Runtime Scheduling in Functional Languages

Alexander Dean
Rochester Institute of Technology
ard4138@rit.edu
Week Report's: dstar4138.com/school/is_rs

August 8, 2013

Contents

Abstract

Current trends in computer science point to an increase in concurrency with more multi and many core processors on the horizon. Programming languages are still lagging behind and require their users to implement extensive memory protection schemes and structures. Functional language's certainly have a leg up in this respect as they restrict memory access by default, and use message passing instead. Thus easing further opportunities for improved concurrency. However, the runtime system for functional languages are still holding further advancements back in some ways. Scheduling systems inject quite a bit of overhead. The purpose of this paper is to survey current designs and concepts to provide it's author and reader's with better intuition and background into runtime scheduling.

1 Introduction

As anyone in the computer science field will tell you, one of the bigger trends facing computer programmers today is to start taking more advantage of the available multi and many-core processors. There are a couple ways they can do this with today's more popular languages, but most of these require very carefully planned out access structures and algorithms to protect shared information. Even very experienced programmers can get these things wrong; as I found in my last independent study [5] I looked at maintaining determinism in concurrent memory models.

As language designers and compiler implementers we can side step many of these issues for our users. Functional languages, for example, enforce stricter rules on memory sharing by paradigm. It forces passing of information safely through typed synchronous channels or asynchronous "mail-box" style passing. Either way it restricts access in a method that adapts itself better to a parallel system.

Another improvement to add parallelism possibilities to a language is adapting the virtual machine or the runtime system that the compiler targets/bootstraps. In both cases there is a runtime that waits for new processes to schedule, and in either case it is an area that can be the most influential in terms of efficiency. It can take into account number of cores, their speeds, the cache sizes, number of processes at any given time, and what kinds of communication is going on.

This run-time scheduling system was therefore the target of this independent study. The goal was to gain some background intuition about decisions currently being made, and to explore the implementation details other researchers provide. This paper is broken up into five sections. First, a summary of requirements and criterion for scheduling systems so that we can know in what way they are judged. The next section is the primary portion of the research this quarter which was devoted to design details of

scheduling and how parallelism is supported in functional languages. Then a brief overview of implementation details pulled from various languages and then compared. Next I will go over my findings of the current state of adaptive scheduling techniques using Control Theory. Finally I will talk about my conclusions from the aforementioned research and the future work that can be pursued.

2 Scheduling Requirements

It's important to know how schedulers are being judged so that we know when a good optimization is made versus a poor one. Many of the papers I read compare the efficiency to that of an optimally scheduled program. So first a bit of theory:

Parallel computation can be represented as a directed acyclic graph (a dag), where each node is an instruction and edges represent thread spawns and joins (as seen in the figure below). Someone can then use this representation to prove properties about the group of all possible schedules for a given parallel computation. For example, since the instruction graph is acyclic we can compute a maximum depth of the graph (as if it were a tree) fairly simply. This depth is the longest sequential instruction dependency branch in the computation. This means each instruction on this branch requires the previous instruction to finish execution before it begins. From this we see that the minimum possible execution time is the time it would take to execute to the dag's depth.

To extrapolate on this, a scheduler, if perfectly efficient, can only improve the speed of a program to $max(D, \frac{W}{P})$ where D is the time it takes to run to the depth of the dag (as explained above), W is the amount of work the program must do, and P is the number of processors able to do that work. If the algorithm is embarrassingly parallel, the system will only be as fast as how much work it can do per processing unit (which is the fractional bit, $\frac{W}{P}$). It may be that in some instances, it is better to run some processes in a certain order, or perhaps more often. So this is still an interesting boundary case to compare schedules, even though it has some difficulty translating to the dag model.

With this in mind we can think of the efficiency of a scheduler as it's distance from optimum. However, for a general program, outside the realm of theory it's a bit difficult to determine a minimum depth. Instead we can describe two values T_1 and T_∞, which stand for the minimum serial execution time and the minimum execution time of the computation on infinite processors. With these we can talk about the computational overhead a scheduler injects. But how are these two values better than D and W? Well because instead of talking about instruction counts, we can talk about time.

