheduling behavior.

4.1.1 Test Case Implementation

Our intentions when choosing our base language constructs were primarily focused on simplifying the base language. This minimalism we hoped would remove any noise which may be caused by the implementation details. We hoped to make, for lack of a better term, a concurrent functional assembly language. As such, there was some concern as to the level of ease we would be able to implement our testing primitives.

We start with a critique of the first implemented test case: $ChugMachine_N$. It soon became apparent that we would need to standardize on a successive spawning method

```
merge = fun a.(let x = newchan in
               let _ = (spawn fun _.(swap x (a nil))) in
               fun b.(let y = newchan in
                      let _ = (spawn fun _.(spawn y (b nil))) in
                      fun m.(m (swap x nil) (swap y nil))))
// ...
(omega fun f,n.(if (leq n 1)
                   (worker_t nil)
                   (merge fun _.(f f (dec n)) worker_t ignore))
            N)
```

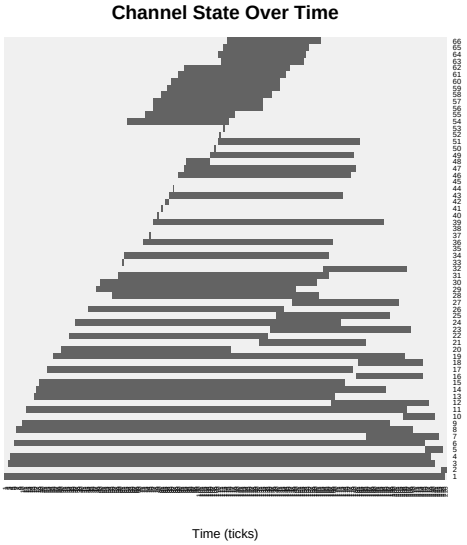
Figure 4.1: Our implementation for a map-reduce style fork-branch, and it's subsequent standardized usage.

for producing several of the same process at once. Rather than modifying our $spawn$ expression we decided to implement this as a fuction in the language as it was. The reasoning behind this was to keep consistent the effects of spawning a process into a process queue. If we allowed the process to produce N processes at once, the lag caused by the runtime to produce them would be more noticable.

We settled on the $merge$ function (figure 4.1), along with a standard method of invoking it. Our $omega$ function would enable recursion, for us to use one branch of the $merge$ call to create a left-loaded binary tree. This would keep a consistent behavior despite a bias towards the initial processes. Note that a work-stealing or global-queue scheduler would gain access to the most recent processes sooner (and would thusly get a chance to reduce sooner).

Also as a side effect of our decision, there is a channel for every process which immediately blocks. This has the added effect of giving us the ability to track when processes terminate, as the long-term blocked channels will start closing. For example, figure 4.2, gives an example parallel Fibonacci program written in this style and it's subsequent channel graph. This is reminiscent of a map-reduce style approach, and the language lends itself to it. Note on the channel graph, a dark line indicates a channel which becomes blocked until the point at which it becomes unblocked.

You may also note that the chart gives an indication as to the size of the time quantum selected for the application's execution. The order in which a channel becomes blocked, and it's order in the processes execution give that hint. The large block of channels at the top of the graph indicate the last batch of spawned processes got time on the cpu before the one's which spawned it, as the scheduler which evaluated the spawn ran out




```
(omega fun f,m.(
    if (leq m 1)
       m
       (merge fun _.(f f (sub m 1))
              fun _.(f f (sub m 2))
              add)) 8)
```

Figure 4.2: Parallel Fibonacci implementation and a potential channel graph.

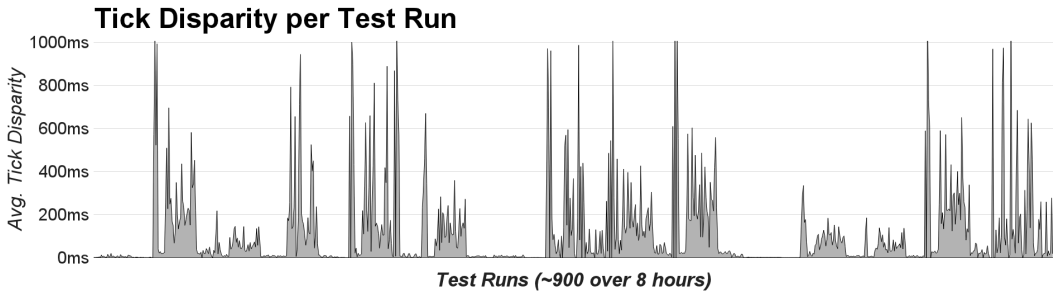
of reductions for that process. These observations will become useful for later scheduler critiques.

