MS Thesis Proposal

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

Alexander Robert Dean
ard4138@cs.rit.edu

Committee Chair: Dr. Matthew Fluet, Ph.D.
Reader: Dr. James Heliotis, Ph.D.
Observer: Dr. Rajendra K. Raj, Ph.D.

Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

May 11, 2014

Abstract

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 a unique feedback mechanism and scheduling algorithm which takes advantage of cooperative behavior. It further compares this algorithm to other feedback metrics via a custom extensible runtime system developed to support swappable scheduling mechanisms, around a minimalistic language with interesting characteristics, which lend themselves to easier statistical metric accumulation.

1 Background

Since the formalization of feedback driven systems and the advent of Cybernetics, multiple fields have attempted to mold these principles to their own models; and run-time schedulers are no exception. This is due, in part, from process scheduling in parallel systems being fundamentally an NP-Complete problem [2]. Instead, focus has been more fruitful when pursuing the optimization of various metrics using some particular objective function [5] to tune for particular edge cases. As such, scheduling based on feedback metrics is not new [4].

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

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

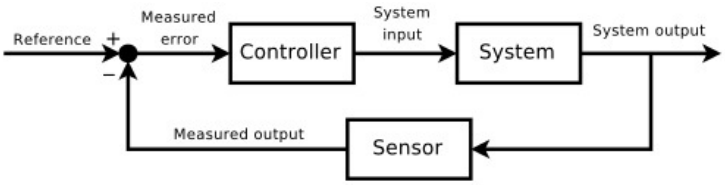

Figure 1: A classical feedback loop representation.

Also, note that run-time systems tend to extend to more than just the process-scheduling systems. They also encompass garbage collection, dynamic type checking, statistical collection and other debugging mechanisms, and general resource allocation (*e.g.* I/O). For the purposes of this paper we will be exclusively talking about the process scheduling system and intentionally ignore the effects of these other, albeit important, sub-systems. This distinction must be made however, as some scheduling systems also take into account the effects of other sub-systems on the placement and order of process evaluation. For example, White *et al.* [9] discuss heap size as a metric for process selection as a means of curtailing garbage collection effects.

1.1 Classical Run-Time Scheduling

Another distinction must be made as far as the level of foresight the scheduling systems have, at least, within this paper. There is a spectrum of clairvoyance in classical job-scheduling, in that on one end, job-schedulers have full foresight over the jobs which will enter the queue and their order. These schedulers have the opportunity to optimize for future events, which is a luxury the scheduling systems that this paper discusses do not have.

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

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

1.2 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. Message passing in general can be broken down into two types, asynchronous or synchronous, and then further by how they are implemented. However, when discussing process scheduling the method of their implementation is often of some consequence .

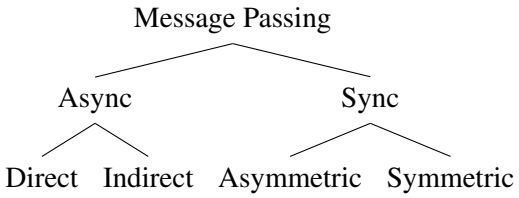

Figure 2: A High-Level Message-Passing Taxonomy

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

In terms of scheduling, a language with asynchronous message passing can ignore the effects of blocking operations, but will need to look closer at process placement to take advantage of possible gains in cache affinity [3]. For example, the effects of the cache on direct message passing (*e.g.* a process mailbox) can be substantial as two processes on different cores will need to copy stack frames to memory and back rather than just a pointer to the message in the originator's stack frame. In indirect message passing the task is even worse as multiple processes may need access to the same data.

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

Asymmetrical message passing is synonymous with Milner's Calculus of Communicating Systems [6] or standard Π-Calculus [7], in that you have a sender and a receiver which will block on their respective functions around an anonymous channel until the pass has been completed. This differs from a symmetrical message passing in that the only operation on the channel is a blocking function which swaps values with the process at the other end.

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

by the scheduling of the system (*i.e.* the order at which the channels evaluate the $swap$ command can be directly governed).

