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.

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

April 28, 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 scheduling system. As the application's phase shifts, the feedback mechanism is tasked with modifying the scheduler to reduce it's overhead and increase the application's efficiency.

Cooperativity is another 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 mode as the number of processes increase.

This work proposes a unique feedback mechanism and scheduling algorithm which takes advantage of this behavior. It further compares this algorithm to other feedback metrics via a custom extensible runtime system developed to support swappable scheduling mechanisms.

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 a NP-Complete problem [**?**]. Instead, focus has been more fruitful when pursuing the optimization of various metrics using some particular objective function [**?**] to tune for particular edge cases. As such, scheduling based on feedback metrics is not new [**?**].

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 extreme 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. But this is only to note that we need to take an opposite approach to our scheduler implementations than classical theory.

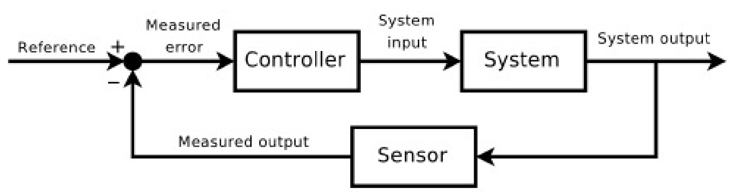

Figure 1: A classical feedback loop representation.

Run-time systems tend to extend to more than just the process-scheduling systems though. 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 these other systems on the placement and order of process evaluation. For example, White *et al.* [**?**] discuss heap size as a metric for process selection as a means of curtailing garbage collection times.

1.1 Classical Run-Time Scheduling

Another distinction must be made as far as the level of foresight the scheduling systems, within this paper, have. 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, in this paper, do not have.

However, as it is a spectrum, there is a single educated guess this subrange of schedulers can make. Namely, 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.

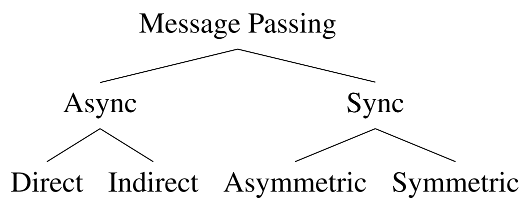

Figure 2: A High-Level Message-Passing Taxonomy

In asynchronous message passing a process can either be provided a rendezvous point or a 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. However, one difference to note is that the rendezvous point allows for the potential of many processes of accessing the message buffer at once.

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 [**?**]. 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 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).

However there is a further classification of synchronous message passing, that of symmetrical vs asymmetrical message passing. Asymmetrical message passing is synonymous with Milner's Calculus of Communicating Systems [**?**] or the standard π-Calculus [**?**], 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 only in that the only operation on the channel is a blocking swap-values with the process on the other end. Note, it has been shown possible to implement symmetrical message passing on asymmetrical message channels [].

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 Cooperativity as a Metric

2 Motivation

It has become a matter of analysis and experimentation to discover which metrics are of value, and to which criterion. There has been work on metrics such as heap size [**?**], process locality [**?**, **?**], and interactivity [**?**] to name a few.

The motivation is twofold. Current feedback scheduling implementations fail to take process-cooperativity into account when applying feedback. A scheduler implementation and metric calculation would aid in future feedback schedulers. However, this hints at other missing metrics that could be useful for adapting to application phases. A closer evaluation of current scheduling techniques on common language primitives will allow for a constructive comparative analysis. This, as far as I'm aware, has not been done at the injected runtime level but rather only at the OS or Application levels with job schedulers.

Thus there would be a desire

3 Proposed Work

I have already begun to work on a compiler built with swappable scheduler's in mind. The language is a simple 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 and visualize the system phases at any given point. I will attempt to do a comparative analysis of several popular scheduling techniques against a new algorithm I will need to design which uses cooperativity. This, in turn, will evaluate the effectiveness of the process and communication channel abstractions utilized in the language.

3.1 The ErLam Language

To compare multiple scheduling systems, we would require a language with minimal overhead but which would easily mimic common process behaviors. That being said, there are three primary qualities of a process which we would like to extract during its evaluation: *a*) the likelihood of communication from this process; *b*) whether the process has children; and *c*) whether two processes have a chance of communicating. The

3.2 Expected 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

```
<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.

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:

April
4/28 - Submission of Proposal with road-map.

May
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] 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.

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

[3] 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.

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

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

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

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

[8] Carl G Ritson, Adam T Sampson, and Frederick RM Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727–740, 2012.

[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.