M A S C O T S ' 99: Seventh International Symposium on Modeling, Analysis and Simulation
October 24-28, 1999, College Park Maryland, USA, Copyright IEEE

Evaluating Concurrency Options in Software Specifications

W. Craig Scratchley and C.M. Woodside
Department of Systems and Computer Engineering
Carleton University, Ottawa, ON, CANADA K1S 5B6
e-mail: {scratch, cmw}@sce.carleton.ca
http://www.sce.carleton.ca/rads/rads.html

Abstract

An approach called PERFECT is described which evaluates the feasibility of proposed software concurrency architectures for a set of scenarios and a set of quality-of-service requirements. An evaluation is performed by constructing and simulating a virtual implementation which conforms to the specified behaviour and the specified concurrency architecture. For simulation, the execution of application activities and kernel primitives must be sequenced for each concurrent thread. The approach is successfully demonstrated on specified scenarios for a Group Communication Server.

1. Introduction

It is normal to establish a concurrency architecture before beginning to design software, and this is often done on the basis of inadequate information. A poor choice at this stage will limit the performance of the design. Too many concurrent processes will introduce excessive overhead for process switching, communication, and data protection while too few may introduce important delays due to non-urgent work being completed before urgent work can begin, or due to blocking. This paper considers the design at an early stage, described by just a set of scenarios, and evaluates a designer's proposals for concurrency architectures by constructing virtual implementations.

The intended domain of application is systems with soft real-time deadlines, such as a required percentage of responses completing within a given delay, and with highly variable demands for execution. These are typical of communications software and business data processing. The workload of the software is represented by responsibilities in the scenario, fulfilled by activities which have known or estimated demands for processor time, disk access and so forth. The demands are specified by a distribution and required parameters.

The scenarios are represented by Use Case Maps [2, 4], which represent paths of execution against a background of the software components that execute them. The quality-of-service requirements are annotated on the scenarios.

There are two aspects to introducing concurrency into software, which we will identify with processes and threads. The software can be partitioned into a number of different processes, and each process can be instantiated as a number of threads. We apply the name *process* to the class, and *thread* to the instance. Together, the partitioning and threads make up the concurrency architecture of the software.

Suggestions for architecting concurrency have been given by a number of authors (eg. [6, 7]), including:

- Structure concurrency so that devices can be used concurrently.
- Structure concurrency so that more urgent work can pre-empt less urgent work.
- Make partitions which do not require excessive communication between processes.
- Try to structure concurrency such that all access to an item of data is done from one thread, so the data doesn't need to be shared between threads.

However these suggestions do not lead to solutions without some quantitative comparisons, for instance to compare the change in overhead caused by of an additional process against the benefit of additional concurrency.

This paper describes an approach (and a tool) called PERFECT (the "PERFormance Evaluation by Construction" Tool) which evaluates the feasibility of a given concurrency architecture for achieving a set of quality-of-service requirements. For the present, attention is limited to software running on a single node, possibly a symmetric shared-memory multiprocessor.

2. Overview of the Method

The partitioning and evaluation begins after the processing scenarios have been captured. This does not require a complete set of scenarios, but only those of major importance for the system's performance. In principle any analysis technique could be used to capture the scenarios. Gomaa has used data flow diagrams for this purpose [7], and Smith's execution graphs could also be used [5]. This work uses Use Case Maps because they include a well-defined methodology for describing concurrency, and they have a tool which has been extended to incorporate performance annotations [11, 3].

The method can be summarized as:

Specify paths of execution, with alternatives, loops and parallel fork/joins. Embed activities with demand parameters. Annotate paths with response-time requirements.

Allocate path segments in the specification to processes. Decide whether each process will be single or multi-threaded.

Evaluate by simulating a virtual implementation of the system, which conforms to the specified behaviour and the specified division into concurrent processes. The simulation is performed with a scheduling discipline that assists the system in meeting response-time requirements. Before simulating, parameters of necessary devices such as processor and disks must be specified.

3. The Specification

A substantial example will be used to explain the UCM notation.

