The Performance of Self-Scheduled Concurrent Systems

Chris Tofts
Publishing Systems and Solutions Laboratory
HP Laboratories Bristol
HPL-2001-121
May 16th , 2001*

E-mail: chris_tofts@hp.com

scheduling, Markov Chain, concurrency, embedded systems, transients

The solution of complex real time tasks is often achieved by pipe-lining. The task is broken up into several 'smaller' blocks which may share both processing and memory resources. When these block execution times are variable, and consequently difficult to schedule, a standard solution is to execute computational elements concurrently and allow them to 'find' a good schedule as a consequence of their interactions. In this paper we examine how long this search may take, and demonstrate that even in relatively ideal circumstances this time can be comparable with the total task duration. We then demonstrate a simple technique that reduces the effects of this problem without requiring the imposition of a global schedule.

* Internal Accession Date Only   Approved for External Publication
© Copyright Hewlett-Packard Company 2001

The Performance of Self-Scheduled Concurrent Systems.

C. Tofts*
HP Laboratories Bristol,
Filton Road,
Stoke Gifford,
Bristol,
BS34 8QZ
chris_tofts@hp.com

May 11, 2001

Abstract

The solution of complex real time tasks is often achieved by pipe-lining. The task is broken up into several 'smaller' blocks which may share both processing and memory resources. When these block execution times are variable, and consequently difficult to schedule, a standard solution is to execute computational elements concurrently and allow them to 'find' a good schedule as a consequence of their interactions. In this paper we examine how long this search may take, and demonstrate that even in relatively ideal circumstances this time can be comparable with the total task duration. We then demonstrate a simple technique that reduces the effects of this problem without requiring the imposition of a global schedule.

1 Introduction

In order to achieve a complex computation the task is usually broken up into several smaller stages, both for ease of comprehension and implementation. One advantage of this approach is that idle time, with respected to limited resources (such as memory) within the tasks can be exploited to improve efficeincy. This is achieved by permitting the tasks to execute in (often pseudo) parallel. The parallelism requires that the limited resource time is allocated to each of the tasks in order to maximise the throughput of the system. Equally a solution method widely adopted to cope with scheduling large complex interacting systems, over shared resources, is to consider them as concurrent tasks, and allow the concurrency [1] to 'sort it out'. This approach is often adopted when the job load to individual tasks is highly unpredictable and variable. Whilst over the long term the system may well achieve a good schedule, in the short term there may well be problems as the system 'hunts' for a good allocation of the resource time. We are interested in the problem of how long such a schedule will take to establish given it is feasible (e.g. total resource demands do not exceed 100% utilisation) and the consequences of variation in the performance of the scheduled sub-systems. In particular we are interested in the area of jobs which are long, maybe of the order of hours, but not persistent, they do eventually terminate. So system latency, in this case the time to achieve the good schedule, could well be a dominant factor in its performance

*This work is supported by A Royal Society Industrial Fellowship and started whilst the author was on leave from the Department of Computer Science, University of Leeds.

Each of the tasks needed to complete a job can be considered to behave in the following fashion:

1. get some data to work on taking a period of time;
2. compute on the data taking some time;
3. write the result back, taking some time;
4. start again.

Each of the times above can be considered to be dependent on the actual problem being worked on, but for a simple model we can consider them to be fixed functions of the computation task. In a complete system the tasks may be dependent for work on the preceding stage. In other words if a stage fails to deliver its work then subsequent stages can be starved.

2 Model without Dependence.

In general we can consider our functional units to be described by the following set of processes (TCCS style)[10] based on Milner's CCS[7, 9]:

$$\begin{aligned} Task_i &\stackrel{def}{=} \overline{getM}.(input_time_i).\overline{putM}.Task_iW \\ Task_iW &\stackrel{def}{=} (work_time_i).Task_iO \\ Task_io &\stackrel{def}{=} \overline{getM}.(output_time_i).\overline{putM}.Task_i \end{aligned}$$

We consider the work time of a stage to be its required compute time, plus any idle time it is willing to spend to keep the system tuned and running smoothly.

The complete system is described as the following:

$$\begin{aligned} Sem &\stackrel{def}{=} getM.putM.Sem \\ Sys &\stackrel{def}{=} (Task_0|Task_1|...|Task_n|Sem)\backslash\{getM, putM\} \end{aligned}$$

We would expect the above to be schedulable that is we can arrange the processes so there are no delays, if $\forall j work_time_j \geq \sum_i (input_time_i + output_time_i)$, however whilst this system can settle down to this state it will take some time to do so. Possibly a very long time!

Of further interest is the question as to what happens when we extend the times to come from some probability distributions rather than some fixed values. We might expect the system to run stably if $\forall j E(work_time_j) \geq \sum_{i \neq j}(E(input_time_i) + E(output_time_i))$,where $E(d)$ is the expected time of a probability distribution. However, this tells us nothing about the intereference on route to stability or its persistence as a result of perturning the access schedule.

2.1 Model Implementation

The simplies instatiation of the above model is one where we have N tasks, and each load/save can be completed in unit time. Then we need a window of $2(N-1)$ between the loads and save to allow other players to complete. This can be modelled as follows (in WSCCS) [12, 14, 15], based on Milner's SCCS [8]:

```
*Simple scheduable model 3 tasks
*the work description for the simple task
bs GPE 1@1.gs^-1#ps^-1#memPE:GPE2 + 1.stallPE:GPE
bs GPE2 1.t:GPEa
bs GPEa 1.t:GPEb
```

```
bs GPEb 1.t:GPEc
bs GPEc 1.t:GPE5
bs GPE5 1@1.gs^-1#ps^-1#memPE:GPE + 1.stallPE:GPE5

*next task
bs GDL 1@1.gs^-1#ps^-1#memDL:GDL2 + 1.stallDL:GDL
bs GDL2 1.t:GDLa
bs GDLa 1.t:GDLb
bs GDLb 1.t:GDLc
bs GDLc 1.t:GDL5
*no data growth
bs GDL5 1@1.gs^-1#ps^-1#memDL:GDL + 1.stallDL:GDL5

*Finally the viewer
bs GV 1@1.gs^-1#ps^-1#memV:GV2 + 1.stallV:GV
bs GV2 1.t:GVa
bs GVa 1.t:GVb
bs GVb 1.t:GVc
bs GVc 1.t:GV5
bs GV5 1@1.gs^-1#ps^-1#memV:GV + 1.stallV:GV5

*Only one thing can 'have' the RTC at a time

bs Sem 1.gs:SB + 1.gs#ps:Sem + 1.t:Sem
bs SB 1.ps:Sem + 1.t:SB

basi P stallPE,stallDL,stallV,memPE,memDL,memV

btr Sys GPE|GDL|GV|Sem/P
```

See [16] for a full explanation of the above syntax. As a quick overview the above describes interacting automata: `bs` being a state binding; iteraction by 'handshaking' on action name duals viz `gs` vs `gs^-1`;`@` representing priority; `#` parallel action product; `|` parallel automata product and `/` action permission. This representation was used as there is a substantial toolset allowing analysis of such prioritised probabiliatic concurrent automata. It should be noted that despite the simplicity of the problem all of this expressive power is required to present it within a compositional form.

This model can be used as a template for fixed time models with differing durations of task. It can also be trivially extended to include more tasks operating on the same patterns.

2.2 Results

The interesting question is how long does the above take to stabilise given differing patterns of upload and download times for each task? In the table below the settle time is the time that the system requires to reach its stable cycle where we will see no more stalls. In other words the point at which the schedule has been established.

All of these results are computed to 99.999% coverage of the probability distribution. I will describe the system by the pattern of memory usage so $[(1,1),(1,1),(1,1)]$ will denote a three

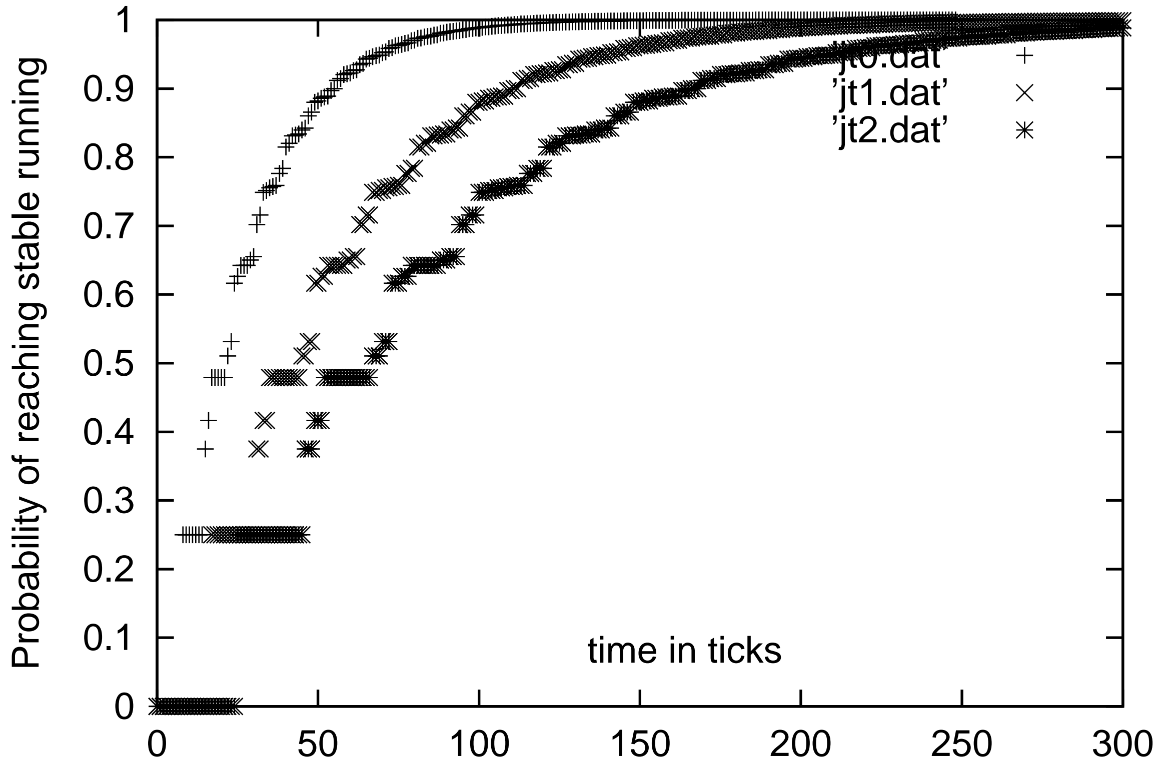

Figure 1: Probability distribution of stabilisation time. For series with $[(1,1),(1,1),(1,1)]$, $[(2,2),(2,2),(2,2)]$, $[(3,3),(3,3),(3,3)]$. Shows time independence

component system, with each component using the resource for 1 unit of time at the start and finish, we assume that these usages are seperated by the total amount of time the other components require for their transfers. In this case 4.