4.1.2 Scheduler API

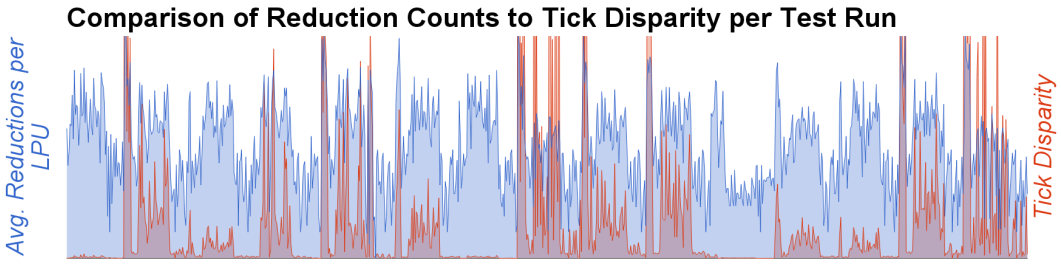
The ErLam Scheduler API was minimally constructed around the single-step scheduling semantics presented earlier in figure 2.2. We were motivated by the simplicity of the description and thus the ability to bring some formalism to the implementations. That being said, we would still require the option to observe practical statistics such as runtime overhead during our scheduler comparisons.

However, wne of the concerns early on, and as described in section 3.3.1, was that the LPUs could greatly differ in the number of ticks they are able to provide their process queue. This could be caused by the OS preempting the ErLam scheduling thread to execute something else. However, from our tests, we found these gaps to be minimal and mainly caused by the scheduling implementation itself.

Figure 4.3(a) shows the average time between ticks averaged over the LPUs for a large subsection of the tests we ran. We call this average time between ticks per LPU the Tick Disparity of the test. From the figure, we notice obvious clustering, and figure 4.3(b)



(a) Tick disparity, or the average time between scheduler ticks per LPU, per Task run. Typical ranges tend to spike in groups, which show consistency based on scheduler implementation, rather than OS intervention.



(b) Tick disparity consistently matches up with increased computation, which is indicitive of inter-scheduler communication requirements.

Figure 4.3: Tick Disparity over nearly 900 tests.

leads us to believe this is primarily caused by spikes in reduction counts. A scheduler chugging on processes the entire time, must wait until preemption in-order to handle any inter-scheduler communication, as in the case of work-stealing schedulers. Thus, for our purposes, all talk of tick-disparity could be considered a discussion of scheduler overhead. As such, comparative analysis of scheduler implementation overhead from test case execution would still be a valid comparison with our design.

Further, to validate that our scheduler implementations functioned as expected after translation was another concern. We gave the example of the CML Dual-Queue translation ($STDQ$) previously in section 3.2.4. We will now examine it's execution and compare it to $STRR$, a single queue naive scheduler to confirm our understanding.

The key differences we would expect to see in a comparison would be that the CML prefers to, and attempts to run the interactive processes first. It would push all computational processes onto the secondary queue, only promoting them when the need arises. To observe this property, we compose the $UserInput_{(T,C)}$ test with $ChugMachine_N$ to cre-

ate an interactivity test primitive $Interactivity_{(N,M)}$ (figure B.6). The primitive launches $ChugMachine_N$ and M instances of $UserInput_{(5,10)}$ (the values of which were arbitrarily chosen so as to execute for long enough to collect coherent data). We then subsequently ran this primitive with $STRR$ and $STDQ$ using multiple values for N and M. Table 4.1 compares an instance of this test set: $Interactivity_{(8,16)}$.