Figure 1 shows a specification of a Group Communications Server (GCS) which manages a set of documents. Users subscribe to specific documents, and also may update the documents. When a user updates a document, all subscribers of that document are notified. Users can request that the current version of the document be sent to them.

To explain the notation we will follow the paths for a document being updated, which is the topmost scenario (numbered 1) in Figure 1. Other scenarios are a client sending a new document (2), a client requesting to be subscribed (3) or unsubscribed (4) from a document, and a client wanting to get the most recent version of a document (5).

The Update scenario starts at the filled black circle, called a starting point, named *msgArr*. When a request arrives at the server computer, think of a token being created at the starting point and starting to travel along the path. To get performance predictions we need an estimate of the workload intensity, and so we have specified that requests arrive according to a Poisson process with an average inter-arrival time of 4 ms.

The first processing on the request is specified by activity *recMsg* which is shared by all five scenarios. Activities are represented here by small diamond shapes (known in UCMs as stubs, and also capable of representing subgraphs), and their execution demands are entered into the tool through a special parameter window. At activity *recMsg* the request-type and document name are extracted from the request.

Next on the path is an OR-fork. Only one of the branches will be followed. The numbers beside the branches indicate that for every 2 send-document requests, on average there are 1 subscribe, 1 unsubscribe, and 50 get-document requests. The Update scenario follows the top branch.

At activity *recFile*, the document text is extracted from the request. Next is activity *getDocInf*, which reads from a datastore. The datastore is a dictionary named *docMap*, and read-only access is indicated by a dashed arrow drawn from the datastore to the activity. Write access would be shown by a dashed arrow in the reverse direction. Using the document name as a key, *docMap* is queried to see if the server has already stored a previous version of the document.

Assuming that an earlier version of the document is already stored on the server, the top branch is followed which takes us to an AND-fork which initiates a set of two parallel sub-paths. One branch of the AND-fork is for writing the document to one of two disks. The other branch is for notifying subscribers that the document has been updated. To notify the subscribers,

- first a notification message is prepared.
- then a temporary copy of the subscribers list is made. The subscribers list is part of a structure called *docInfo* which is maintained for each document.
- then the server loops through the copy of the distribution list. In each loop iteration, the address of a different subscriber is extracted from the copied list and the path is forked for the sending of the notification message to the extracted subscriber. Forking the path in this way allows the sending of notification messages to be done potentially in parallel.

When the path exits from the loop, it joins with the disk-writing sub-path, and a "success" message is returned to the document-sending client. Error conditions are not explicitly considered in this case study, but an error message would be returned instead of a success message should an error arise.

A response-time requirement can be specified between any two scenario-connected timestamp points, which are marked by filled triangular arrowheads along a path and can be used to instrument a simulation or design. When a message arrives at the server computer, consider that a token is created and immediately passes timestamp point *Tarr* where it is stamped with the current (i.e. arrival) time. When a token crosses the timestamp point at the end of a path, such