Work Pattern	Average Settle	Cycles
$[(1,1),(1,1),(1,1)]$	26.9973515378	4.5
$[(2,2),(2,2),(2,2)]$	54.9946932549	4.58
$[(3,3),(3,3),(3,3)]$	81.9920447927	4.56
$[(1,1),(1,1),(1,1),(1,1)]$	61.1932750914	7.65
$[(1,1),(1,1),(1,1),(1,1),(1,1)]$	117.40695874	11.7
$[(1,1),(1,1),(1,1),(1,1),(1,1),(1,1)]$	208.412535603	17.34
$[(1,1),(1,1),(1,1),(1,1),(1,1),(1,1),(1,1)]$	353.8619251	25.28
$[(1,1),(1,1),(2,2)]$	32.0261161239	4.0
$[(1,1),(1,1),(3,3)]$	40.5773281519	4.05
$[(1,1),(1,1),(4,4)]$	49.6312614071	4.13

2.3 Non-atomic transfers

A further consideration is that of block size. In the above models memory transfers were treated as atomic and uninterruptable once started, no matter what their duration. In this series we treat the access as interuptable. Obviously the (1,1) types are uninteresting in this view. The model

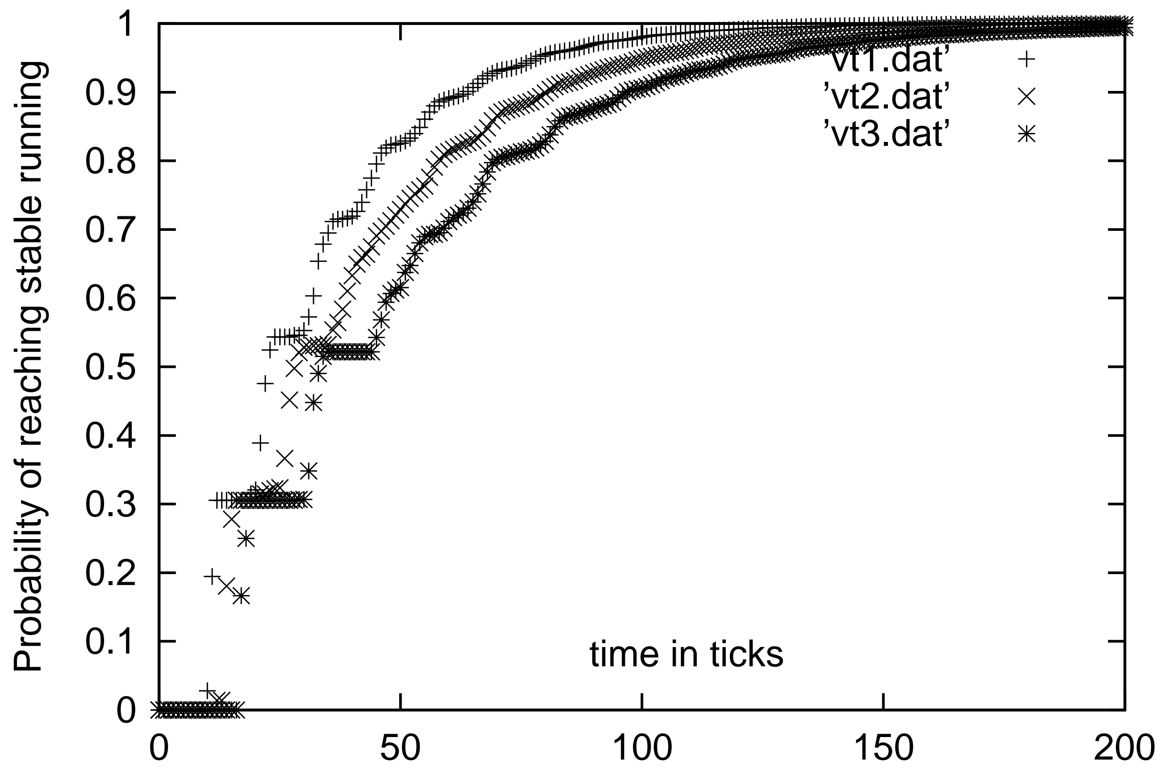

Figure 2: Probability distribution of stabilisation time. For series with $[(1,1),(1,1),(2,2)]$, $[(1,1),(1,1),(3,3)]$, $[(1,1),(1,1),(4,4)]$ Shows time independence again.

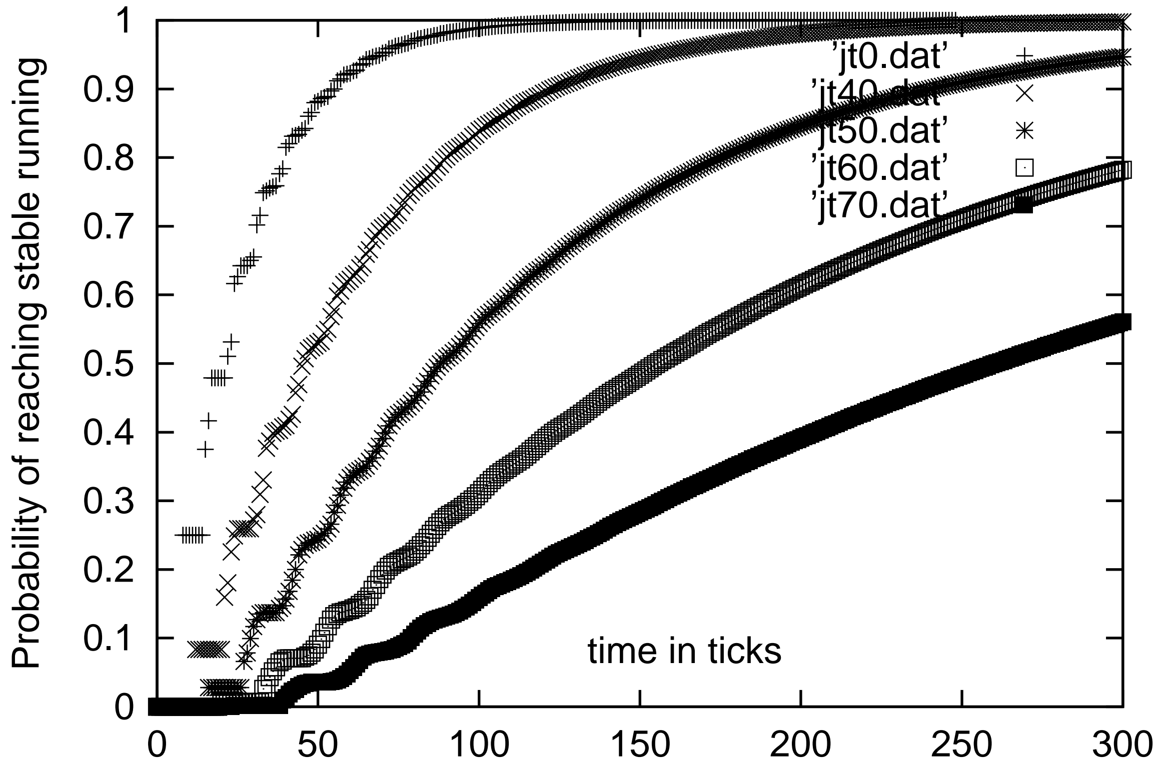

Figure 3: Probability distribution of stabilisation time. For series with $[(1,1),(1,1),(1,1)]$, $[(1,1),(1,1),(1,1),(1,1)]$, $[(1,1),(1,1),(1,1),(1,1),(1,1)]$, $[(1,1),(1,1),(1,1),(1,1),(1,1),(1,1)]$, $[(1,1),(1,1),(1,1),(1,1),(1,1),(1,1),(1,1)]$. Shows effects of introducing more transfers in parallel.

prototype:

```
*Simple scheduable model
*the work description for the simple task
*2,2 2,2 2,2 interleaved version

bs GPE 1@1.gs^-1#ps^-1#memPE:GPE1 + 1.stallPE:GPE
bs GPE1 1@1.gs^-1#ps^-1#memPE:GPE2 + 1.stallPE:GPE1
bs GPE2 1.t:GPEa
bs GPEa 1.t:GPEb
bs GPEb 1.t:GPEc
bs GPEc 1.t:GPEd
bs GPEd 1.t:GPEe
bs GPEe 1.t:GPEf
bs GPEf 1.t:GPEg
bs GPEg 1.t:GPE5

bs GPE5 1@1.gs^-1#ps^-1#memPE:GPE6 + 1.stallPE:GPE5
bs GPE6 1@1.gs^-1#ps^-1#memPE:GPE + 1.stallPE:GPE6

*Only one thing can 'have' the Resource at a time

bs Sem 1.gs:SB + 1.gs#ps:Sem + 1.t:Sem
bs SB 1.ps:Sem + 1.t:SB

basi P stallPE,stallDL,stallV,memPE,memDL,memV

btr Sys3 GPE|GPE|GPE|Sem/P
```

The results for the interleaved model:

Work Pattern	Average Settle	Cycles
$[(2,2),(2,2),(2,2)]$	153.057467444	12.67
$[(3,3),(3,3),(3,3)]$	412.091077322	22.89
$[(4,4),(4,4),(4,4)]$	822.871615425	34.29
$[(5,5),(5,5),(5,5)]$	1399.1757078	46.63
$[(6,6),(6,6),(6,6)]$	2151.76327363	59.77
$[(7,7),(7,7),(7,7)]$	3045.74432622	72.52
$[(8,8),(8,8),(8,8)]$	4218.40442886	87.88
$[(1,1),(1,1),(2,2)]$	56.7548804331	7.09
$[(1,1),(1,1),(3,3)]$	108.785176272	10.88
$[(1,1),(1,1),(4,4)]$	191.166229448	15.93