The Reduction Density graphs explain everything we need to know. The darkened portions at the beginning of the chart indicate that the $STRR$ scheduler has no regard for the interactive processes, whereas the $STDQ$ scheduler remains spread out. This seems to be on the opposite if we were to consult the Communication Density charts though. It would appear here that $STRR$ is getting more frequent communication, and $STDQ$ condenses all synchronizations to the end. This is however an issue of verbiage as Interactivity does not correlate to Communication Density. By this we mean, $STRR$ is indeed allowing communication to happen evenly due to a round-robin schedule, but it is not attending to spontanious events (*i.e.* responses back from $hang$). $STDQ$ on the otherhand pushes all inter-process communication backwards as it waits to respond to the $hang$ing processes.

We see this effect more clearly once we compare with the Channel State charts. In $STRR$ we see an even regard for all processes. The tapering of the channel blocks at the beginning of the graph is consistent with our understanding of the $merge$-based successive spawning. However, $STDQ$ is completely different, and all $UserInput_{(T,C)}$ processes seem to be pushed to the beginning of the execution pool. This is infact due to another difference between the execution styles of the two schedulers. $STDQ$ replaces the currently running process with the child process it spawned, enqueuing the parent. We therefore can confirm our understanding, in this case, that the scheduler is reacting to the test primitives as expected.

The work-stealing mechanic is another instance for interesting comparison. We've built two of the cooperativity-conscious schedulers we discuss on top of the $MTRRWS$-$*$ process queue implementation. It is this unified queue implementation which allows us to toggle between the two implementations. With $MTRRWS$-SQ, the process queue will respond to function calls from schedulers for other LPUs, it will perform a quick dequeue from the bottom only blocking the host LPU from accessing the queue for a minimal amount of time. With $MTRRWS$-IS, the process queue goes untouched by other LPUs, instead the scheduler on the LPU itself will receive the messages and respond during periods inbetween process operations.