Notice that $T_1 \geq |D|$ and $T_\infty \geq |\frac{W}{P}|$, in other words the time to serially

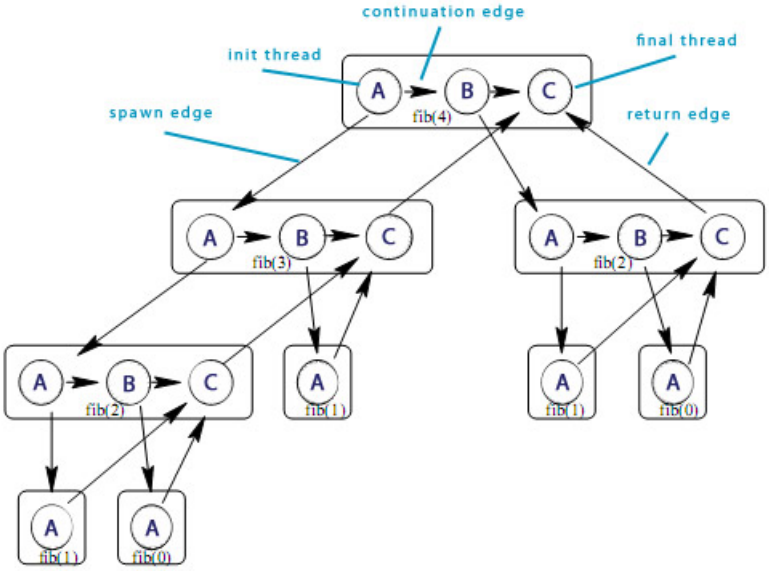

Figure 1: A directed acyclic graph representation of instructions in multiple threads. (Credit: Introduction to Algorithms, CLRS)

execute the computation will not necessarily be the time to compute the max depth, and the time it takes to execute the computation with infinite cores will most likely be more than a perfectly parallelizable computation. There will always be a bit more overhead in both instances. These two values are not necessarily defined, but are use to justify the runtime of the scheduler relative two the computation's complexity as these are algorithmic induced limits rather than hardware or runtime induced limits.

Overhead is therefore an obvious criterion on which to judge a scheduler. There are couple other criterion as well: how well does it distribute work over the available Processors (i.e. how close to T_∞ can we get it?), and can we get the scheduler to perform linearly as we increase P. There are some abstract criterion as well like it's ability to adapt to changing phases of the application.

3 Design Details

There are a substantial number of details that one must go through when designing a scheduler. For example, as we are focusing on multi and many-core systems, a big detail would be how the scheduler balances the set of processes (work) over the set of processing units. There are in fact two

fundamental paradigms for balancing processes locally, Work-Stealing and Work-Sharing.

3.1 Work-Stealing vs Work-Sharing

The basic Work-Stealing algorithm is fairly simplistic, in that it is essentially a depth-first traversal of the computation's dag. Each processor starts out in 'thief' mode, and the global dequeue starts with the main start-up thread. The first processor, will then steal this thread and begin computing. If it hits a spawn call, it will work on the new thread, and demote the current thread to the bottom of its personal dequeue (this repeats until the thread exits or another spawn occurs). Meanwhile the other processors, still in 'thief' mode, will steal (at random) from the top of other processor's dequeues. If the thread it stole is disabled (since it's waiting for a join), it will steal again, otherwise it will proceed with computation. The above repeats until all threads finish their computations.

Work-sharing is the somewhat the opposite. Instead, whenever a spawn call is hit, the processor checks the "utilization level" of the other processing units and then passes the thread to an underutilized processor. However in practice work stealing is preferable. This is because Work-Sharing typically results in much more thread migrations than in stealing. For example, in the event of a high-load program where there a couple very intensive processes. A processor, X, may look as though it is underutilized, but is in fact working on the computationally intensive task. A spawning processor, Y, may see X and spawn several new threads to it, all of which may not be reached before Y needs more work to do. In this event, if the system does not have work-stealing capabilities as well, the system is now hanging on X to finish.