2.3.1 Implications for Peturbation

All of the above models assumed that the stages took a fixed time to perform their actions, and therefore where eventually schedulable. In a real system there will be variation. If any stage varies when we have tuned the system then it will impact on the slot of another stage. In particular we now have a question as to whether the system can reach a smoothly running state at all. If

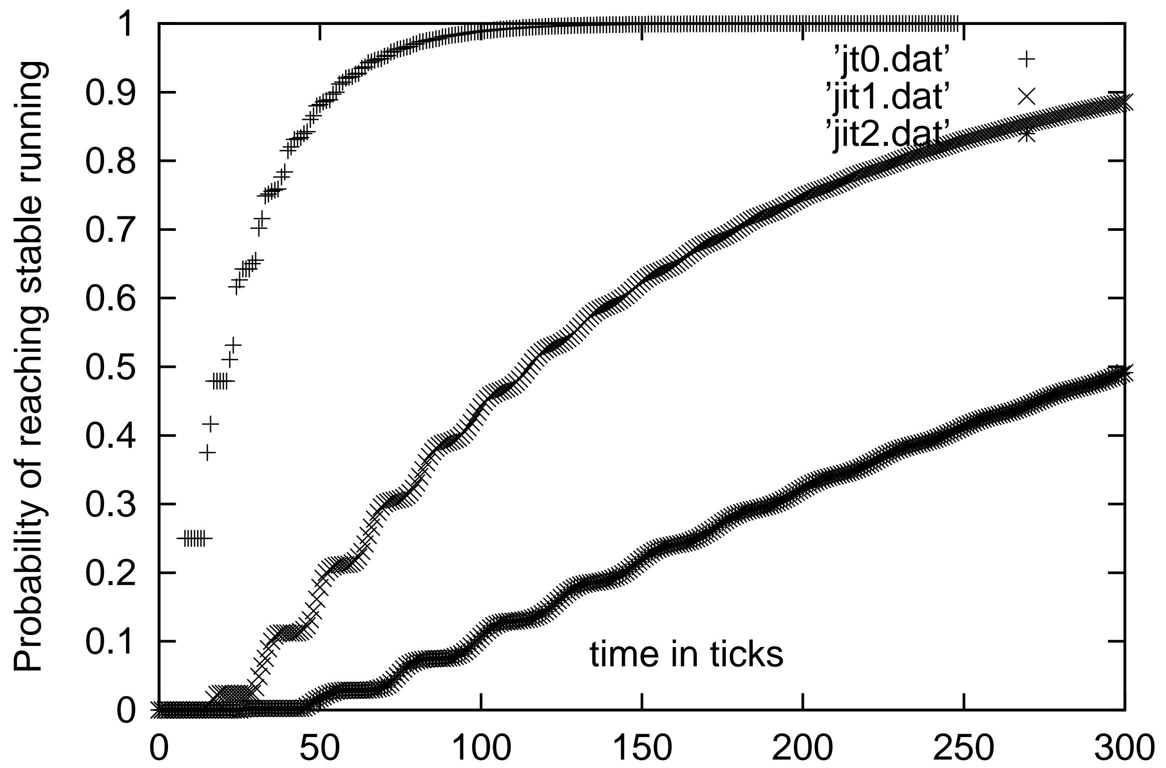

Figure 4: Probability distribution of stabilisation time. For series with $[(1,1),(1,1),(1,1)]$, $[(2,2),(2,2),(2,2)]$,$[(3,3),(3,3),(3,3)]$. Shows strong dependency on number of exchanges required.

we assume that perturbation is rare, then if time to stabilise is much shorter than peturbation rate the system could be expected to spend the majority of its time running efficiently. Given a time to stability t then the system will certainly not tolerate perturbation frequencies of greater than $\frac{1}{t}$ assuming that only one stage is responsible for the peturbation. If n stages contribute to the peturbation and they have equal probability p of causing a peturbation, then we require that $\frac{1}{t} > 1 - (1-p)^n$. Which for $n > 1$ will require that p is very small indeed.

3 Adding perturbation

The simplest perturbation we can add is to permit one of the stages to take one tick longer than it should (for the schedule to be maintained) with some probability. If it is the first stage then we alter the model above as follows:

```
*Simple scheduable model 3 tasks
*the work description for the simple task
bs GPE 1@1.gs^-1#ps^-1#memPE:GPE2 + 1.stallPE:GPE
bs GPE2 1.t:GPEa
bs GPEa 1.t:GPEb
bs GPEb 1.t:GPEc
bs GPEc (1-p+pe).t:GPE5 + p-pe.t:GPEd
bs GPEd 1.t:GPE5
bs GPE5 1@1.gs^-1#ps^-1#memPE:GPE + 1.stallPE:GPE5
```

With these models we get the expected result of no stability, and if the peturbation rate is of the order calculated above then the system spends essentially no time operating at full capacity.

Function for probability, obtained as in [16], of stalling:

```
[0.075,0.125]-: ((9.80721562581 * 1.0 + (543.073900111*pe * pe * pe) + (71.4412148578*pe)
                 - (192.504176011*pe * pe)))/((90.0 * 1.0))
[0.125,0.175]-: ((12.9574300306 * 1.0 + (319.415605751*pe * pe * pe) + (55.605903295*pe)
                 - (129.833552194*pe * pe)))/((90.0 * 1.0))
[0.175,0.225]-: ((15.4486826836 * 1.0 + (201.338227107*pe * pe * pe) + (44.676836668*pe)
                 - (91.6648611986*pe * pe)))/((90.0 * 1.0))
[0.225,0.275]-: ((17.4760847264 * 1.0 + (133.955916834*pe * pe * pe) + (36.8297157888*pe)
                 - (66.9521638903*pe * pe)))/((90.0 * 1.0))
[0.275,0.325]-: ((19.1654658274 * 1.0 + (92.9263876288*pe * pe * pe) + (31.0243143334*pe)
                 - (50.1773490782*pe * pe)))/((90.0 * 1.0))
[0.325,0.375]-: ((20.6019317553 * 1.0 + (66.9109564888*pe * pe * pe) + (26.6312841964*pe)
                 - (38.331465786*pe * pe)))/((90.0 * 1.0))
[0.375,0.425]-: ((21.8454327248 * 1.0 + (49.8746229433*pe * pe * pe) + (23.2529674589*pe)
                 - (29.6595335269*pe * pe)))/((90.0 * 1.0))
[0.425,0.475]-: ((22.9397638366 * 1.0 + (38.4125033764*pe * pe * pe) + (20.6295500751*pe)
                 - (23.0934808292*pe * pe)))/((90.0 * 1.0))
[0.475,0.525]-: ((23.9180339972 * 1.0 + (30.6040183252*pe * pe * pe) + (18.5869142979*pe)
                 - (17.9536394516*pe * pe)))/((90.0 * 1.0))
```

Plotted using central values above in Figure 5.

4 Model with dependence

In this instance the tasks act like an n-place buffer passing the work between them. We assume that they are buffered by processes and our functional units to be described by the following set of

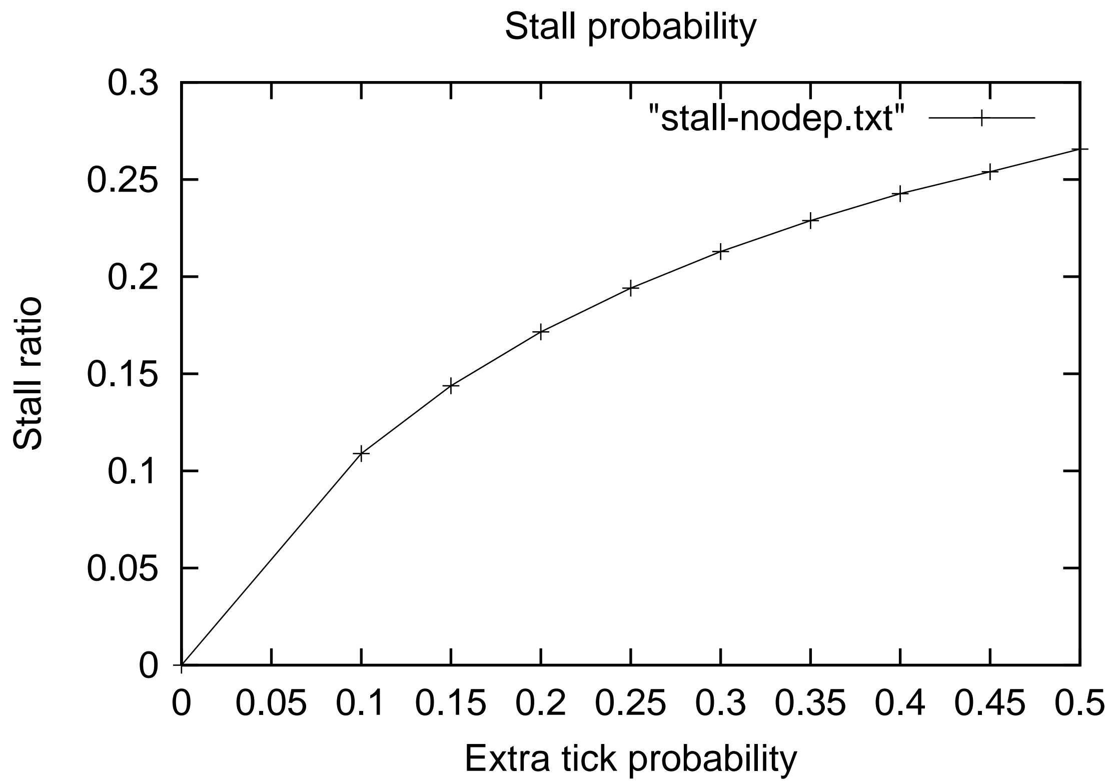

Figure 5: Ratio of time that some processing element is stalled against a single tick error probability, for 3 component system.