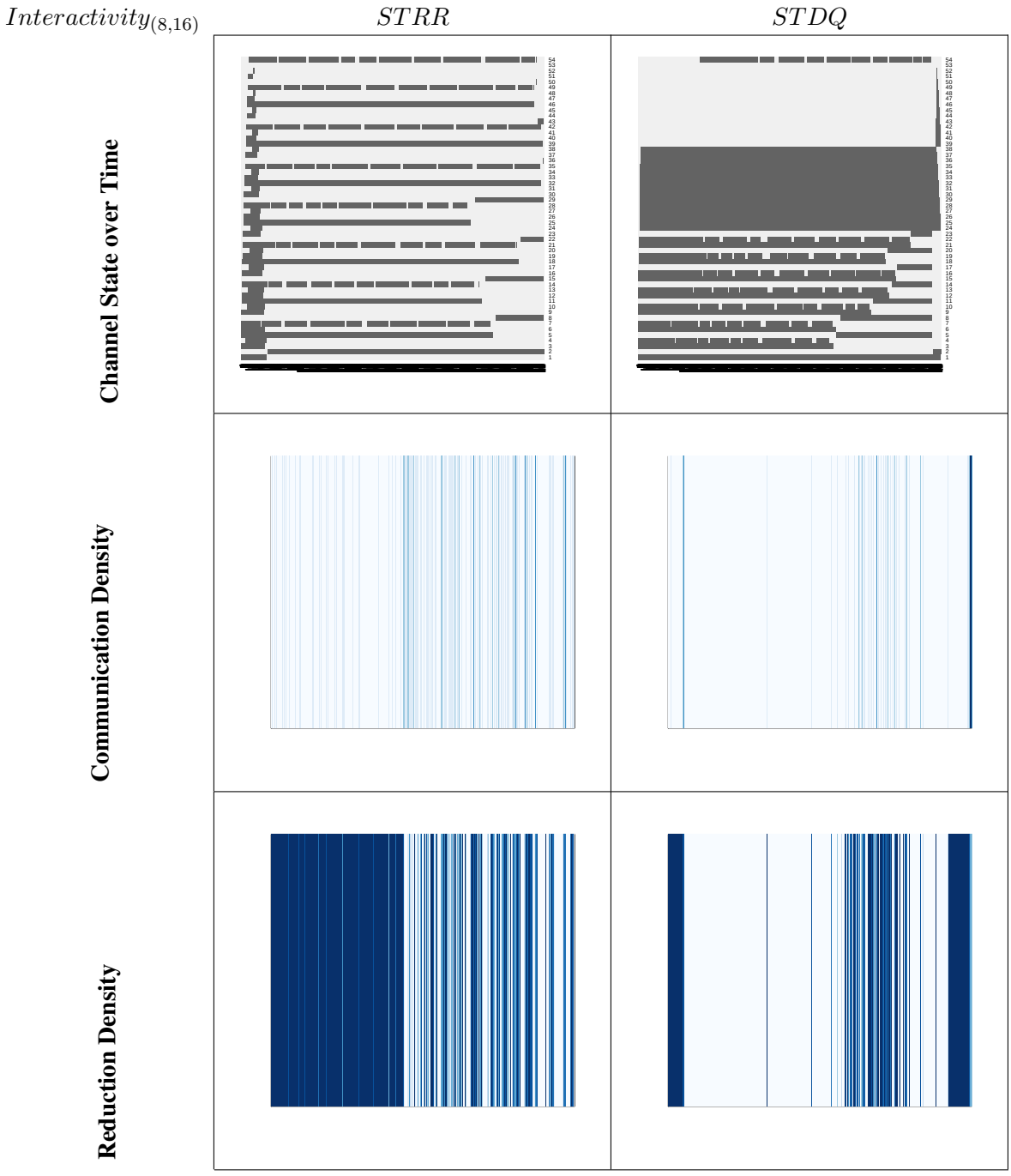


Table 4.1: Comparison of $STRR$ and $STDQ$ using Channel State, Reduction and Communication Densities.

To compare these, we would expect to see minor differences in the processes scheduler state as we could expect $MTRRWS\text{-}IS$ to wait longer for responses from their theif messages. But otherwise we would like to make sure the saturation of the cores are as identical as possible. In other words, if either method is unable to saturate the cores effectually, then it would not be worth further testing (or there is an issue with the implementation).

However, work-stealing schedulers work best if there is a constant probability of loosing a process by completion. Our test primitives, as they stand, do not account for application phases in terms of process quantity. Thus, to effectively examine these stealing mechanisms, we substitute process loss via completion with process loss via channel absorption. Thus, table 4.2 gives a comparison of the two work-stealing techniques in terms of scheduler state over time and the process queue length with a run of $PTree_{(9,10)}$ with channel absorption turned on.

Upon review of this comparison, our initial assumptions are validated. The process queue length looks extremely similar. The LPU which owns the initial process maintains the largest queue, where all others work steal a single process and work on it. Any gains in queue size for these LPUs are via channel absorption, but this can not compare with the initial spawns. We note here that an obvious introduction of smarter work-balancing would be critical for a realistic or even practically useful scheduler. However, for our purposes of verifying the work-stealing mechanism, this is sufficient.

The scheduler state comparison is however, in this instance, not consistent with all the original assumptions. Namely, $MTRRWS\text{-}IS$ seems not stay in waiting mode as long as $MTRRWS\text{-}SQ$ does. However, we can pose two possible reasons for this reaction: there is an obvious scaling bias introduced by squeezing over 11000 ticks into the same length of space as SQ does with 8000. Alternatively, the scheduling reduction quantum for this execution was 20 and all processes chug for only 5 max, this allows the scheduler more opportunity to tend to inter-process communication at a detriment to the shared-queue implementation due to higher chance of blocked queue access.

4.1.3 Channel Implementations

The above graphs (*e.g.* table 4.1, and table 4.2) were generated, using the CML-like Process Absorption channels. We would like to now breifly visualize how a blocking channel may effect the scheduling of an application.

Give example of two runs with the same test/scheduler but with ca and cb.

