What are Cycle-Stealing Systems Good For? A Detailed Performance Model Case Study

Wayne Kelly and Jiro Sumitomo
Queensland University of Technology, Australia
{w.kelly, j2.sumitomo}@qut.edu.au

Abstract

The idea of stealing cycles has been hyped for some years, boasting unlimited potential by tapping the computational power of millions of under utilized PCs connected to the Internet. Despite a few spectacular success stories (eg SETI@HOME), cycle-stealing is today not a widely used technology. We believe two principal impediments need to be overcome. The first is ease of development and use. Most of the problems faced in developing cycle stealing applications are not specific to those applications, so generic cycle stealing frameworks such as our G2 framework can play a vital role in this regard. The second is uncertainty. Potential developers don't know whether if they went to the effort of developing a parallel application for a cycle stealing environment, it would pay off, i.e. whether they would get a reasonable speedup. To minimize this risk, we propose the development and use of detailed performance models.

1. Introduction

Applications such as SETI@HOME have demonstrated that cycles can be effectively harvested from large collections of PCs connected via the Internet. Such applications are ideally suited to this environment as the tasks involved are completely independent, the sizes of the inputs and outputs to these tasks are relatively small, yet each task typically requires several hours of processing. What about other classes of application - applications that may not be embarrassingly parallel, which consume or produce relatively more data or which require less processing time per task? Even dedicated supercomputers with high-performance interconnects have their limits when it comes to achieving speedups if the computation to communication ratio is not sufficiently high. Cycle stealing systems which operate over the Internet have substantially higher overheads, so what is the limit of their applicability? Clearly there are some applications that work effectively in this environment and others that will not. What is crucial is a means of distinguishing without having to implement and test empirically.

In this paper we present a detailed performance model for our Internet cycle stealing framework, G2 [1, 2], and show that it can be used to prescribe the properties that applications need to exhibit in order to perform well in this environment.

In Section 2 we outline the architecture of G2. Despite its seeming simplicity, we found it surprisingly difficult to accurately model the performance of G2 applications. We started by abstracting aspects of parallel execution such as average job execution time and job parameter sizes and modelling the various points at which jobs and results are queued. It soon became apparent that we needed to start by developing a discrete event simulator for our framework so that we could be sure that we were not overly simplifying crucial aspects of the actual system. What started out as a simple discrete event simulator soon turned into a very complex and detailed simulator. This simulator is described in Section 3. This gave us a good understanding of how the actual system behaved at a more abstract level, but ultimately we desired an analytical model. As well as allowing predictions to be made more rapidly, analytical models also allow us via investigation of the equations themselves to discover new rules and relationships.

What we are modelling is clearly a queuing system; however, we had to further simplify some aspects of our discrete event simulator in order to allow state of the art queuing theory techniques to be applied. To determine the extent to which such further simplification affected the accuracy of the predictions, we constructed a simpler simulator that reflected only those aspects of our original simulator that could be modelled using queuing theory techniques. This simplified simulator is outlined in Section 4 and a comparison of results obtained by the two simulators is given in Section 7. In Section 5 we present our analytical performance model based on Mean Value Analysis (MVA) [3] for multi-class closed queuing networks. In Section 7 we compare empirical performance results to those predicted by our complex and simple simulators and our analytical model, with conclusions in Section 8.

2. G2 Architecture

The G2 Framework is designed to make programming cycle stealing applications as simple as possible. It does so by hiding application programmers from much of the underlying physical topology, job management and communication that takes place. To model the performance of such a system we need, however, to delve into these internal implementation details. This section gives a brief overview of these details.

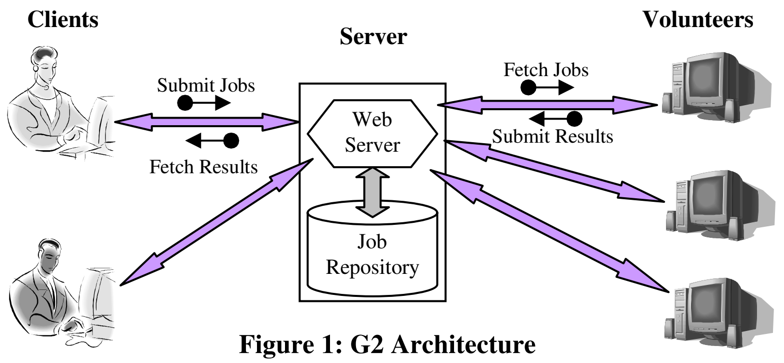

Figure 1: G2 Architecture

A G2 system comprises a *Server* that acts as clearing house for *jobs* waiting to be executed and *results* waiting to be collected. *Client* applications use automatically generated application specific proxy classes to submit *jobs* to the server via a web service interface. A job consists of executing a programmer specified method with a given set of parameters and can be executed completely independently from all other jobs.