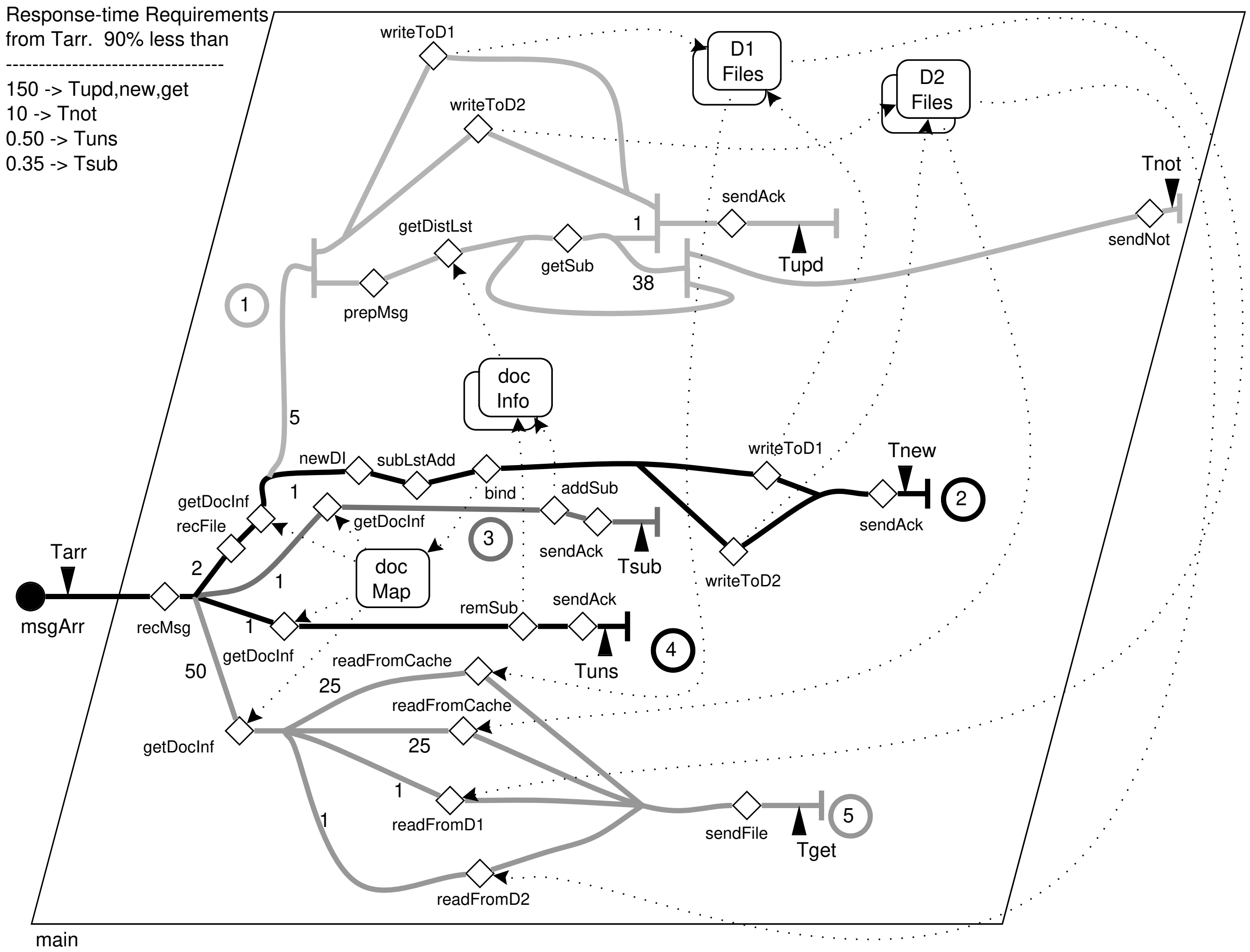

Figure 1. Use Case Map for Group Communication Server with a single-threaded process (Case 1)

as *Tupd*, the response time for that response is determined (named *Rupd* in this case). The response-time requirements in this example all specify that 90% of responses be completed within a specified delay. For test purposes the delays ending at *Tupd*, *Tnew*, and *Tget* all have requirements of 150ms, with values of 10 ms for *Tnot*, 0.35 ms for *Tsub* and 0.50 ms for *Tuns*.

The execution environment, not shown in the Figure, consists of a processor and two disks called *disk1* and *disk2*.

4. Partitioning of path elements

The partitioning of the execution between processes is indicated by drawing the processes in the background. Each path element in a UCM belongs to the process drawn behind it. A single-threaded process is indicated by a parallelogram, as in Figure 1, and a multi-threaded process by a stack of parallelograms. Figure 2 shows a partitioning of the path elements in the GCS application into two multi-threaded processes. The goal of this partitioning is to allow concurrency in processing the updating of a document: the updated document can be saved to a disk while the document's subscribers are being notified. The goal of the multithreading is to handle multiple requests at one time.

A partitioning implicitly introduces overhead operations. In Figure 2, for each execution of an update scenario, there are messages between processes, context switches between threads, and access control for the shared data objects.