Granted the above algorithms can be mutated in a multitude of ways, and this is partially the point. It is inevitably the tiny modifications and implementation details that can add or remove overhead. Providing multiple dequeues for different types of threads (communication vs computation based), not adhering to the busy-leaf property (don't immediately start computing on the spawned thread), etc.

For the above reasons, I primarily focused on work-stealing algorithms and the mutations that seemed to boost performance. It's worth something to note that the efficiency of work-stealing was shown to execute with time $O(T_1/P + T_\infty)$. That is, the worst a work-stealing scheduler can do is still fairly tight on the expressed optimum.

3.2 Alternate Stealing Algorithms

One of the most intriguing modification to the standard work-stealing scheduler is utilized by the KRoC compiler for the occam-π programming language. Instead of stealing individual processes, steal batches of processes.

This allows you to have a couple of cool things: 1) batch related processes into logical working units (such as in an alternating producer/consumer group of processes), and 2) limit synchronization by making the lists of batches lock free. These coupled with one another allow for a drastic increase in efficiency.

By dynamically grouping processes we can keep processes that communicate with each other in the same batch. What this helps to ensure is that they will be executed together and thus reduce the amount of time it would take to copy that data to another core for the consumer process to use. This also serves another purpose, it reduces the chance that a particular process will have many cache misses during a context switch. If the batch is run a couple times, there is less of a chance for other processes to flood and overwrite the cache. As Vella mentions in [14], "[t]he expense of repopulating the processor's cache with a newly dispatched thread's footprint becomes significant when viewed in relation to the shortened thread dispatch time. To make matters worse, an individual thread is unlikely to accumulate a significant cache footprint by itself: only when threads are considered in groups can a long term cache footprint be identified".

Grouping processes provides another advantage, which boils down to efficient stealing. Not only do you reduce the amount of inter-processor communication required since you don't have to keep asking for another process when yours blocks, but you also make sure that any processes that yours may rely on, comes with it. At least that is the theory, and the results of both [12, 6] conclude that batching certainly helps up to a point. There is a cut off point where batches get too large, or they are too small to really be worth batching.

3.3 Segregation of Computation

Scheduler's maintain a runtime queue that they pull processes off of to run. There are many modifications to how this queuing mechanism works, and CML uses one that discriminates between processes that communicate with one another and those that spend their time chugging away on computation.

CML discriminates between the two by checking if the process voluntarily yields. If it does, then it meant it blocked itself waiting for a communication. Otherwise, it spent its time slot computing and can be labeled, at least for now, as a computation-bound thread. The way CML label's it as computation-bound is to segregate it into a secondary runtime queue. It will promote processes from the secondary queue when it has done too much computation. This method attempts to be more "fair" with regard to the system's load. In other words, a bunch of computation-bound threads wont halt the communication in the system (i.e the progress bar will still be able to update itself even when a lot of computation is getting done).

3.4 Programmer Cues

The above schedulers and mutations don't really consider *order* of the process execution. Except perhaps process batching may be addressed on accident. For example, a particular process is doing some computation while the others are just communicating. The communicators will probably be batched while the one who focuses on computation will be separated to some extent. This would allow it to run, more or less, without competition on another core while the others handle the other batches.

Let's assume you could determine the (time|space)-complexity of parallel portions of code. You could then pre-build the thread queue in memory upon start up to reduce some costs of thread creation. It would also allow you to order the threads ahead of time. However, this assumption quickly leads to the Halting Problem. In other words, given the domain of the input can we can't really guarantee a worst case time limit (even in terms of big-O) as we can't know if it will actually halt. Well it's possible to skirt around the question without "guessing" at time complexity and instead use programmer-provided cues [1], or in this case, cost functions.