Volunteers use a web browser to launch a generic volunteer host application that fetches jobs from the server, executes them and submits the results back to the server (again via a web service interface).

The volunteer host application maintains its own local queues of jobs waiting to be executed and results waiting to be submitted. It does this so as to pipeline the execution of jobs - overlapping the execution of one job with the fetching of the next job and the submitting of the previous result**.** If a volunteer attempts to fetch a job and there are none currently on the server then it sleeps for a while before reattempting.

When results arrive back at the server they cannot be simply forwarded to the appropriate client as we assume conservatively that *not* all clients will be accessible from the server due to firewalls etc. The only machine that we require to be universally accessible to the other machines making up the cycle stealing network is the server itself. This means that clients and volunteers can be located anywhere on the Internet provided they can contact the server. This restriction means that the clients need to "pull" job result off the server by means of a FetchResults web method. To minimize network traffic each client only ever has outstanding one FetchResults request. Such a request returns all results currently waiting on the server for that client. If there are no results currently present then the request will wait (on the server side) until at least one result becomes available.

3. A Discrete Event Simulator

In producing our discrete event simulator for G2 we ended up producing a generic distributed computing discrete event simulation framework. Only a small part of the simulator deals specifically with the behaviour of the G2 framework. Our simulation framework models a collection of workstations connected by network links along which IP packets flow in a FIFO fashion. Each workstation possesses one or more CPUs which execute a collection of local threads in a round robin fashion. The behaviour of each thread is controlled by a finite state machine that expresses an abstraction of the application's logic. Each thread state determines a task to be performed. Some tasks are purely CPU bound, requiring a certain number of cycles to complete, others are entirely memory or disk bound which means they have no need for the CPU during that period. Other tasks deal with sending and receiving messages via a network card.

The workstations may possess network interface cards which maintain physical I/O buffers and the ability to send an interrupt to the workstation's CPU when a new packet arrives. Such workstations may implement a TCP/IP stack which allows socket connections to be established and maintained with other machines. The TCP protocol uses special connect, accept and acknowledgment messages in order to control data flow such that buffers do not overflow. The key outcome of this mechanism from our point of view is that receivers are able to exert backward pressure on senders, preventing them from sending additional packets until the receivers have caught up to a certain extent. Figure 1 shows the components that make up our G2 simulator.

The G2 specific parts of the system are three subclasses of workstation for the clients, server and volunteers respectively. The client workstation creates a job submission thread and a fetch results thread. The volunteer workstation creates threads for fetching, executing and submitting. The server workstation simulates the behaviour of the IIS web server and the G2 web service implementation via a Listen thread which listens for TCP connection requests on port 80. A fixed pool of recyclable worker threads is used to process the incoming HTTP requests on the sockets established by the listen thread. The worker threads must first de-serialise the incoming web service requests before being able to extract the name of the web service method and process it appropriately according to the abstracted semantics of that method.

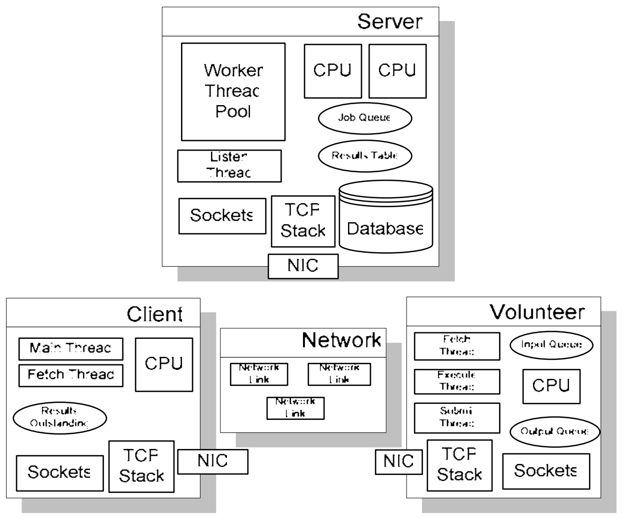

Figure 1: G2 Simulator Components

Most of these web methods involve a "database" task which consists of a series of CPU phases and memory/disk phases. This detailed modelling was necessary in order to accurately predict the behaviour of the server when large numbers of web requests are being processed concurrently as the CPU bound phase of one web request can be overlapped with the memory/disk bound phase of another.