processes (TCCS style):

$$DTask_i \stackrel{def}{=} get_i.\overline{getM}.(input_time_i).\overline{putM}.DTask_iW$$
$$DTask_iW \stackrel{def}{=} (work_time_i).DTask_iO$$
$$DTask_io \stackrel{def}{=} \overline{getM}.(output_time_i).\overline{putM}.\overline{put_{i+1}}.DTask_i$$

and the complete system is described as the following:

$$Sem \stackrel{def}{=} getM.putM.Sem$$
$$BufN(0) \stackrel{def}{=} put.BufN(1)$$
$$BufN(i) \stackrel{def}{=} put.BufN(i+1) + get.BufN(i-1)(0 < i < N)$$
$$BufN(N) \stackrel{def}{=} get.BufN(i-1)$$
$$Buf_i(0) \stackrel{def}{=} BufN(0)[get_i/get, put_i/put]$$
$$DSys \stackrel{def}{=} (DTask_0|Buf_1(0)|DTask_1|...Buf_{N-1}(0)|DTask_n|Sem)\backslash\{getM, putM, get_i\}$$

4.1 Realised Model

Below is the prototype for the model with the base timings:

```
*Simple scheduable model
*the work description for the simple task
*1,1 1,1 1,1 version
*In this version we model some 1 place queues
*to look at the effects of starvation


bs ST1 1@1.gs^-1#ps^-1#mem1:ST1_2 + 1.stall1:ST1
bs ST1_2 1.t:ST1_a
bs ST1_a 1.t:ST1_b
bs ST1_b 1.t:ST1_c
bs ST1_c 1.t:ST1_5
bs ST1_5 1@1.gs^-1#ps^-1#put1^-1#mem1:ST1 + 1.stall1:ST1_5

*A fast 1 place buffer
bs B1E 1.put1#get1:B1E + 1.put1:B1F + 1.t:B1E
bs B1F 1.get1#put1:B1F + 1.get1:B1E + 1.t:B1F

bs ST2 1@1.gs^-1#ps^-1#get1^-1#mem2:ST2_2 + 1.stall2:ST2
bs ST2_2 1.t:ST2_a
bs ST2_a 1.t:ST2_b
bs ST2_b 1.t:ST2_c
bs ST2_c 1.t:ST2_5
bs ST2_5 1@1.gs^-1#ps^-1#put2^-1#mem2:ST2 + 1.stall2:ST2_5


*A fast 1 place buffer
bs B2E 1.put2#get2:B2E + 1.put2:B2F + 1.t:B2E
bs B2F 1.get2#put2:B2F + 1.get2:B2E + 1.t:B2F
```

```
bs ST3 1@1.gs^-1#ps^-1#get2^-1#mem3:ST3_2 + 1.stall3:ST3
bs ST3_2 1.t:ST3_a
bs ST3_a 1.t:ST3_b
bs ST3_b 1.t:ST3_c
bs ST3_c 1.t:ST3_5
bs ST3_5 1@1.gs^-1#ps^-1#page#mem3:ST3 + 1.stall3:ST3_5

*Only one thing can 'have' the RTC at a time

bs Sem 1.gs:SB + 1.gs#ps:Sem + 1.t:Sem
bs SB 1.ps:Sem + 1.t:SB

basi P stall1,stall2,stall3,mem1,mem2,mem3,page

btr Sys ST1|B1E|ST2|B2E|ST3|Sem/P
```

Again this can be considered as a template for a range on investigations...

4.2 Results

When the stages have some depedence:

Work Pattern	Average Settle	Cycles
$[(1,1),(1,1),(1,1)]$	65.5872493386	10.91
$[(2,2),(2,2),(2,2)]$	132.174488746	11.01
$[(3,3),(3,3),(3,3)]$	197.761738085	10.98

The same models but with interleaved access:

Work Pattern	Average Settle	Pages
$[(2,2),(2,2),(2,2)]$	202.690496981	16.89
$[(3,3),(3,3),(3,3)]$[1]	599.705759723	33.32

4.3 Dependence and Variation

In this model we allow the first stage in the dependent model to vary its excution time in the same manner as we did before. The model being:

```
*Simple scheduable model
*the work description for the simple task
*1,1 1,1 1,1 version
*In this version we model some 1 place queues
*to look at the effects of starvation and variation
*gen_fun("Sys","stall","p",0.1,0.75,0.05,500,200,3);

bs ST1 1@1.gs^-1#ps^-1#mem1:ST1_2 + 1.stall:ST1
bs ST1_2 1.t:ST1_a
bs ST1_a 1.t:ST1_b
bs ST1_b 1.t:ST1_c
bs ST1_c (1-(p-pe)).t:ST1_5 + (p-pe).t:ST1_d
```

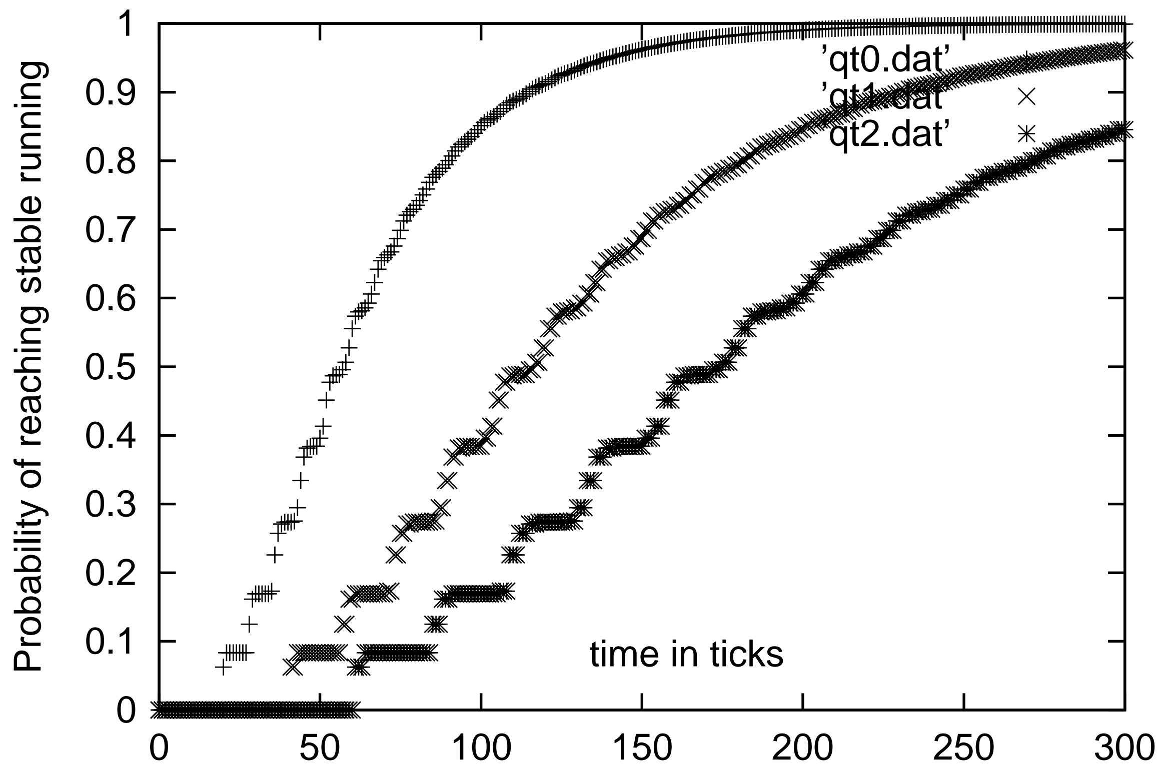

Figure 6: Probability distribution of stabilisation time. For series with $[(1,1),(1,1),(1,1)]$, $[(2,2),(2,2),(2,2)]$,$[(3,3),(3,3),(3,3)]$ with queue dependencies between the stages.

```
bs ST1_d 1.t:ST1_5
bs ST1_5 1@1.gs^-1#ps^-1#put1^-1#mem1:ST1 + 1.stall:ST1_5

*A fast 1 place buffer
bs B1E 1.put1#get1:B1E + 1.put1:B1F + 1.t:B1E
bs B1F 1.get1#put1:B1F + 1.get1:B1E + 1.t:B1F

bs ST2 1@1.gs^-1#ps^-1#get1^-1#mem2:ST2_2 + 1.stall:ST2
bs ST2_2 1.t:ST2_a
bs ST2_a 1.t:ST2_b
bs ST2_b 1.t:ST2_c
bs ST2_c 1.t:ST2_5
bs ST2_5 1@1.gs^-1#ps^-1#put2^-1#mem2:ST2 + 1.stall:ST2_5


*A fast 1 place buffer
bs B2E 1.put2#get2:B2E + 1.put2:B2F + 1.t:B2E
bs B2F 1.get2#put2:B2F + 1.get2:B2E + 1.t:B2F

bs ST3 1@1.gs^-1#ps^-1#get2^-1#mem3:ST3_2 + 1.stall:ST3
bs ST3_2 1.t:ST3_a
bs ST3_a 1.t:ST3_b
bs ST3_b 1.t:ST3_c
bs ST3_c 1.t:ST3_5
bs ST3_5 1@1.gs^-1#ps^-1#page#mem3:ST3 + 1.stall:ST3_5

*Only one thing can 'have' the RTC at a time

bs Sem 1.gs:SB + 1.gs#ps:Sem + 1.t:Sem
bs SB 1.ps:Sem + 1.t:SB

basi P stall1,stall2,stall,mem1,mem2,mem3,page

btr Sys ST1|B1E|ST2|B2E|ST3|Sem/P
```

The function expressing the expected number of stalls is:

```
[0.075,0.125]-: ((34.5442989379 * 1.0 - (1854.7111199*pe * pe * pe) - (250.093881933*pe)
                - (639.206366747*pe * pe)))/((208.0 * 1.0))
[0.125,0.175]-: ((45.7668108721 * 1.0 - (1010.6953828*pe * pe * pe) - (196.357982816*pe)
                - (435.660124237*pe * pe)))/((208.0 * 1.0))
[0.175,0.225]-: ((54.6684836203 * 1.0 - (637.567449593*pe * pe * pe) - (158.610204375*pe)
                - (313.480859261*pe * pe)))/((208.0 * 1.0))
[0.225,0.275]-: ((61.9162149547 * 1.0 - (439.769221405*pe * pe * pe) - (131.148221969*pe)
                - (232.256687812*pe * pe)))/((208.0 * 1.0))
[0.275,0.325]-: ((67.9561003341 * 1.0 - (318.561199877*pe * pe * pe) - (110.722717549*pe)
                - (174.954542245*pe * pe)))/((208.0 * 1.0))
[0.325,0.375]-: ((73.0968004741 * 1.0 - (238.336050491*pe * pe * pe) - (95.3202069576*pe)
                - (132.974330643*pe * pe)))/((208.0 * 1.0))
[0.375,0.425]-: ((77.5598888106 * 1.0 - (183.545844444*pe * pe * pe) - (83.6277502955*pe)
```