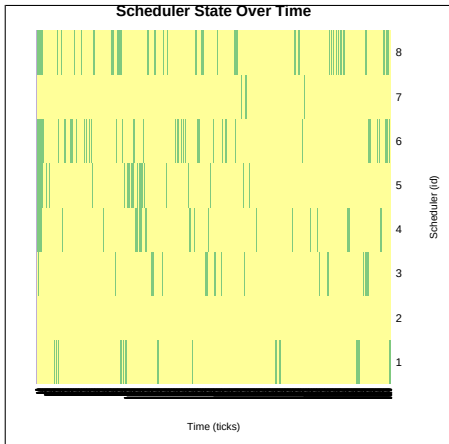
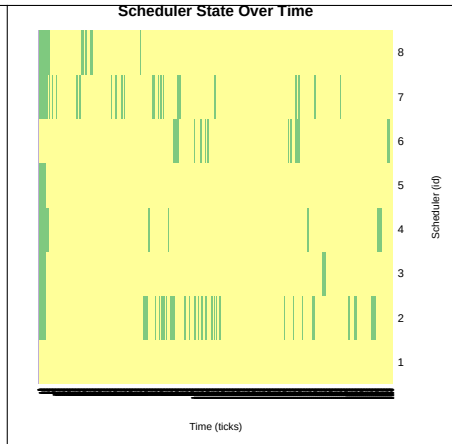
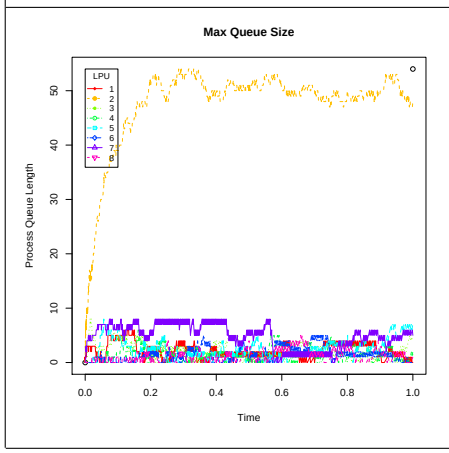
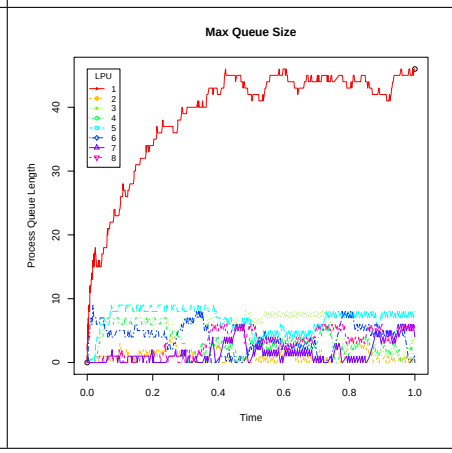
$PTree_{(9,10)}$	$MTRRWS\text{-}IS$	$MTRRWS\text{-}SQ$
Scheduler State		
Process Queue Length		

Table 4.2: Comparison of $MTRRWS\text{-}IS$ and $MTRRWS\text{-}SQ$ using Scheduler State and Queue Size.

4.2 Cooperativity Mechanics

We now turn to the discussion of cooperativity-conscious scheduling. Using our classical scheduler results as a base line we can evaluate the feedback schedulers independently. This will give an indication of what may warrant further investigation, and testing. However, there are three questions which are of immediate interest:

- Are there instances where the stealing mechanism matters?
- When does being cooperativity-conscious lead to better schedules?
- Does the overhead of feedback systems outweigh the benefits?

4.2.1 Longevity-Based Batching

4.2.2 Channel Pinning

4.2.3 Bipartite-Graph Aided Shuffling

4.3 A Comment on Swap Channels

Swap channels provided a number of benefits on the side of the language designer. They reduce the complexity of channel implementation, and as shown, they lend themselves to a number of possible designs. We demonstrated just two possible implementations, the Blocking and Absorption channels. As such, the concept of a swap channel as a language primitive is extremely attractive. However, swapping poses some problems for realistic applications which we would now like to discuss.

First, a swap channel does not lend itself to a level of fairness that would be expected by a programmer. We lead with our implementation of $ClusterComm_{(N,M)}$ as example, which due to being on top of swap channels, was made to be much more enigmatic. Figure 4.4 gives an example implementation of $ClusterComm_{(N,M)}$. Note the third parameter to the application, which denotes the number of time's each of the N processes should communicate. The system therefore blocks until all processes synchronize X times before quiting.

Due to this, we must pose several restrictions on the possible values of N and M. Namely, N must be an even number, since all processes would need to have a partner to swap with, and M must be no greater than $\lfloor N/2 \rfloor$, any more and it would be possible for a process to hang indefinitely.

```
fun N,M,X. // Number of Processes, Channels, and times a Process should Swap
(
    // Create the list of channels for random access.
    let chanlist = (omega fun f,L,s.(if (leq s 0)
                                        L
                                        (f f (cons newchan L) (dec s)))
                    newlist M)
    in

    // Randomly swap, by selecting random channel in chanlist.
    let rswap = fun _.(swap (index chanlist (rand M)) nil)
    in

    // Worker Thread, loops for X times and randomly swaps.
    let worker_t = fun _.(omega fun f,c.(if (leq c 0)
                                           1
                                           (ignore (rswap 0)
                                                   (f f (dec c))))
                                X)
    in

    // "Main" Function, spawns N worker threads and waits for their return.
    (omega fun f,c.(if (leq c 1)
                       (worker_t nil)
                       (merge (fun _.(f f (dec c)))
                              worker_t
                              ignore))
     N)
)
```

Figure 4.4: A naive but ineffectual $ClusterComm_{(N,M)}$ implementation.