The runtime would then run a cost function with the provided value to get an estimation of work. This estimation of work need not be exact, and would only need to guess at the lower bound. It only needs to know if its enough work to parallelize. This means it would be fairly equivalent to giving a big-O representation of the algorithm based on the value, v. So if the algorithm was exponential, then the cost function would return a value roughly exponential to the given value. The total work would then be used to determine whether it was worth spawning another process to run it in parallel, or just running it sequentially beforehand. This would be the task of the Oracle, which would maintain some cut-off point that would be set ahead of time.

Programmer cues would be a great addition to a language for optimization purposes. But if we treat the programmer, as we should, like an opponent (not necessarily malicious but far from perfect), we can't guarantee that they will be accurate. Cues could quickly become a burden or a source of error.

4 Low Level Implementation Choices

There are quite a few design choices that can be made, which may influence design choices. Although, it could be entirely possible that changes like the ones listed above could be fairly contained or plug-n-play with respect to the rest of the scheduler (i.e. like deciding between work-stealing vs work-sharing could have no baring on the implementation of a "process" internally). However, this was not what I found.

I decided to delve into a couple functional languages and their compilers,

to get a flavor for how these decisions permeated through the system. I limited myself to only three to four languages, but I'm going to ignore SML here as I looked at SML/NJ and MLton only briefly and instead focused on a superset language PML (Parallel ML).

The three languages that I will go over are PML, occam-π, and Erlang. Erlang because I had been using it on-and-off for the past several years. occam-π and PML because a few of the papers I read during this quarter were targeting these languages on the KRoC and Manticore compilers (respectively).

With more time, this would be where I would have like to do more digging. Perhaps looking at multiple compiler's for the same language (compare and contrast MLton, and SML/NJ based on their end goals, etc).

4.1 PML: Manticore

Manticore's PML is a superset of SML aiming to be general-purpose and adds the explicit concurrency constructs of CML as well as some implicit constructs like parallel tuples and arrays. By having both types of parallelism it adds another layer of complexity as implicit constructs can have a hand-wavy requirement for parallelism whereas the explicit ones will need something much firmer to optimize for responsiveness and throughput.

On startup, the Manticore runtime will initialize P PThreads, where P is the number of logical processors (if not overridden), that will be our virtual processor (vproc) processes. The lead vproc will set up the main schedulers for all of the vprocs, while the others go into an idle state (which means to wait for a signal from their landing pads).

Manticore uses a scheduler hierarchy insofar as the primary scheduler can schedule "threads" (groups of fibers) which have their own schedulers. The primary scheduler is a lazy-tree splitting, work-stealing scheduler. This means for every parallel tuple it will split and compute half while postponing the other halve by throwing it to the end of their primary queue. Thieves from neighboring vprocs can steal that other half and then merge the result for the thunk that needed it once it's done. This hierarchy lets implicit constructs to spawn work-groups that have their own scheduler.

Although, on a side note, I was confused at first because there was a reference to two runtime queues in the vproc headers. I thought that Manticore might have added the CML dual-queue scheduler on-top of their work stealing schedulers already. But it turns out that each vproc has a single runtime queue that consists of three linked lists. The local vproc can dequeue from primary, enqueue to secondary, and can enqueue on a remote's ternary list (they call this the landing pad). This is how they transfer thief fibers between two vprocs. The landing pad has two operations, push a single message onto a remote, and pop all local messages.

My next question was how do the thieves work then if there are multiple

lists to steal from? After being transferred to a victim's landing pad, the vproc will transfer to primary queue (since it's an atomic appended to the front of the queue) and run it. The thief when run, will reach down into the vproc's "resume" thread and try to copy those out, if that is empty it will go after the primary queue. If both are empty then it will wait and try again later (unless canceled).