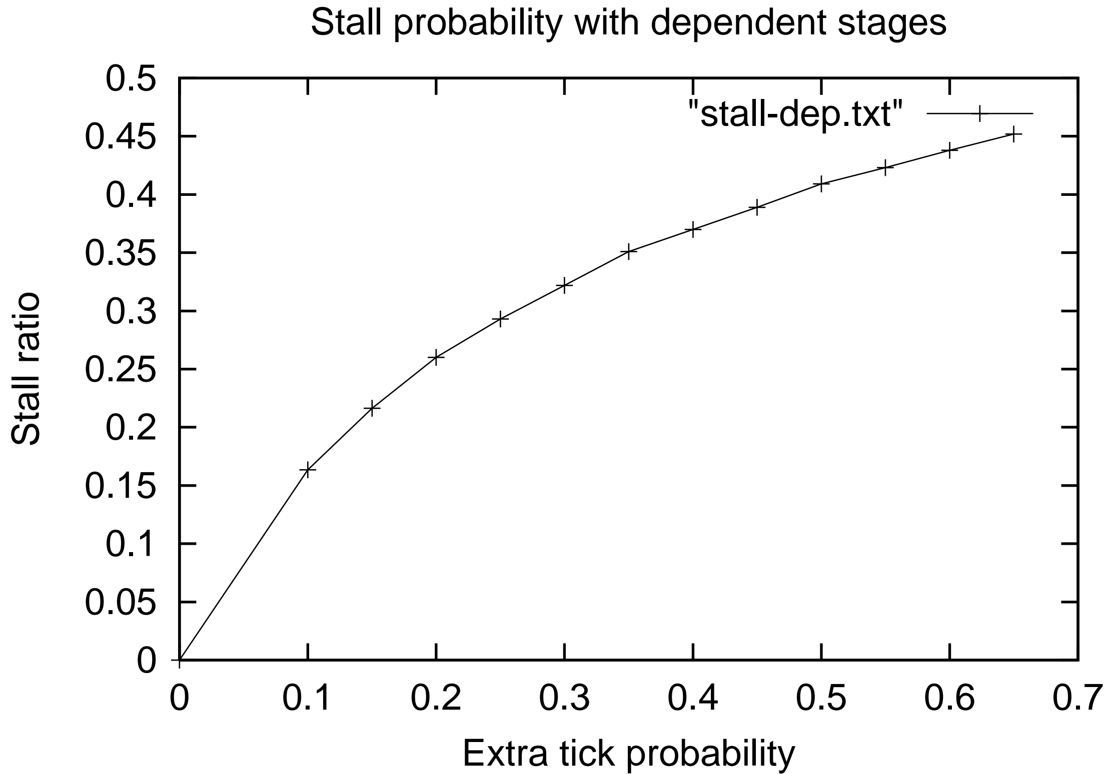

Figure 7: Ratio of time that some processing element is stalled against a single tick error probability, for 3 component system.

```
               - (101.259251116*pe * pe)))/((208.0 * 1.0))
[0.425,0.475]-: ((81.5101123724 * 1.0 - (145.69530865*pe * pe * pe) - (74.7642229496*pe)
               - (76.5613812393*pe * pe)))/((208.0 * 1.0))
[0.475,0.525]-: ((85.0742769398 * 1.0 - (119.657297432*pe * pe * pe) - (68.1296143466*pe)
               - (56.6874399714*pe * pe)))/((208.0 * 1.0))
[0.525,0.575]-: ((88.3535728471 * 1.0 - (102.28878828*pe * pe * pe) - (63.3155145827*pe)
               - (40.085022649*pe * pe)))/((208.0 * 1.0))
[0.575,0.625]-: ((91.4320348286 * 1.0 - (91.685625453*pe * pe * pe) - (60.0511280471*pe)
               - (25.5913582564*pe * pe)))/((208.0 * 1.0))
[0.625,0.675]-: ((94.3827271402 * 1.0 - (86.8521682759*pe * pe * pe) - (58.1714774733*pe)
               - (12.2669145987*pe * pe)))/((208.0 * 1.0))
[0.675,0.725]-: ((97.2726778987 * 1.0 - (87.6811746224*pe * pe * pe) - (57.6013097306*pe)
               + (0.740232438739*pe * pe)))/((208.0 * 1.0))
```

Plotted using central values above in Figure 7.

5 A Possible Solution

One method which reduces the scope for long term drifting before reaching stability is to 'tie' together the write back and read phases of the functional unit. Consequently, our system would be described as follows:

$$TTask_i \stackrel{def}{=} \overline{getM}.(input_time_i).\overline{putM}.TTask_iW$$
$$TTask_iW \stackrel{def}{=} (work_time_i).TTask_iO$$
$$TTask_io \stackrel{def}{=} \overline{getM}.(output_time_i).TTask_iN$$
$$TTask_iN \stackrel{def}{=} (input_time_i).\overline{putM}.TTask_iW$$

and the complete system is described as the following:

$$Sem \stackrel{def}{=} getM.putM.Sem$$
$$Sys \stackrel{def}{=} (TTask_0|TTask_1|...|TTask_n|Sem)\backslash\{getM, putM\}$$

Which is realised as the following model:

```
*Simple scheduable model
*with tied write-back read...
*the work description for the simple task
*1,1 1,1 1,1 version

bs GPE 1@1.gs^-1#ps^-1#memPE:GPE2 + 1.stallPE:GPE
bs GPE2 1.t:GPEa
bs GPEa 1.t:GPEb
bs GPEb 1.t:GPEc
bs GPEc 1.t:GPE5
bs GPE5 1@1.gs^-1#memPE:GPER + 1.stallPE:GPE5
bs GPER 1@1.ps^-1#memPE:GPE2 + 1.stallPE:GPE

*Only one thing can 'have' the RTC at a time

bs Sem 1.gs:SB + 1.gs#ps:Sem + 1.t:Sem
bs SB 1.ps:Sem + 1.t:SB

basi P stallPE,stallDL,stallV,memPE,memDL,memV

btr Sys GPE|GPE|GPE|Sem/P
```

This will stabilise in the time taken to perform one complete cycle, independent on the number of components. Note that both the read and write are jointly atomic at the point the next task starts.

5.1 Variation and tying

The effect of variation on such systems will be greatly reduced as they always restabilise within 1 cycle, for completeness we present such a model

```
*Simple scheduable model
*with tied write-back read...
*the work description for the simple task
*1,1 1,1 1,1 version
*and one variable stage to demonstrate neato behaviour

bs GPE 1@1.gs^-1#ps^-1#memPE:GPE2 + 1.stallPE:GPE
```

```
bs GPE2 1.t:GPEa
bs GPEa 1.t:GPEb
bs GPEb 1.t:GPEc
bs GPEc 1.t:GPE5
bs GPE5 1@1.gs^-1#memPE:GPER + 1.stallPE:GPE5
bs GPER 1@1.ps^-1#memPE:GPE2 + 1.stallPE:GPE

bs VS 1@1.gs^-1#ps^-1#mem1:VS2 + 1.stallV:VS
bs VS2 1.t:VSa
bs VSa 1.t:VSb
bs VSb 1.t:VSc
bs VSc (1-p-pe).t:VS5 + (p-pe).t:VSd
bs VSd 1.t:VS5
bs VS5 1@1.gs^-1#mem1:VSR + 1.stallV:VS5
bs VSR 1@1.ps^-1#mem1:VS2 + 1.stallV:VS

*Only one thing can 'have' the RTC at a time

bs Sem 1.gs:SB + 1.gs#ps:Sem + 1.t:Sem
bs SB 1.ps:Sem + 1.t:SB

basi P stallPE,stallDL,stallV,mem1,memPE,memV

btr Sys GPE|GPE|VS|Sem/P
```

Function for probability of stalling:

```
[0.075,0.125]-: ((1.27850414294 * 1.0 + (12.8779124654*pe * pe * pe)
                - (3.49968577715*pe * pe) + (12.5613215633*pe)))/((39.0 * 1.0))
[0.125,0.175]-: ((1.90243299844 * 1.0 + (18.9732168524*pe * pe * pe)
                - (1.56550634794*pe * pe) + (12.3304655633*pe)))/((39.0 * 1.0))
[0.175,0.225]-: ((2.51613385024 * 1.0 - (6.63815417227*pe * pe * pe)
                - (0.935011677382*pe * pe) + (12.180659459*pe)))/((39.0 * 1.0))
[0.225,0.275]-: ((3.12000272106 * 1.0 - (12.8085414013*pe * pe * pe)
                - (2.17742802525*pe * pe) + (12.0084877485*pe)))/((39.0 * 1.0))
[0.275,0.325]-: ((3.71428708506 * 1.0 + (4.40283776806*pe * pe * pe)
                - (2.63904040928*pe * pe) + (11.7858850301*pe)))/((39.0 * 1.0))
[0.325,0.375]-: ((4.2992133606 * 1.0 + (11.4437897463*pe * pe * pe)
                - (1.63284312143*pe * pe) + (11.5838147755*pe)))/((39.0 * 1.0))
[0.375,0.425]-: ((4.87500043445 * 1.0 - (2.12037575711*pe * pe * pe)
                - (1.05669223035*pe * pe) + (11.4322395186*pe)))/((39.0 * 1.0))
[0.425,0.475]-: ((5.44186056914 * 1.0 - (11.8965134169*pe * pe * pe)
                - (1.90012125786*pe * pe) + (11.2741494256*pe)))/((39.0 * 1.0))
[0.475,0.525]-: ((5.99999944602 * 1.0 + (0.265541069844*pe * pe * pe)
                - (2.62582986671*pe * pe) + (11.0694666371*pe)))/((39.0 * 1.0))
```