1.3 Example Scheduling Feedback Systems

A basic feedback scheduler can be explained using a metric like 'waiting time vs running time'. A standard round-robin scheduler may utilize a queue of processes and attempts to be fair by giving every process an equal chance to run. However, in a contrived example such as a fork-bomb, the scheduler may never reach the end of the queue and the rest of the system could starve. In our example feedback enabled system each process could be ranked according to its ratio of waiting time over running time. Thus as time went on, even in the event of a fork-bombing effects the beginning of the queue would still get some time to run.

There is quite a literature behind the types of metrics used though. Yet statistical collection is only a third of the feedback loop. A classification of the system must be made by a control mechanism based on the recorded state, and then a modification must be made to the scheduler which should positively effect the state of the system's future. The key is to make sure this effect is transparent and results in minimal overhead, while still achieving the objective function.

This objective function, for the purposes of simplicity is typically execution time. However, it can also take into account resource utilization, process/channel fairness, fulfillment of real-time guarantees, *etc.* to rate the schedule as a whole. Note that this function is an external measure (evaluation) of the schedulers effectiveness, which will be brought up further in section 3.3. For now however, note that the objective function and control metrics are separate and evaluate two different things.

Before discussing Cooperativity as a feedback metric though, it is important to introduce previous metrics and their respective control schemes. Interactivity is one metric which evaluates the system's current scheduler. It looks at the ratio of communication to computation currently taking place in the system as a whole. One scheduling system which utilizes this metric indirectly is in CML [8].

It's method by which it grades the system is the recognition of *computation*-bound processes versus *communication*-bound processes. A *communication*-bound process is one which has previous blocked itself without using it's entire scheduled quantum, the rational for this is only a synchronizing thread would be willing to block; thus a *computation*-bound process is the opposite (*i.e.* one that always uses its full quantum). After labeling the processes the scheduler pushes *computation*-bound processes to a secondary queue so they don't eat up all of the time. This improves the time the scheduler can spend attending the processes which need to synchronize, which in turn improves the Interactivity of the application (*i.e.* GUI widgets will finish their message passing while the back end computes).

1.4 Cooperativity as a Metric

As mentioned, note that the Interactivity metric looks only at the ratio of *communication*-bound processes to *computation*-bound processes and misses the quantity of communication on a per-process level which could indicate needed changes in scheduling quantum or process locality with those which are frequently in touch. This hints at a need to look for some level of cooperation between processes in the phases of an application's life-time.

Rather than a ratio of *communication* to *computation*, instead we need to look at a frequency of communication and between which processes (*i.e.* rate at which a process communicates with a particular channel). One method this can be achieved by is, in addition to recording which processes are *communication* and *computation*-bound, the scheduler can also tag the process with it's frequency of communication on each

channel it touches over time. By doing so, the scheduler can calculate a process's cooperativity with one or more processes by comparing frequency counts on particular channels, perhaps by a sum of ratios.

Cooperativity, in the general sense, has been a probabilistic interpretation of how two actors can be affected by the actions of others. This has been a sociological term as well as a biological one. Abeliovich [1], thusly defines a notion of positive cooperativity as:

> A process involving multiple identical incremental steps, in which intermediate states are statistically underrepresented relative to a hypothetical standard system [...] where the steps occur independently of each other. Likewise, a definition of negative cooperativity would be a process involving multiple identical incremental steps, in which the intermediate states are overrepresented relative to a hypothetical standard state in which individual steps occur independently.

In other words, while a process can be cooperative, a system can be shown to possess *positive* or *negative* cooperativity as the set of cooperative processes increases or decreases (respectively). This lends itself quite readily as a metric for an evolving system of phases. Assuming the cooperativity can be directly probed we can check it against an evolving standard state the scheduler proposes. The feedback mechanism would be to attempt to raise or lower the cooperativity of the system to get it back to this standard state. It can do so by increasing or decreasing the synchronization allowance each scheduler gets per a given time frame for example. This would have the effect of bounding the frequencies of communication.

2 Motivation

There are a few motivating examples based on conceptually common scenarios. The first is a Ring based structure where a token and possibly more data is passed predictively around a set of n processes across a set of logical processing units.

In this scenario there is a distinct structure to the process layout and the order of evaluation. However a simplistic work-stealing scheduler may have no notion of this and the current token-owning process may randomly jump from one processing unit to any other without regard for possible inefficiencies. A cooperativity conscious scheduler may find that a single core running them in sequential order is preferable.