4.2 occam-π: KRoC

The occam sequence of languages focus on general purpose explicit concurrency, based on C. A. R. Hoare's Communicating Sequential Processes (CSP) and Π-Calculus.

occam-π batches processes to take advantage of cache-affinity. It then maintains a double-ended runtime queue to run the batches from. The queue is lock-free and the schedulers from other cores can steal work from the end while the local scheduler can run from the front.

Each job is a bare minimum closure (a function pointer, a pointer to the argument, a pointer to the return address) with a small pointer to the pthread that's running it. This differs from Manticore Fibers, the fibers are fairly equivalent to a single-job batch, insofar as they wrap the closure with some state and the thread's id. Both languages use Channels for communication, but I think occam-π might be a closer CSP implementation. I say this as occam's channels are implemented as primitive data types in the interpreter, and Manticore implements them in the PML language itself. Granted the PML also contains low-level 'primcode' (Manticore specialized code that lets you access and control thread-level data).

KRoC's batch structure actually isn't much more complicated, but there are actually two types apparently. Those with a variable size list of jobs to complete, and those with a single job (perhaps for critical jobs). The batches have a bit of state for migration purposes (dirty/clean markers for whether its been copied already, etc.), a core affinity which can be set to disable migration on a per-batch basis, and a priority.

4.3 Erlang: Ericsson's Compiler

Erlang is a soft-real-time, priority based scheduler that also worries about distributed computing. Erlang's focus is on fault-tolerance and is willing to give up some cycles to guarantee distribution and balance.

Erlang uses work-stealing/load-balancing, adaptive time slices, and priority queues, to schedule processes per core. Each core will, after some time, check if the load is balanced on the system, if it's not it can migrate (steal) processes, and/or kill schedulers with low load. These load-balancers also take into account process priority and they type of job on the system. By "type of job" I mean figuring out whether its a pure function or a port job.

A port job is one that is communicating with outside of the Virtual Machine (and can have long latency, which is something Erlang tries desperately to avoid).

All ports and processes seem to be stored (pointer wise) in a global ptab (process table), which can be used to access all other processes fairly quickly. I'm not actually sure what the function of this is, my speculation is for process migration or fast message passing.

There is a difference in how the virtual machine handles load-balancing versus work-stealing. In the Erlang world (as opposed to the previous two languages), processes can have priorities. So Erlang's VM will load-balance the high priority processes equally amongst the cores so that they will get preferential treatment in each sub-scheduler. Although the load-balancing function (*check_balance*) will also take into account the number of lower-level processes too, so that everything evens out. So its not just making sure priority sums are equal for each core. Erlang's work-stealing (*try_steal_task*) will also take into account process priority, so that the load balancing takes less time.

5 Adaptive Scheduling

Schedulers attempt to make decisions that seem fair when dealing with average programs: segregating computation so that the system doesn't jitter, batching processes to reduce communication time, priorities to influence order of process execution, etc. However, these modifications don't really understand the current state of the system. If the system could listen to how the system is running, and then learn from it, it could adapt and run the program better (whether that means faster, with less memory, etc.).

Engineers have been building systems which adapt to changes for a while now. They use Control Theory to process the system and feed that information back and adapt to it.

5.1 Control Theory

But first, what is Control-Theory (CT)? A popular example of Control Theory in action is the concept of cruise control in your car. The desired goal is to maintain a particular speed, but there are a couple dynamic inputs that alter the behavior of the function (incline, weight of car, current speed, etc), and then there are the control points (the throttle, breaks) which the function can manipulate. The control system would take the incoming inputs and attempt to adjust the control points to achieve the goal. It becomes increasingly more interesting when there are more than one control point; This is when you start to have to worry about the efficiency of each point.