The stall rates are presented using the central values in Figure 8

The relative advantage of tieing is presented in Figure 9

5.2 Dependence and tyeing

We consider the effect of tieing the write/read phases together in the queueing context. The model examines is a folows:

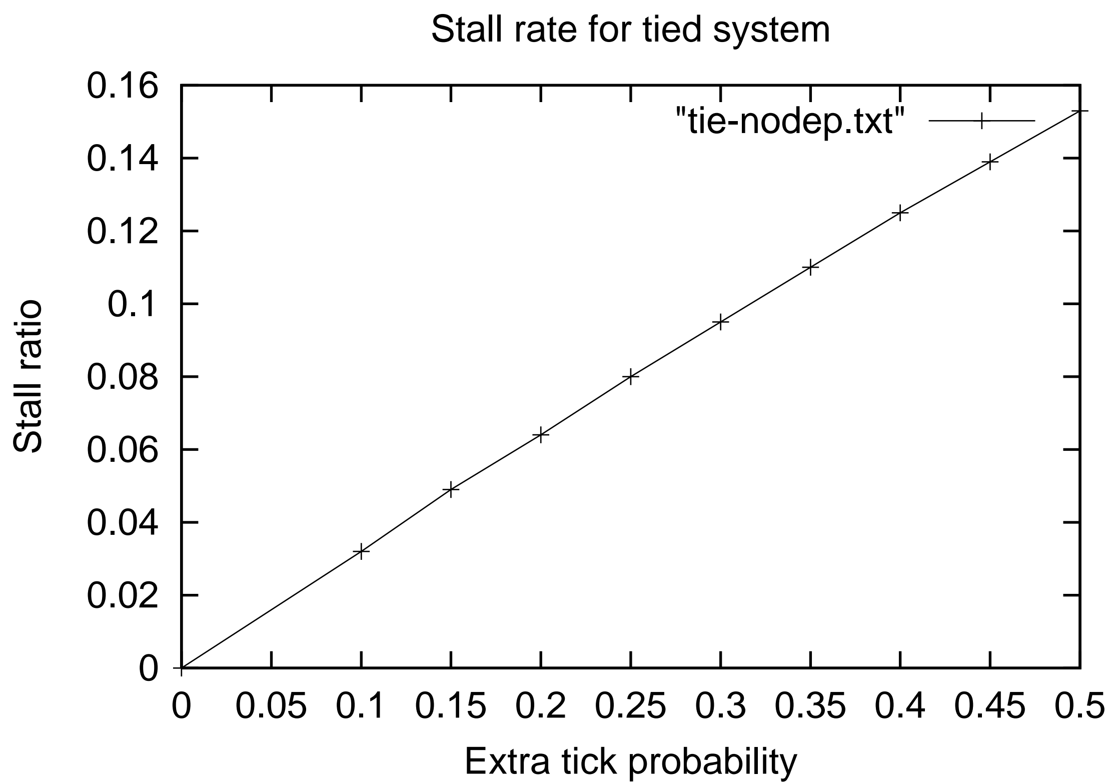

Figure 8: The stall rates tieing the write out and read next phases for probability of variation to 0.5

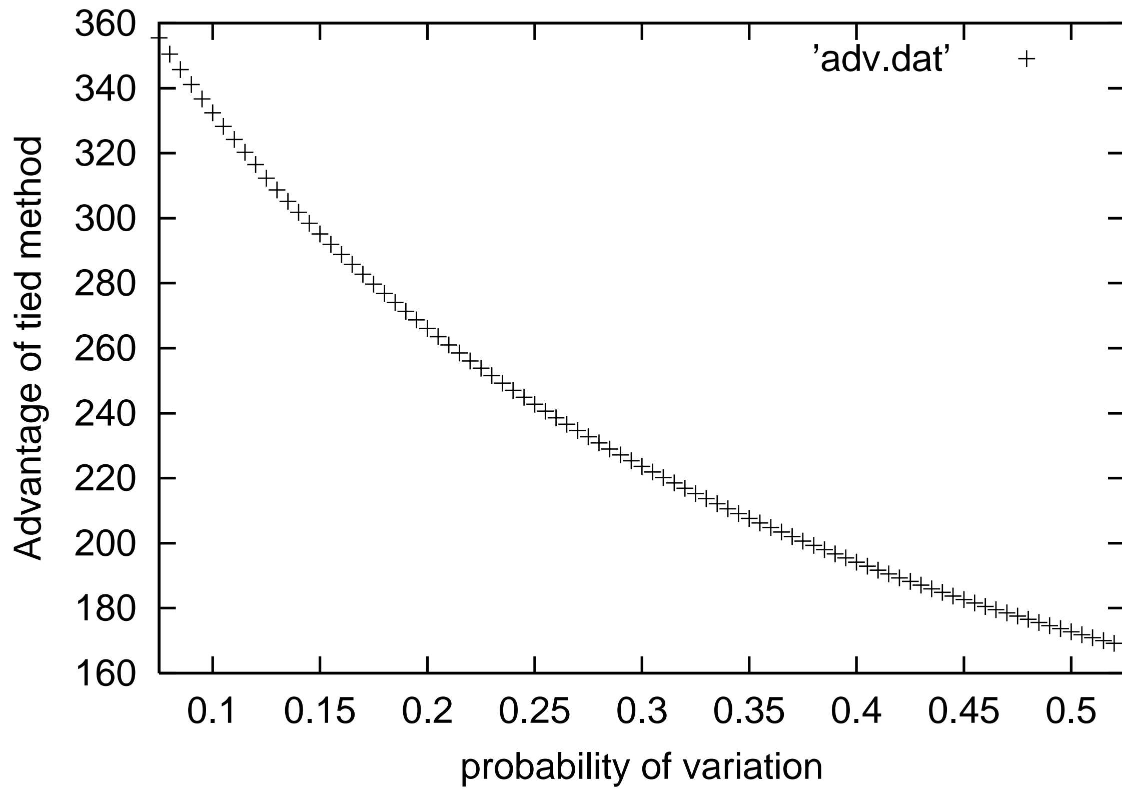

Figure 9: The advantage of tieing the write out and read next phases, expressed as precentage increase in the number of stalls, for probability of variation from 0.075 to 0.525.

```
*Simple scheduable model
*the work description for the simple task
*1,1 1,1 1,1 version
*In this version we model some 1 place queues
*to look at the effects of starvation and variation
*now tie the write back/reads together...if possible.
*gen_fun("Sys","stall","p",0.1,0.75,0.05,500,200,3);

bs ST1 1@1.gs^-1#ps^-1#mem1:ST1_2 + 1.stall:ST1
bs ST1_2 1.t:ST1_a
bs ST1_a 1.t:ST1_b
bs ST1_b 1.t:ST1_c
bs ST1_c (1-(p-pe)).t:ST1_5 + (p-pe).t:ST1_d
bs ST1_d 1.t:ST1_5
bs ST1_5 1@1.gs^-1#put1^-1#mem1:ST1_5b + 1.stall:ST1_5
bs ST1_5b 1.ps^-1#mem1:ST1_2


*A fast 1 place buffer
bs B1E 1.put1#get1:B1E + 1.put1:B1F + 1.t:B1E
bs B1F 1.get1#put1:B1F + 1.get1:B1E + 1.t:B1F

bs ST2 1@1.gs^-1#ps^-1#get1^-1#mem2:ST2_2 + 1.stall:ST2
bs ST2_2 1.t:ST2_a
bs ST2_a 1.t:ST2_b
bs ST2_b 1.t:ST2_c
bs ST2_c 1.t:ST2_5
bs ST2_5 1@1.gs^-1#put2^-1#mem2:ST2_5b + 1.stall:ST2_5
*if the second bit gets stalled then may be waiting on data
bs ST2_5b 1@1.ps^-1#get1^-1#mem2:ST2_2 + 1.ps^-1#stall:ST2


*A fast 1 place buffer
bs B2E 1.put2#get2:B2E + 1.put2:B2F + 1.t:B2E
bs B2F 1.get2#put2:B2F + 1.get2:B2E + 1.t:B2F

bs ST3 1@1.gs^-1#ps^-1#get2^-1#mem3:ST3_2 + 1.stall:ST3
bs ST3_2 1.t:ST3_a
bs ST3_a 1.t:ST3_b
bs ST3_b 1.t:ST3_c
bs ST3_c 1.t:ST3_5
bs ST3_5 1@1.gs^-1#page#mem3:ST3_5b + 1.stall:ST3_5
bs ST3_5b 1@1.ps^-1#get2^-1#mem3:ST3_2 + 1.ps^-1#stall:ST3

*Only one thing can 'have' the RTC at a time
*need quick hand over both types
```

```
bs Sem 1.gs:SB + 1.gs#ps:Sem + 1.t:Sem
bs SB 1.ps:Sem + 1.t:SB

basi P stall1,stall2,stall,mem1,mem2,mem3,page

btr Sys ST1|B1E|ST2|B2E|ST3|Sem/P
```