Another scenario could be the Map-Reduce based structure where a parent spawns several children (possibly communicates with them) and then waits for their return (which is another round of communication). This demonstrates the effect of phases within an application, and has an opposite signature from the previous example in that the n processes spawned should all run at the same time. In this case a cooperativity conscious scheduler may find that following the simplistic work-stealing approach works best.

The overall motivation is thusly, twofold. I believe there is an additional feedback metric that has gotten little publicity despite it's apparent usefulness. Current feedback scheduling implementations fail to take process-cooperativity into account when applying feedback. A scheduler implementation using this metric calculation would aid in future feedback schedulers, as well as possibly introduce simplistic but beneficial modifications to current scheduling systems.

However, this hints at other missing metrics that could be useful for adapting to application phases. A closer evaluation of current scheduling techniques on a core set of common language primitives would allow for a positive comparative analysis. Thus the first step would be to synthesize a subset of primitives useful for metric gathering while introducing as little noise as possible.

From the examples given, the internal characteristics provided by most message passing systems can still look synchronous to the scheduler and seem to give little leeway as far as order of evaluation and priority. Otherwise, there may be bottle necks in asynchronous message passing as the run time system accesses

the underlying channel representation (*e.g.* share memory space, message box, *etc.*). In either event it seems likely that a simplified symmetric synchronous message passing primitive would lead to interesting techniques as far as scheduling is concerned.

Given this messaging primitive, cooperativity metrics should follow from monitoring frequency of interaction with these channels between partners of processes. Yet, as mentioned, these assumptions need to be tested on an even playing field among other scheduling techniques and using a diverse set of example simulations.

3 Proposed Work

I have already begun to work on a compiler built with swappable scheduler's in mind. The language is a simple concurrent lambda and process calculi with synchronous swap channels so as to minimize the language characteristics which may mask the process-based metrics, such as differentiating between CPU, I/O, and Channel bound processes.

This compiler will also allow me to directly test multiple metrics and scheduling techniques as well as visualize the system phases at any given point. I will attempt to do a comparative analysis of one or more popular scheduling techniques against a new algorithm I am designing which uses cooperativity. This, in turn, will evaluate the effectiveness of the process and communication channel abstractions utilized in the language.

Thus this section is divided into four sub-sections: first, section 3.1 will discuss the language, some initial assumptions about it's implementation, the abstractions made and the rational for these decisions; next, section 3.2 explains the scheduling interface the compiler will provide and what schedulers I intend to port for use in the ErLam run-time system; then, section 3.3 will discuss the evaluation methods I intend to use on the cooperativity scheduler and all ported schedulers; finally, section 3.4 will finalize what the outcomes are for this thesis as well as the required deliverables.

3.1 The ErLam Language

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

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

The hope is that now it should be easy to recognize *computation*-bound processes (as they do not block before a set number of reductions), *communication*-bound processes (as they will synchronize and block on a swap channel), level of cooperativity between processes (*i.e.* from keeping track of which processes swap with which others), and any other metric we can think of (as we still have stack, process locality, channel information, *etc.*).

3.2 Ported Schedulers

Ultimately it would be interesting to test a large collection of scheduling mechanisms and metrics within the same framework. However with time constraints I will limit it to the most promising and comparable

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

Figure 3: The ErLam language grammar, without syntax sugar or types.

feedback schedulers in use.

The first is CML's interactivity-based scheduler. This scheduling system uses a simple but very effective two-queue method for segregating *computation* and *communication*-bound processes so as to have preferential treatment of the communication in the system without sacrificing too much computation time.

The last is Occam-π's process locality-based scheduler which attempts to catch a portion of cooperation by batching processes into groups and only work-stealing groups. Their scheduler looks very promising as it has been shown to take much more advantage of the cache than simple work-stealing schedulers. However, the implementation complexity of this scheduler may push this to being a stretch goal rather than a required deliverable.

These two schedulers will need to be ported along side the new cooperativity scheduler for side-by-side testing. It would also be required to have a baseline for these tests so a simple non-feedback-enabled round-robin work-stealing scheduler will also be implemented.

3.3 Evaluation

There are two portions of this proposal which will require evaluation: the schedulers and the language abstractions:

Scheduler Comparisons: The schedulers will be compared directly using quantitative analysis of several metrics. Namely I will be looking at each schedulers' queues, the number of blocked/running/waiting processes, and the timeframes in which each of the processes are being evaluated. Using a consistent set of tests, such as the Ring and MapReduce examples given previously, it should be possible to evaluate what consistent behaviors each scheduling algorithm produces over multiple runs.