I was given a link to [15] which discusses CT in terms of Heap size and Garbage collection. The paper explains their intentions to adjust heap size

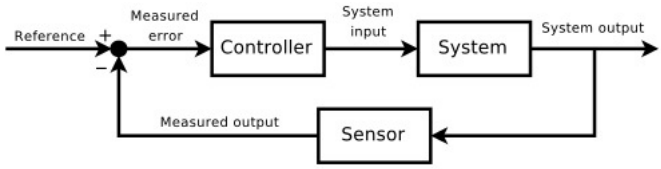

Figure 2: An example generic feedback loop from Control Theory. (Credit: Wikimedia)

as a function of garbage collection overhead. They would like to be able to have tighter bounds than past complicated heuristics. If the heap size gets too large, then you start to feel the effects of page-faults. If it gets too small, then you feel the effects of more frequent garbage collection. However, as they explained, it's common for programs to go through phases of intense and then minimal memory and disk usage. As such, it would be optimal for the new algorithm to smooth over tiny spikes in usage to avoid over-resizing. The feedback loop that the paper suggests can better be visualized through figure 2.

This is fairly standard, but the portions are interchangeable; particularly the Controller. The one that they discussed in the paper was the Proportional-Integral-Derivative (PID) Controller. Each of the three pieces of the controller (P, I, D) help to correct the error over a particular time-slice in a different way. The proportional portion quickly addresses the direction of the error (overshot or undershot), the Integral portion keeps in mind the duration of the previous error (how long it has been over/under shot), and the derivative portion tends to settle the osculation around the reference signal.

These controllers still need some tuning to work well for the system on which its running. So for run-times, it will either need to be bootstrapped during compilation or adjusted before the runtime or scheduling takes place. A note which [15] made was that for virtual software systems, we can disregard the suggestions for tuning points that are made for mechanical systems. Software will favor tight oscillations which, if it happened in a normal mechanical system, could cause harm to the machine as it strained to reach that point. This, I thought, was an extremely important distinction when adapting CT for use in runtime-systems.

5.2 Program Interactivity

However, before being able to merge CT with schedulers, we need to know what about the system we would be controlling. There may be more possibilities, but the one that interested me was quantifying how "capable" the system is at handling the interactivity in the system (communication vs

computation).

Interactivity can be seen as an adaptive balancing act between having enough time devoted to computation but still allowing for enough message passing time so that the system doesn't get backed up or hang. A good example is that of a GUI progress bar indicating the progress of some intensive computation. Obviously the computation is the primary goal of the program, but the progress bar still needs to function correctly or else it is useless. But after a certain point, sending updates to the progress bar doesn't make sense. There is only so much "progress" that can be seen by a user. In other words, an update every millisecond might not be helpful as the screen doesn't even update that quickly.

So there is an obvious ceiling for communication, but a floor for computation (if it goes too low, nothing will get done). The balancing act comes at trying to find an oscillation point for a particular system. This oscillation point would represent a happy medium between communication and computation. However, the question first becomes, how do you measure or quantify the idea of "interactivity".

I personally think of it as a ratio of communication to computation. I will define interactivity of a system (for the time being and for the sake of argument) to mean the ratio of blocked (B) processes to the number of interrupted processes (I) and recently unblocked (U) during a given time slice (phase): $\frac{B+1}{I+U}$.

In other words, if a system is able to keep up with the production of messages (assuming a thread blocks on a channel when waiting to send a message and unblocks when successfully sent/received) the interactivity of a system is low, otherwise, the number of blocked processes is over saturating the number computation-bound processes. However, a downside to using this measurement as the value for interactivity, is that it's not immediately obvious what an optimum system would look like. This isn't able to give us a deviation from where we should be in order to set a direction for tweaking (e.g. remember the delta taken from our car's cruise control).

However, looking at the ratio, it has some properties we would need in a better interactivity measurement:

- Unblocked processes are offsetting the number of blocked. In a communication heavy application (ping-pong) we would want to reduce the ratio so that either we have less blocked threads, or more unblocking.
- Communication-bound threads are offset by Computation-bound. This will need to be weighed. A sum as the denominator is most likely incorrect and it will need to be more complex.
- As more processes become blocked, it would make sense for the level of interactivity to increase as well.