The transitions which the state machines make is determined in part by G2 specific state maintained by the client, server and volunteer components, such as the number of jobs currently waiting to be processed on the server or the number of results on a volunteer waiting to be submitted. Figure 2 shows a summary of the state machine that controls the G2 server worker threads.

The length of the ticks that control the flow of the discrete events can be easily adjusted so as to trade-off the accuracy of the simulation with the length of time required to run the simulator. The number of ticks attributed to each task for the G2 threads was determined by extensive experimentation and various methods of profiling designed to isolate the constituent costs. The experiments were repeated for various parameter values and equations fitted to the empirical data. In most cases, the equations were linear in one or more of the input parameters.

4. Towards an Analytical Model

In order to allow analytic queuing theory techniques to be applied we had to simplify some of the aspects that we modelled in our original simulator. We made these simplifications first to our simulator in order measure the exact effect of doing so. If we had attempted to go directly to an analytical solution we would not have known if the (presumably) different results were due to these simplifications or due to a flaw in the process of translating our *dynamic* simulator into a *static* analytical model.

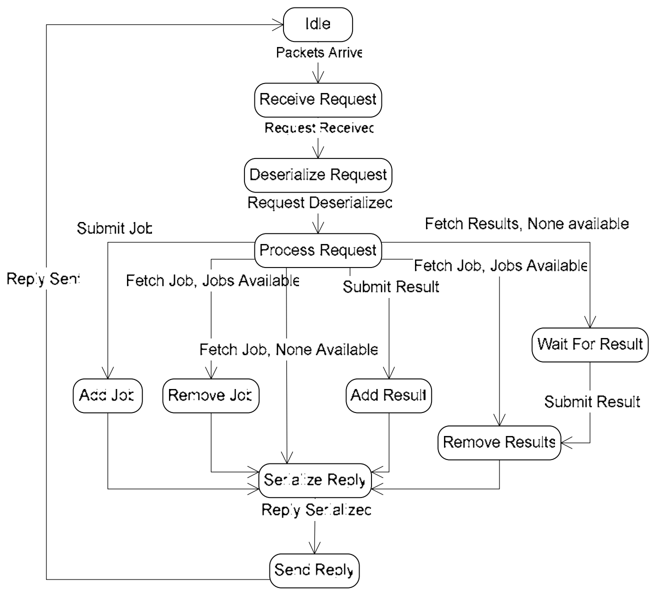

Figure 2: Worker Thread State Diagram

The following simplifications were made:

- We eliminated the server multithreading.
- The CPU and memory/disk phases of the server were amalgamated into a single service time.
- The TCP/IP elements of the communication were dramatically simplified by incorporating all communication costs into the request generation and the response processing times on the clients/volunteers.

5. An Analytical Performance Model

We model the server as a multi-class FCFS queuing centre with class dependent service times. The classes are Submit Job (SJ), Fetch Job (FJ), Submit Result (SR) and Fetch Results (FR). We are able to model the system as closed since the number of requests for each class remains constant for a given number of clients and volunteers.

Mean Value Analysis [3] tells us that the average response time (R) of a request is equal to the average time the request waits in the queue plus the average service time (S) spent actually servicing the request. The average time waiting in the queue is approximately equal to the average number of requests (Q) in the queue multiplied by their respective average service times:

$$R_c = S_c(1 + Q'_c) + \sum_{d \neq c} S_d Q_d$$

where the subscripts represent the various classes.

Q'_c represents the average number of requests that would be in the queue for a closed system containing one fewer total requests (N_c) of class *c*. In general, this would require us to recursively solve the entire problem with this reduced number of requests for this class. Doing so would be extremely costly in both time and space, but in our peculiar scenario it is not even valid as the numbers of requests in each class are coupled. It is not possible for example to consider a G2 system in which there is 1 FetchResult request, but 0 SubmitJob requests as without jobs being submitted, no results will ever be fetched. Fortunately there is a well known approximate MVA algorithm [4] for multi-class queuing networks that solves this problem. The approximate MVA algorithm approximates Q'_c by:

$$Q_c\left(\frac{N_c-1}{N_c}\right).$$

So, for example, we have:

$$R_{SJ} = S_{SJ}\left(1 + Q_{SJ}\frac{N_{SJ}-1}{N_{SJ}}\right) + S_{FJ}Q_{FJ} + S_{SR}Q_{SR} + S_{FR}Q_{FR}$$

N_{SJ} and N_{FR} will be equal to the number of clients while N_{FJ} and N_{SR} will be equal to the number of volunteers.