Function for probability of stalling:

```
[0.075,0.125]-: ((3.7694040293 * 1.0 - (28.8762209337*pe * pe * pe) - (10.1735708261*pe * pe)
                 - (37.0868725663*pe)))/((115.0 * 1.0))
[0.125,0.175]-: ((5.609493871 * 1.0 - (54.8382523631*pe * pe * pe) - (4.51332159559*pe * pe)
                 - (36.3639588525*pe)))/((115.0 * 1.0))
[0.175,0.225]-: ((7.41931176193 * 1.0 + (19.6379594807*pe * pe * pe) - (2.72740485816*pe * pe)
                 - (35.9172709897*pe)))/((115.0 * 1.0))
[0.225,0.275]-: ((9.1999951753 * 1.0 + (37.7361292036*pe * pe * pe) - (6.41309038249*pe * pe)
                 - (35.4093525573*pe)))/((115.0 * 1.0))
[0.275,0.325]-: ((10.9523823044 * 1.0 - (13.0108851412*pe * pe * pe) - (7.78002882898*pe * pe)
                 - (34.7530660939*pe)))/((115.0 * 1.0))
[0.325,0.375]-: ((12.6771676836 * 1.0 - (33.7628007314*pe * pe * pe) - (4.81479796541*pe * pe)
                - (34.157289865*pe)))/((115.0 * 1.0))
[0.375,0.425]-: ((14.3750025584 * 1.0 + (6.24165501646*pe * pe * pe) - (3.11659975737*pe * pe)
                - (33.7103793594*pe)))/((115.0 * 1.0))
[0.425,0.475]-: ((16.0465142539 * 1.0 + (35.0760828631*pe * pe * pe) - (5.60414174623*pe * pe)
                - (33.2442540966*pe)))/((115.0 * 1.0))
[0.475,0.525]-: ((17.6923095344 * 1.0 - (0.774106823822*pe * pe * pe) - (7.74435829931*pe * pe)
                - (32.6407706339*pe)))/((115.0 * 1.0))
[0.525,0.575]-: ((19.3129744707 * 1.0 - (49.3239215182*pe * pe * pe) - (4.86097027699*pe * pe)
                - (32.0569974103*pe)))/((115.0 * 1.0))
[0.575,0.625]-: ((20.909074667 * 1.0 - (21.6099097455*pe * pe * pe) - (0.473423695727*pe * pe)
                - (31.692723087*pe)))/((115.0 * 1.0))
[0.625,0.675]-: ((22.4811786551 * 1.0 + (69.3220020434*pe * pe * pe) - (2.80093060871*pe * pe)
                - (31.4101429721*pe)))/((115.0 * 1.0))
[0.675,0.725]-: ((24.0299713239 * 1.0 + (86.582796753*pe * pe * pe) - (11.8757689631*pe * pe)
                - (30.8184002191*pe)))/((115.0 * 1.0))
```

The stall rates are presented using the central values in Figure 10

The relative advantage of this method is given in Figure 11

6 A general model

In this section we reformulate our system presentation to permit time to be parameterised. Unfortunately, to perform this in a finite state manner requires that all times within the system come from a finitely describable probability distribution (which fix times are not!). One clear consequence is that we cannot get smooth running, but can examine the system over a wider range of possibilities.

The new system realisation is:

```
*An atttempt at a general version of the scheduling thingy
*will allow lots of variation so that we can parameterise
*time...
*This version build the Erlangs correctly for greater
```

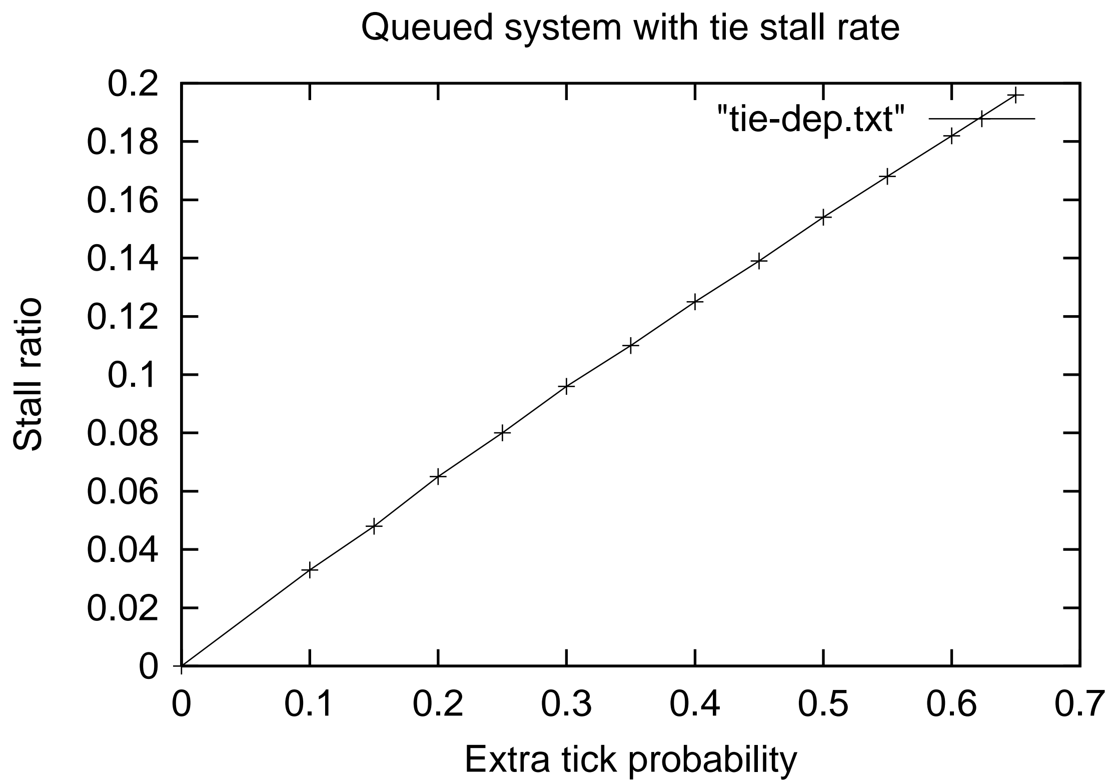

Figure 10: The stall rates tieing the write out and read next phases for probability of variation to 0.5

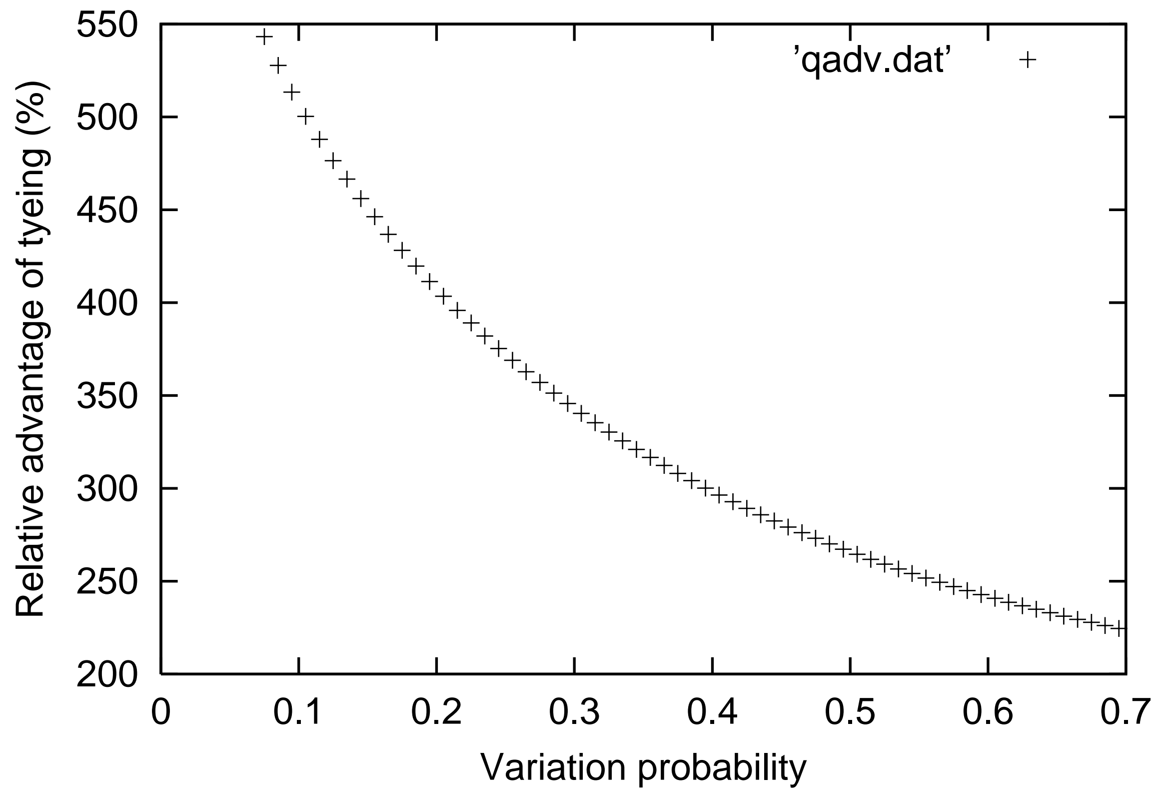

Figure 11: Relative Advantage of tieing in a queueing setting.

```
*accuracy...

bs GetS 1@1.gs^-1#mem:GetDo3 + 1.stall:GetS
bs GetDo3 (p^3*(1-p)).mem#ps^-1:GetDone6 + (p^2*(1-p)).mem:GetDo1 \
          + (p*(1-p)).mem:GetDo2 + (1-p).mem:GetDo3
bs GetDo2 (p^2*(1-p)).mem#ps^-1:GetDone6 + (p*(1-p)).mem:GetDo1 \
          + (1-p).mem:GetDo2
bs GetDo1 p.mem#ps^-1:GetDone6 + (1-p).mem:GetDo1

*for symmetry reasons use an erlang 6 here...
bs GetDone6 (q^6*(1-q)).t:PutS + (q^5*(1-q)).t:GetDone1 \
                               + (q^4*(1-q)).t:GetDone2 \
                               + (q^3*(1-q)).t:GetDone3 \
                               + (q^2*(1-q)).t:GetDone4 \
                               + (q^1*(1-q)).t:GetDone5 \
                               + (1-q).t:GetDone6

bs GetDone5  (q^5*(1-q)).t:PutS + (q^4*(1-q)).t:GetDone1 \
                                + (q^3*(1-q)).t:GetDone2 \
                                + (q^2*(1-q)).t:GetDone3 \
                                + (q^1*(1-q)).t:GetDone4 \
                                + (1-q).t:GetDone5

bs GetDone4  (q^4*(1-q)).t:PutS + (q^3*(1-q)).t:GetDone1 \
                                + (q^2*(1-q)).t:GetDone2 \
                                + (q^1*(1-q)).t:GetDone3 \
                                + (1-q).t:GetDone4

bs GetDone3 (q^3*(1-q)).t:PutS  + (q^2*(1-q)).t:GetDone1 \
                                + (q^1*(1-q)).t:GetDone2 \
                                + (1-q).t:GetDone3

bs GetDone2 (q^2*(1-q)).t:PutS  + (q^1*(1-q)).t:GetDone1 \
                                + (1-q).t:GetDone2

bs GetDone1 q.t:PutS + (1-q).t:GetDone1

*and the symmetric return
bs PutS 1@1.gs^-1#mem:PutDo3 + 1.stall:PutS
bs PutDo3 (p^3*(1-p)).mem#ps^-1:GetS + (p^2*(1-p)).mem:PutDo1 \
          + (p*(1-p)).mem:PutDo2 + (1-p).mem:PutDo3
bs PutDo2 (p^2*(1-p)).mem#ps^-1:GetS+ (p*(1-p)).mem:PutDo1 + (1-p).mem:PutDo2
bs PutDo1 p.mem#ps^-1:GetS + (1-p).mem:PutDo1

*Only one thing can 'have' the RTC at a time
bs Sem 1.gs:SB + 1.gs#ps:Sem + 1.t:Sem
```