5. Virtual implementation

In the virtual implementation there is a process for each process in the specification. Each process has its own *process* mailbox. Any thread in the process can receive from this mailbox. If an execution path leaving a thread later returns, it will arrive as a message to a *return* mailbox private

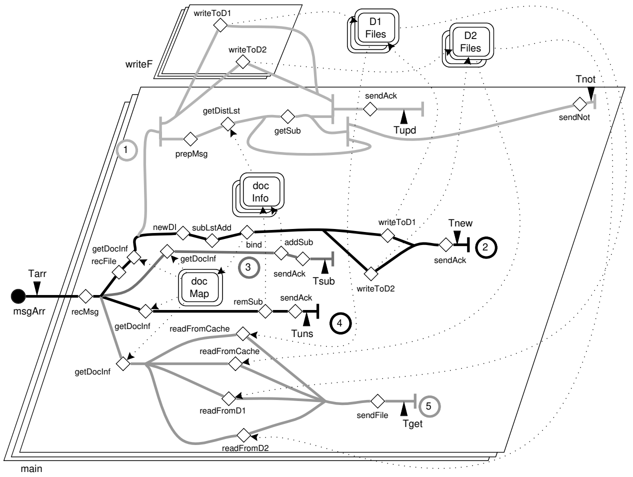

Figure 2. Use Case Map showing Parallelism-in-updating design (Case 4)

to the thread.

5.1. Virtual thread behaviour

The behaviour of a thread is governed by a controller which sequences the execution of activities and the invocation of kernel primitives. It does this by interpreting the UCM specification of the path segments allocated to the process. This control logic takes the role assumed by a finite state machine in some software development methods (eg. ROOM [14]). The state of a thread consists of a set of tokens being processed, each of which knows its location on the UCM subpaths, and the number of tokens which are expected to return to the thread.

Each thread starts by obtaining a token from its *process* mailbox. Stored in the obtained token is its location on a UCM path. The controller moves the token to the next path element and updates the token's location attribute.

When the token arrives at an activity, service at a sequence of devices will be requested. If the activity needs access to data shared between threads, then read or write permission as appropriate will be requested before any other processing is done for the activity. The request may cause the thread to be blocked. To avoid deadlock trying to access shared data, shared data stores will be ordered, and requests will be made according to that order. When processing for the activity is finished, the data stores will be released by the activity.

When a token leaves a process, the thread will send a message referencing the token. If the token will later return to the process then the address of the thread's *return* mailbox is stored in the token. The thread's count of returning tokens should be incremented. If the token has previously visited the process that it is entering, then send the message to the appropriate *return* mailbox. Otherwise, send the token to the *process* mailbox.

When a token arrives at an AND-fork, the original token will be routed to one of the branches, and clones of the token will be created and routed to each other branch.

When a token arrives at an AND-join, one of two things will happen. If at least one other input branch has not yet received a token, then the token should wait. If all other input branches have received a token, then merge into the arriving token the pertinent information, such as *return* mailbox addresses, stored in the waiting tokens and delete the waiting tokens. The arriving token continues on.

5.2. Scheduling Policy

The scheduling of the processor in the simulator was chosen to aid in the evaluation of feasibility of the response requirements. Thus, it should be a "good" scheduler for systems with soft deadlines and random or statistically determined arrivals and execution times. These properties are similar to those of real-time database systems where Earliest Deadline First (EDF) scheduling has been investigated (eg. [1, 16, 8, 10]). All the studies show that EDF scheduling works well at lower loads, allowing most of the responses to complete within their deadlines. We are most interested in these cases.

During periods of overload, EDF suffers from the "domino effect"[15], whereby a response which is already "late" is given the highest priority, thus delaying responses which could otherwise more easily meet their deadlines. To stabilize overload performance, Haritsa, Livny and Carey [8] propose an algorithm called Adaptive Earliest Deadline. In this research, we use pure EDF and find that it works well.