However, these two restrictions are not enough to guarantee the process terminates. In fact, most runs of this application, with any value of $N > 2$ would most likely hang forever. This is due to the bias our program creates when spawning processes, as well as the type of fairness the swap channel semantics provides.

First, our bias we introduce is merely because we cannot batch spawn a set of processes at the same time. As such we will spawn one process at a time which may get a chance to run before all others. As such the first several processes may reach their synchronization limit before we are even done spawning the rest of the processes. Due to this, we may have a case where all but M processes have completed, and thus all channels are blocked indefinitely.

Secondly, the channel semantics have no inherent preference for unseen or new processes. The scheduler may easily get in a loop of running the same subset of processes repeatedly, this would have the same effect as the above even if we were able to solve the bias problem. As such, this is inherently an issue with the capabilities of swap channels.

Thus, the best we can do for $ClusterComm_{(N,M)}$, is to run until at least $N - M$ processes have met their quota. Note this approach is only acceptable under Symmetric message passing constructs. In asymmetrical, even if there was a guarantee of an equal number of senders and receivers, all senders could be blocked on one channel while all receivers could be blocked on another.

But this issue points to another problem with swap channels, insofar as they do not lend themselves to being primitives at all. Due to this fairness issue, a language with swap channels would be unable to build the asymmetrical constructs most user's would like. As such, they have been useful merely for simulation purposes.

However, for further simulation of cooperativity, it may be adventagous to also consider the directionality of communication. The recognition of consumer and producer processes may lend itself to further gains as the recognition of communication and computation bound processes did. This is not to say all gains in utilizing pure synchronization have been obtained.

Chapter 5

Conclusion and Future Work

5.1 The ErLam Toolkit

5.2 Effectiveness of Cooperativity as a Metric

5.3 Future Work

There are a number of appealing avenues of improvement for the ErLam toolkit. The report generation mechanism could be extended and tied into the logging system a bit more closely. For example, the implementation of a real-time viewer would be an interesting extension. There are obviously a larger number of metrics which may lead to better cooperativity classifications as well. It may be more fruitful, for example, to keep track of communication partners rather than channel names. The core language is also an appealing simulation implementation language, as such a larger library of testing primitives would benefit the language designer community greatly. Furthermore a complete catalog of parameterized executions would also aid in analysis and scheduler comparison.

In terms of cooperativity as a feedback metric, it would be interesting to further tweak the three cooperativity-conscious schedulers already implemented. Perhaps composing the scheduler mechanics themselves may lead to a more robust algorithm. For example, the combination of the Channel Pinning scheduler and the Bipartite-Graph Aided scheduler may complement each other. However, in short, the future of cooperativity as a feedback metric looks promising.

Appendices

Appendix A

ErLam Operational Semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix B

ErLam Test Cases

We provide here, for the reader's analysis, the source code of the ErLam test cases we utilized throughout the testing and evaluation phases of our document.

```
//
// ChugMachine: Parallel non-cooperative processes
//
fun N, // Number of Processes
    A, // Minimum number of reductions per worker.
    B. // Max number of reductions per worker.
(
    // WORKER:
    let worker_t = fun _.(chug (add A (rand (sub B A)))) in

    // MANAGER:
    (omega fun f,c.(if (leq c 1)
                       (worker_t nil)
                       (merge (fun _.(f f (dec c)))
                              worker_t ignore))
                   N)
)
```