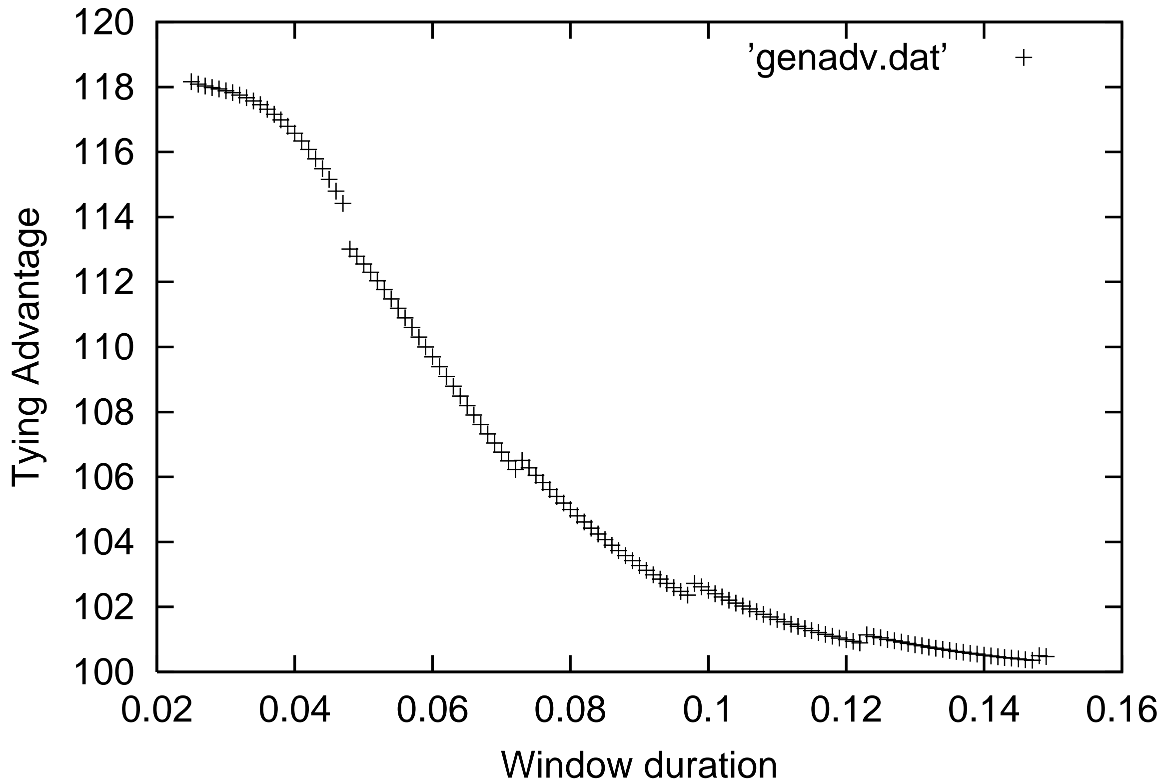

Figure 12: Relative Advantage of tying in the general setting with total utilisation time=60 against the window time duration probability, as the window narrows, stalls become increasingly common, and the tying method starts to have minimal advantage.

```
bs SB 1.ps:Sem + 1.t:SB

basi P stall,mem

btr Sys GetS|GetS|GetS|Sem/P
```

In this setting the underlying variation, caused by the probabilistic representation of the durations, dominates the system behaviour. We get performances well less than 100than the resource utilisation time. However as a check of the effectiveness of the tying approach we can see (Figure 12) that even in a context where it's advantage has been minimized. That is there is no 'good' set of states, it does not make matters worse.

7 Problems with Simulation

These tasks with fixed times pose an interesting problem for conventional discrete event simulation systems. Firstly most of the interesting behaviour is a transient. In which case the classical wait a while and then gather data approach of simulation is liable to underestimate any problems in the system. Of far greater interest and potentially far more damaging is the problem of the core of the simulation approach to concurrent systems. In the core of any simulation engine is an event queuer

[2, 3, 4]. This maintains a list (tree, hash...) of what events are pending and when they should be activated. In all of these situations an interesting problem arises when we have to insert an event at a time equal[6] to that of an event that has already been scheduled. It is usual to assume that this event should be queued after the one that is already there. This allows accurate shutdown and statistics gathering to be embedded as part of the event model. One consequence is that if events occur at regular fixed times, then entities can gain priority as a consequence of when they were originally enqueued. In the case of our perturbed stable system above this will completely change the values obtained for the consequences of a single error. The perturbed process will be the first equeued and consequently we will not see the sepreation of its write and read phases induced by competition and the system will restabilise almost immediately! With no 'knock-on' errors. This will be a gross underestimate of the consequences of the problem.

8 Conclusions

Relying on concurrency to schedule tasks, other than on a single processor system, can have severe implications for system performance. Attempting to run these systems close to or at the limits imposed by shared resources can have major implications for the system performance. The use of these methods is often advocated to cope with systems where the timing can be variable and the scheduler cannot know what loads are going to be imposed in advance. In this situation, then system performance will be very adversely affected. In a true parallel system if there are many interactions with a shared resource, and especially if those interactions are not atomic, then even though a stable schedule may exist the system can well take an effectively infinite amount of time to find it. Even if this time is relatively short any variation will can lead to the system having to restart the hunt for the stable schedule. The values we obtained for this seek time imply that with even extremely low probabilities of variation the system may well never achieve its smooth running state.

All of the above can certainly be eased by tying together the write out and read back of the functional blocks into a single atomic action. This has the effect of establishing the stable cycle in the shortest possible (and deterministic) time. In the presence of variation this has major implications for this approach to scheduling. In this case the variation simple costs the delay time of the varying functional block, with no 'knock-on' consequences.

When the functional stages are mutually dependent as in a processing pipeline, then the situation is substantially worse. The stability time increases, and the consequences of stalls also include the ability to stall dependent stages. This could be reduced, for non-persistent processing, such as printing (there are no infinite documents) by pre-loading the pipe to ensure downstream stages will not be starved. Further this has the effect of permitting stages that 'miss' their window in the schedule to wait for the next window, and hence not induce a thrash on the shared resource. The calculation of preload is very complex, involving the length of the total task, and the anticipated amount of variation in the stages, but approximations for this function may well be cheap, and standard queue methods should give insights as to the depths required.

A further observation is the temptation to use sink processes to absorb free-time on the shared resource, when other users are active, if this is the only shared resource in a system this may well be a good idea. However, if the sink process is exploiting any other shared resource then it may well unbalance the system further by over claiming it. For example, computing results in advance requires memory to store them, this memory may well be needed by the overuuning process whose time slot is being exploited and consequently cause it to perform even more poorly.

Finally as a general observation. Even if the system is not tuned then making the write back and read next phases of the system atomic will still be beneficial. Simply, at the point of a write back, it is unlikely that any more work can be undertaken locally until a read is completed. In comparison, the period before a write includes that for any computational work and does not therefore necessarily constitute a waste of system resources.

References

[1] Gregory R. Andrews **Concurrent Programming**, Principles and Practice, Benjamin Cummings, 1991.

[2] G. Birtwistle, O-J Dahl, B. Myhrhaug and K. Nygaard, Simula Begin, 2nd Edition, Studentlitteratur, Lund, Sweden, 1979.

[3] P. Bratley, B. Fox and L. Schrage, A guide to simulation, second edition, 1987.

[4] G. Birtwistle, DEMOS — discrete event modelling on Simula. Macmillen, 1979.

[5] G. Birtwistle and C. Tofts, A Denotational Semantics for a Process-Based Simulation Language, ACM ToMaCS, 1998.

[6] G. Birtwistle and C. Tofts, Relating operational and denotational descriptions of πDemos, Simulation Practice and Theory, 5:1-33 (1997).

[7] R. Milner, Calculus of Communicating System, LNCS92, 1980.

[8] R. Milner, Calculi for Synchrony and Asynchrony, Theoretical Computer Science 25(3), pp 267-310, 1983.

[9] R. Milner, Communication and Concurrency, Prentice Hall, 1990.

[10] F. Moller and C. Tofts, A Temporal Calculus of Communicating Systems, Proceedings Concur '90, LNCS 458, 1990.

[11] A. Paz, Introduction to probabilistic automata, Academic Press, 1971.

[12] C. Tofts, A Synchronous Calculus of Relative Frequency, CONCUR '90, Springer Verlag, LNCS 458.

[13] C. Tofts, Exact Solutions to Finite State Simulation Problems, Research Report, Department of Computer Science, University of Calgary, 1993.

[14] C. Tofts, Processes with Probabilities, Priorities and Time, FACS 6(5): 536-564, 1994.

[15] C. Tofts, Using Process Algebra to Describe Social Insect Behaviour, Transactions on Simulation, 1994.

[16] C. Tofts, Analytic and locally approximate solutions to properties of probabilistic processes, Proceedings TACAS '95, LNCS 1019, 1995.

[17] C. Tofts, Some formal musings on the performance of asynchronous hardware, Computer Science Department Report number UMCS-96-2-2, University of Manchester, 1996.

[18] C. Tofts,Compositional Performance Analysis, Proceedings TACAS '97, pp 290-305, LNCS 1217, 1997.

[19] S. F. M. van Vlijmen, A. van Waveren, An Algebraic Specification of a Model Factory, Research report, University of Amsterdam Programming research Group, 1992.

[20] J. Wilkinson, The Numerical Eigenvalue Problem, Oxford University press 1965.