Fetch results (R_{FR}) is a little different as the FetchResults web service is implemented asynchronously. If there are no results for that client currently waiting on the server, the server will wait until such a result arrives before processing the FetchResult request. This adds an additional wait time (W) to the response time $R_{FR.}$ The average wait time W is given by the probability WP that no results are available when the FetchResults request is first serviced, multiplied by the average duration WD we will have to wait for a new SubmitResult request to arrive.

If we assume the arrival of the requests follows a Poison stream we have:

$$WP = e^{-\left(\frac{X_{SR}}{X_{FJ}}\right)} \text{ and } WD = \frac{1}{X_{SR}}$$

where X_c is the average throughput of class *c*.

The average queue length is given by the standard multi-class MVA equation:

$$Q_c = X_c R_c$$

The standard MVA equation for throughput (X) is:

$$X_c = \frac{N_c}{R_c + Z_c}$$

where Z_c is a lower bound on the time that the requests will spend away from the server. This includes the time required by the clients/volunteers to generate each new request, the time to transport the request over the Internet to the server, the time to transport the server's reply back to the client/volunteer and the time required by the client/volunteer to process the reply before the next request of that class can be generated.

However, we have so far ignored the fact that the four classes of request are coupled to one another. The actual throughputs (X_c) may therefore be lower than the above formula predicts due to these other influences.

A FetchJob request will return a job if one is available on the server, otherwise it will return a null value. The percentage (*r*) of FetchJob requests that will *succeed* is determined by the rate at which jobs are being submitted and the rate at which jobs are being fetched:

$$r = \min\left(\frac{X_{SJ}}{X_{FJ}}, 100\ \%\right)$$

This FetchJob success rate *r* affects the average service time S_{FJ} on the server as well as the behaviour of the volunteer receiving the FetchJob replies. If a volunteer receives a null value it will sleep for some period before attempting to request another job. If, however, a job is successfully received, the volunteer will execute it which will take some other period of time. We account for this difference by having two different Z values, one for the case where the volunteer sleeps (Z_{sleep}) and one for the case where the volunteer executes the job ($Z_{execute}$).

Whilst the volunteer has jobs to execute, the volunteer's "fetch" thread will be kept busy trying to fetch new jobs from the server to replace the ones that the volunteer's "execute" thread is removing from the local job queue. The minimum time required to generate, transport and process the results of these FetchJob requests is represented by $Z_{FJ.}$

Whilst the volunteer's "execute" thread is executing jobs and placing their results in the local result queue, the volunteer's submit thread is kept busy trying to submit those results to the server. The minimum time required to generate, transport and process the results of these SubmitResult requests is represented by Z_{SR}.

If the output queue becomes full, then job execution pauses until the submit thread can catch up. Any of these three volunteer threads can therefore become the bottleneck that limits the FetchJob throughput. So we have:

$$X_{FJ} = \frac{N_{FJ}}{(Z_{sleep} + R_{FJ})(1-r) + \max(Z_{FJ} + R_{FJ}, Z_{execute}, Z_{SR} + R_{SR})r}$$

The rate at which we submit results cannot be higher than the rate at which we successfully fetch jobs, so we have:

$$X_{SR} = \min\left(\frac{N_{SR}}{Z_{SR} + R_{SR}}, rX_{FJ}\right)$$

The rate at which we fetch results cannot be higher than the rate at which we submit results since FetchResult requests wait on the server side until at least one result becomes available, so we have:

$$X_{FR} = \min\left(\frac{N_{FR}}{Z_{FR} + R_{FR}}, X_{SR}\right)$$

If we are interested in an optimal steady state then there is no point in submitting jobs at a rate higher than the rate at which we can fetch, so we have:

$$X_{SJ} = \min\left(\frac{N_{SJ}}{Z_{SJ} + R_{SJ}}, E \cdot X_{FR}\right)$$

where E is the average number of results returned per FetchResults request.

$$E = \frac{X_{SR}}{X_{FR}}$$

E also affects the average FetchResults service time (S_{FR}) on the server and the time Z_{FR} required by the client to process that number of results. So, S_{FR} and Z_{FR} are a function of E, while S_{FJ} and X_{FJ} are a function of r.

These and other cyclic dependences in our analytical equations mean that we cannot solve them directly. We instead solve them in an iterative manner.

6. Volatile Volunteers

One of the biggest challenges with cycle stealing is that volunteered computers may (by fault or intention) leave the network without warning at any time. The G2 framework transparently handles this problem quite simply by retaining a record of each job on the server until a result has been submitted. If the volunteer that had been assigned that job leaves, the server can simply reassign it to some other volunteer.