Figure 3: Four processes competing for time, the last is a computational thread. (Credit: John Reppy)

- As more processes are interrupted (i.e. computation-intensive), it would make sense for the level of interactivity to decrease in some way (but not necessarily inversely proportional).

This measurement of interactivity is actually not taking into account the extent of the program's execution though. It is currently merely looking at single time-slices. This is not advantageous as each time slice may differ dramatically from one to the next. Interactivity of the system will need to be able to average itself over several time slices before a feedback system modifies the system. It may be a good idea to think of this as the osculation of the system. How widely it differs between each time slice could be our deltas that we keep track of (i.e. Max and Min interactivity levels instead of a single measurement).

5.3 Feedback Scheduling

Let's assume the measurement(s) of interactivity above is/are perfect and we can determine a good idea of the run-time's efficiency from our findings. The next question would be when to update the system. The above only really works if there is enough of a time-slice on which to gauge the control value, so I mentioned the word "phase". How long should the phase last? Once around the runtime queue? Not really, since spawning and exiting will really throw off the differences from last time. In fact this kind of an issue will crop up if it's done at a fixed time-slice away. This is why the phase size will need to be dynamic and based on self-measurement.

However, assuming we reach the end of a phase, and we notice that interactivity is either too high or too low. We would still need to modify the system so that the interactivity could move into a better oscillation.

Here are a couple of ideas for changes we could make to the system during runtime as a reaction to measuring interactivity:

- Use the dual queue from CML, and hand-throw some computational threads onto the primary queue.

- Adjust the time-slice signal for each process: Increasing/decreasing it will be of more benefit/harm for computational-intensive tasks, and won't have an effect on communication-based tasks.

- Unblocking a process has the effect of putting it on a different queue (rather than the primary) to be promoted later. This has the effect of preferential treatment to communication rather than computation (i.e. the unblocked process will most likely chug on incoming data, work on making more data to send, or exit).

- "Batch" processes to repeat (rather than strictly round-robin). There are a number of ways to do this, I'm not sure the best way. However, the idea of grouping and then repeatedly running through that set (ignoring the others) would give an advantage to whoever you grouped separately. (This is perhaps an abstraction of the first point...)

There are many other possibilities, and these can change in effectiveness based on how the value of interactivity is defined.

6 Conclusions and Future Work

Run-time schedulers are going to be of increasing importance due to more processing units on personal computing devices. There is obviously a lot that can be done in the realm of adaptive schedulers; but it would also be beneficial to think further about defining a program's interactivity ratio. The concept of measuring a systems oscillation so as to tighten it might make for a good definition of a feedback scheduler. I intend to pursue this current line of research during my graduate year, and hope to provide some more conclusion on the matter.

References

[1] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Oracle scheduling: controlling granularity in implicitly parallel languages. In *ACM SIGPLAN Notices*, volume 46, pages 499–518. ACM, 2011.

[2] Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. Lazy tree splitting. In *ACM Sigplan Notices*, volume 45, pages 93–104. ACM, 2010.

[3] Guy E Blelloch, Phillip B Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 1–12. ACM, 1995.

[4] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[5] Alexander Dean. Memory models and determinism for concurrent/parallel programming languages. *RIT Independent Study*, 2011.

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

[7] 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.

[8] Derek L Eager, John Zahorjan, and Edward D Lazowska. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on*, 38(3):408–423, 1989.

[9] Jeff Edmonds. Scheduling in the dark. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 179–188. ACM, 1999.

[10] Steven P Reiss. Dynamic detection and visualization of software phases. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–6. ACM, 2005.

[11] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[12] 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.

[13] Alexandros Tzannes, George C Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. *ACM Sigplan Notices*, 45(5):179–190, 2010.

[14] Kevin Vella. A summary of research in system software and concurrency at the university of malta: Multithreading. In *Proceedings of CSAW'03*, page 116, 2003.

[15] David R White, Jeremy Singer, Jonathan M Aitken, and David Matthews. Automated heap sizing in the poly/ml runtime (position paper).