Language Abstraction Evaluation: After the schedulers have been compared there's now the objective evaluation of each scheduler on the language itself. It may be possible that the scheduling systems exhibit interesting behaviors within the abstractions I enforced (namely swap channels and it's implementation). This step can be performed during the implementation and testing phases of the road-map and will primarily be summarized in weekly blog posts and in the language implementation section of the final report.

To perform the scheduler comparisons I will need to take a number of statistics about the scheduler's themselves. These will be separate from the scheduling process itself, but could be useful in analyzing the operation of the scheduler over time and as the application's phase shifts. Possible metrics for scheduler evaluation are as follows:

Process Location: If a process keeps getting juggled, or a particular LPU does not steal correctly, it would be helpful to know where all the processes are and the paths they take.

Swap Channel Timing: This will catch how long it takes to actually perform a swap. Which should, in conjunction with knowing process location (logical unit ID), capture cache affinity over time.

Process State Timing: A process can be either Blocked, Waiting or Running. By keeping track of each and how long a process has been in each state before switching we can check on possible fairness concerns.

3.4 Outcomes and Deliverables

The end goal will be to have, at minimum, the scheduling algorithm based on process cooperativity and the ErLam compilation framework for testing it along-side other algorithms, completely implemented. The framework will have two parts: A new tool for comparative analysis of the inner workings of concurrent schedulers, and a set of high level language abstractions for future run-time scheduler testing.

Thus, there are two deliverables expected alongside the Thesis Report; the source code for a new compiler and runtime framework, and the implementation and design description of a new scheduling algorithm. Both of which will be made available for public use after completion.

4 Roadmap

Based on the layout of the 10-week Summer session of 2138, where the tenth week is the defense. It will primarily be a top-heavy load that will shift as needed with the inevitability of roadblocks:

May
4/28 - 5/02 - Thesis Proposal
5/25 - 5/31 - Finish base ErLam Compiler and Plug-in Scheduler Interface

June
6/01 - 6/07 - Port CML's interactivity scheduler to Erlang*
6/08 - 6/14 - Cooperativity based Algorithm Development/Implementation
6/15 - 6/21
6/22 - 6/28 - Implement Test Cases for Scheduler Comparisons

July
6/29 - 7/05 - Draft of Thesis report submission *(hand off to Readers**)*
7/06 - 7/12 - Run tests and compile results *(update Thesis)*
7/13 - 7/19 - *(Schedule Defense)*
7/20 - 7/26

August
7/27 - 8/02 - Primary Thesis Defense dates
8/03 - 8/09 - *(Backup Defense dates)*

* *Stretch goal to implement a batching Occam-Pi scheduler also.*

** *The goal is to get the thesis to the readers as far ahead of time as possible, this week is buffer.*

Every week will contain at least one meeting with my chair and every two weeks must result in an update to my website to lay out my progress and future work. I intend to update my Thesis report as the session progresses, my Chair will be able to view the progress through a shared version-controlled repository. The Readers will get two weeks with the draft (without finalized results) to make suggestions before accepting a defense date. I will have three weeks concurrent to this time for testing and report generation.

References

[1] Hagai Abeliovich. An empirical extremum principle for the hill coefficient in ligand-protein interactions showing negative cooperativity. *Biophysical journal*, 89(1):76–79, 2005.

[2] John L Bruno, Edward Grady Coffman, RL Graham, WH Kohler, R Sechi, K Steiglitz, and JD Ullman. *Computer and job-shop scheduling theory*. Wiley, 1976.

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

[4] Richard D Dietz, Thomas L Casavant, Mark S Andersland, Terry A Braun, and Todd E Scheetz. The use of feedback in scheduling parallel computations. In *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings., Second Aizu International Symposium*, pages 124–132. IEEE, 1997.

[5] Michael R Garey, Ronald L Graham, and DS Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, 1978.

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

[7] 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*, pages 256–265. ACM, 1997.

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

[9] David R White, Jeremy Singer, Jonathan M Aitken, and David Matthews. Automated heap sizing in the poly/ml runtime. *Trends in Functional Programming*, 2012.