Figure B.1: ChugMachine application, generates N processes to just compute.

```
//
// UserInput: Single Event processing simulation
//
fun X, // Number of events to consume before quiting.
    H, // Max number of seconds to wait for an event.
    L. // Min number of seconds to wait for an event.
(
    // Event Channel
    let chan = newchan in

    // User
    let get_event = fun c.(ignore (hang (add L (rand (sub H L))))
                                  (swap chan c)) in
    let user_t = fun start,_.(omega fun f,c.(
                        if (leq c 0) (swap chan nil)
                                     (ignore (get_event c)
                                             (f f (dec c))))
                               start)
    in
    let _ = (spawn (user_t X)) in

    // Consumer/MAIN
    (omega fun f.(if (swap chan nil) (f f) 1))
)
```

Figure B.2: UserInput application, utilized in the Interactivity composure, it artificially hangs without anything to do for a random period of time so as to simulate an interactive process.

```
//
// ClusterComm: Parallel Full-App Communication
//  Simulates unstructured communication across all processes in the
//
fun N, // Number of Processes in the system.
    M, // Number of channels for synchronization.
    X. // Number of times each process should synch.
(
    // Result Notification:
    let win_chan = newchan in
    let result_chan = newchan in
    let wait_for = (sub N M) in
    let _ = (spawn fun _.(omega fun f.(ignore (swap win_chan result_chan)
                                             (f f)))) in

    // Channel List setup:
    let chanlist = (omega fun f,l,s.(if (leq s 0) l
                                    (f f (cons newchan l) (dec s)))
                    newlist M)
    in
    let rswap = fun _.(swap (index chanlist (rand M)) nil) in

    // WORKER:
    let rchug = fun n.(chug (rand n)) in
    let finish = fun _.(omega fun f.(let r = (swap win_chan nil)
                                     in (if (is_chan r)
                                            (swap r 1)
                                            (ignore (rchug 5) (f f))))) in
    let worker_t = fun _.(omega fun f,c.(if (leq c 0) (finish nil)
                                            (ignore (rswap 0) (f f (dec c)))) X)
    in

    // MAIN FUNCTION:
    let _ = // Spawn Worker Set
        (spawn fun _.(omega fun f,c.(if (leq c 1)
                                        (worker_t nil)
                                        (ignore (spawn worker_t)
                                                (f f (dec c))))
                                  N))
    in // Hang for Result
        (omega fun f,c.(if (leq c 0) 1
                           (ignore (swap result_chan nil) (f f (dec c))))
                wait_for)
)
```

Figure B.3: ClusterComm application, generates N processes and M channels, and waits for at least $N - M$ processes to finish synchronizing X times.

```
//
// PRing: Simulate full-system cooperativity via a ring network
//
fun N, // Ring size in number of processes.
    L, // Number of times to pass the token around the ring.
    C. // Number of times to chug at each loop around the ring.
(
    let token = 1 in
    let kill_token = 2

    // An individual node in the ring.
    in let node = fun rside,lside,_.(omega fun f.(
                let recv = (swap rside nil)     in
                let _    = (chug C)             in
                let dead = (eq kill_token recv) in
                let _ = (swap lside recv)       in
                    if dead nil //(printl 1)
                            (f f)))

    // Parent node which will count the loops.
    in let pnode = fun rside,lside.(omega fun f,c.(
                    let recv = (swap rside nil)                    in
                    let dead = (eq c L)                            in
                    let _ = (swap lside (if dead kill_token recv)) in
                        if dead nil //(printl 1)
                                (f f (inc c))))

    // Start the Parent node after starting the token passing.
    in let start_pnode = fun rside,lside.(
                            let _ = (swap lside token) in
                            (pnode rside lside 1))

    // Spawns a new node with a channel and returns it's left side channel.
    in let spawnnode = fun rside.(
             let lside = newchan in
                 (ignore (spawn (node rside lside)) lside))

    // Generate the ring and connects the last two. Then triggers the
    // ring to start passing.
    in let mychan = newchan
    in let last_chan = (rep N spawnnode mychan)
    in (start_pnode last_chan mychan)
)
```

Figure B.4: PRing application, simulates full-system cooperativity, where N processes pass a token around a ring network.

```
//
// P-TREE: Parallel Cooperative Sets
//  Spawns W work-set and M cooperative processes per set. The processes
//  then run a random number of reductions and then proceed to synchronize.
//  They do this X times before stopping.
//
fun W, // Number of Work-Sets
    M, // Number of Members per Work-Set
    X. // Number of times members should synch
(
    let clustercomm = (
{{{ClusterComm.els}}}
    ) in

    // A workgroup thread of M processes and X syncs
    let workgroup_t = (fun _.(clustercomm M 1 X)) in

    // Run W workgroups
    (omega fun f,w.(if (leq w 1)
                       (workgroup_t nil)
                       (merge (fun _.(f f (dec w)))
                              workgroup_t
                              ignore))
            W)
)
```