This behaviour was easy to incorporate into the simulators but we are still working on incorporating it into our analytical model. This behaviour is actually quite important, as without the assumption that volunteers may leave without warning there is nothing to dissuade us from performing as much computation in each job as possible. Choosing the appropriate job execution time becomes a balancing act between keeping the computation to communication ratio high while not making the execution times so long that large amounts of computation are lost when a volunteer departs.

7. Experimental Validation

Figures 3 through 5 compare the performance of our actual cycle stealing system with the performance predicted by our complex simulator, simple simulator and analytical model as the data size, execution time and numbers of volunteers are varied respectively.

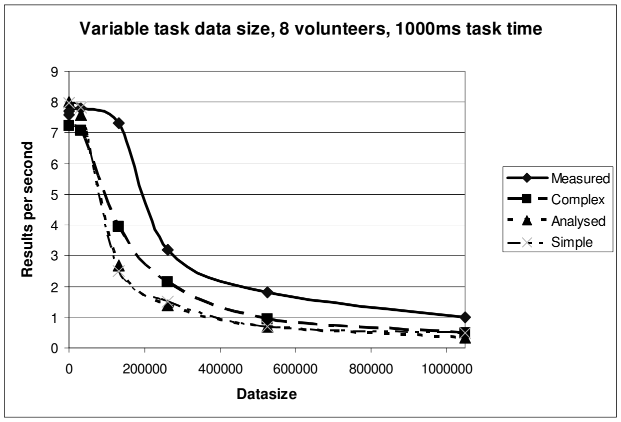

Figure 3: Throughput vs Data Size (i/o)

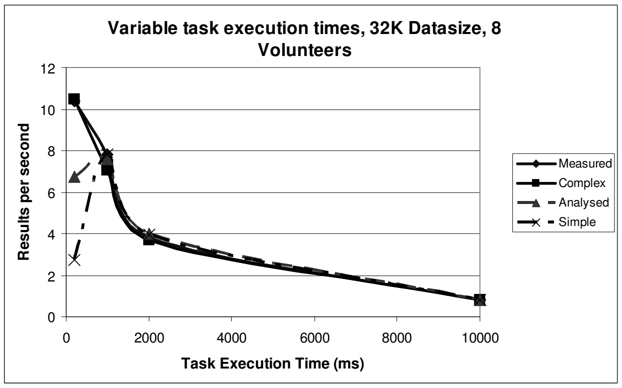

Figure 4:Throughput vs Job Execution Time

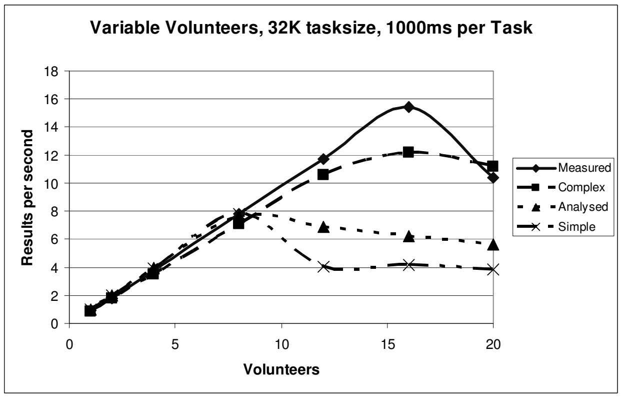

Figure 5: Throughput vs Nr. of Volunteers

8. Conclusions

Our complex simulator reasonably accurately predicts the performance of the actual system. The simple simulator and analytical model predict well in most cases but underestimate performance when the server is heavily loaded. This is due to the lack of concurrency modelled in those systems. We are currently investigating multi-server models that may help address these inaccuracies.

References

[1] Wayne Kelly, Paul Roe and Jiro Sumitomo, *G2: A Grid Middleware for Cycle Donation using .NET*, International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, June 2002.

[2] Wayne Kelly and Paul Roe, *A Framework for Automatic and Secure Cycle Stealing*, International Conference on High Performance Computing and Grid in Asia Pacific Region, July 2004.

[3] M. Reiser and S.S.Lavenberg, *Mean-Value Analysis of Closed Multichain Queuing Networks*, Journal of the ACM, 27(2):313-322, April 1980.

[4] P.J. Schweitzer, *Approximate analysis of multiclass closed networks of queues*, Proceedings of the International Conference on Stochastic Control and Optimization, 25-29, Amsterdam, Netherlands, 1979.