Figure B.5: PTree application, utilizes composure of ClusterComm to simulate partial-system cooperativity as a set of work-groups.

```
//
// Interactivity Testing
//
fun X, // Number of UserInput processes to spawn.
    Y. // Number of processes to spawn in the ChugMachine
(
    // IMPORTED FUNCTION DEFINITIONS
    let userinput = (
{{{UserInput.els}}}
    ) in
    let chugmachine = (
{{{ChugMachine.els}}}
    ) in

    // User Input thread and generation loop.
    let ui_t = fun _.(userinput 10 5 1) in
    let ui_l = (omega fun f,c.(if (leq c 1)
                                  (ui_t nil)
                                  (merge ui_t
                                         fun _.(f f (dec c))
                                         ignore)))

    // Spawn X User inputs and Y computational-bound processes
    //  - Wait for all to finish before returning.
    in (merge (fun _.(chugmachine Y 200 150))
              (fun _.(ui_l X))
              ignore)
)
```

Figure B.6: Interactivity Application, utilizes a composure of UserInput.els and ChugMachine.els to test a scheduler's ability to handle interactivity.

```
//
// JumpShip: Parallel Workgroups which hop channels
//
fun N,M,C, // Number of Processes, Channels and Chugs.
    T,X, // Number of Rounds and itterations per Round.
(
    // List Access
    let chanlist = (omega fun f,l,s.(if (leq s 0)
                                        l
                                        (f f (cons newchan l) (dec s)))
                    newlist M) in
    let next_chan_index = fun i.(if (eq i M) 1 (inc i)) in

    // WORKER:
    let worker = fun cl,init,_.(
                let run_round = (fun c.(ignore (chug C) (swap c nil)))      in
                (omega fun f,r,i,c,n.(
                            if (leq r 0) 1
                               (if (gt i 1)
                                   (ignore (run_round c) (f f r (dec i) c n))
                                   (let new_n = (next_chan_index n) in
                                    let new_c = (index cl new_n) in
                                    (f f (dec r) X new_c new_n))))
                 T X (nth cl init) init))
    in

    // MANAGER:
    let worker_t = (worker chanlist) in
    let workers_per_chan = (div N M) in
    let manager_t = fun ci,_.(omega fun f,c.(
                                if (leq c 1) (worker_t ci nil)
                                             (merge (fun _.(f f (dec c)))
                                                    (worker_t ci)
                                                    ignore))
                             workers_per_chan)
    in

    // MAIN Function: - Spawn M managers for each initial channel group.
    // The groups all start at their respective index in the channel list (i.e.
    // group 1 is index 1, group 2 is index 2, etc.) On each round their channel
    // index is incremented and modulo M.
    (omega fun f,c.(
            if (leq c 1) (manager_t c nil)
                         (merge (fun _.(f f (dec c)))
                                (manager_t c)
                                ignore)) M)
)
```

Figure B.7: JumpShip application, similar to PTree and ClusterComm, except posses to demonstrate application phases as the work-groups change channels.

Bibliography

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

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

[3] 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.

[4] Evangelos P Markatos and Thomas J LeBlanc. "Load balancing vs. locality management in shared-memory multiprocessors." In: (1991).

[5] Evangelos P Markatos and Thomas J LeBlanc. "Memory-Conscious Scheduling in Shared-Memory Multiprocessors." In: *Computer Science Dept., University of Rochester* (1991).

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

[7] 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.

[8] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. "An experimental time-sharing system." In: *Proceedings of the May 1-3, 1962, spring joint computer conference*. ACM. 1962, pp. 335–344.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 0.80. Arpaci-Dusseau Books, 2014.

[10] Kenneth Hoganson. "Reducing MLFQ scheduling starvation with feedback and exponential averaging." In: *Journal of Computing Sciences in Colleges* 25.2 (2009), pp. 196–202.

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

[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] 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.

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

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

[16] 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.

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

[18] 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.

[19] 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.