5.3. How the scheduling policy influences desired thread behaviour

With EDF scheduling, one of the thread controller's jobs is to determine its own deadline. The approach taken is that when a token crosses a timestamp point which starts a particular response-time interval, that a deadline will be defined by adding the response-time requirement's delay value to the elapsed execution time. Multiple deadlines can be defined at the same moment if a timestamp point starts multiple response-time intervals (as *Tarr* does). The token stores its collection of deadlines, and the earliest of those deadlines will be identified. Due to the creation of tokens at an AND-Fork (as was described above) the controller for a thread might have to interleave the processing of a number of "ready" tokens. Of the ready tokens, the token with the earliest deadline will become the active token. The current priority of a thread will be assigned based upon the earliest deadline chosen from among the earliest deadline of the active token, the earliest deadline of each token waiting at any AND-Join and served by the thread, the earliest deadline of each token waiting at a mailbox exclusive to the thread, and the earliest deadline of each token waiting for access to data locked by the thread. Thus deadlines can be inherited in order to reduce priority inversion. If no token currently associated with the thread has a deadline, then the thread will be assigned the minimum possible priority.

When a message is sent from a thread to a mailbox, priorities may need to change. If the message is being sent to a mailbox exclusive to a thread, and the earliest deadline of the token referenced in the message is earlier than the current deadline of the receiving thread, then replace the deadline of the thread with the token's earliest deadline. Otherwise the message is being sent to the *process* mailbox of a multi-threaded process and the earliest deadline of the token should be assigned to an idle thread.

After sending a message a thread will have to obtain another token. If the set of ready tokens is not empty, then the thread will take a token with the earliest deadline. Otherwise, the process will receive from a mailbox. If the number of returning tokens is greater than zero the thread will decrement the count and receive from its *return* mailbox. Otherwise the thread will receive from its *process* mailbox.

At an AND-fork it was previously mentioned that a token is cloned. If a deadline belonging to the original token is only relevant to some of the branches, it is removed from the tokens going to the other branches. A token with the earliest deadline will be selected as the active token.

At an OR-fork, deadlines which the token is carrying which are specific to branches other than the one chosen can be discarded. This may result in a change in the token's earliest deadline, and possibly the switching of active tokens and/or the scheduling of another thread. This suggests that OR-forks should sometimes be placed more towards the start of a scenario than is necessary for functional correctness. Such placement may allow a token's earliest deadline to become less urgent earlier, and hence allow a now more urgent token to execute earlier thereby yielding more efficient scheduling. Consider the *getDocInf* activities in the case study. Instead of replicated activities, one instance could have been placed before the OR-fork, but this would delay the deadline change for less-urgent scenarios.

6. Evaluation of the Group Communication Server

For evaluation, the specification is entered into the UCM Navigator tool [11, 3]. Execution times for activities are entered as a distribution and its parameters, and probabilities or ratios are given for OR-forks. The values used in the study were measured from an implementation. The UCM output file is passed to PERFECT which constructs and instruments the virtual implementation, runs a simulation, and collects metrics for the duration of the run. At the end of a run, the metrics are output together with confidence intervals. PERFECT is built on top of the PARASOL simulation engine [12], but other simulation engines could equally be used, such as CSIM [13].

Five concurrency architectures were considered for the Group Communication Server described above, with the results shown in Table 1. The left side of the table gives device utilizations, and breaks out the concurrency-related overhead in the CPU utilization ("ctxsw" for context switching, "msg" for message passing, "adj dl" for adjusting deadlines, and "sem4" for semaphores used in protecting access to data). The right side summarizes the response times by mean value and percent satisfactory. The target for satisfaction is 90% of responses within the specified delay in all cases. The statistics for response times which do not achieve 90% success are shown italicized and in bold face.

Case 1: Single-thread design The first design that will be evaluated is the most simple: all path elements are performed by a single thread (Figure 1). It has the advantage that, because there is only one thread, there is no need to protect shared data and hence no overhead due to semaphore calls.

The PERFECT tool reported that, although the device utilizations were below 40% and overhead was insignificant, the response times were growing larger and larger as the simulation run continued. The system is unstable because the average thread service time is too long. If we add the 3 device utilizations together, the result is 100% showing that the thread never waits at the *process* mailbox.

Case 2: Thread-per-disk design To allow the multiple devices to be used simultaneously, consider the design shown in Figure 3 with a number of single-threaded processes. A process *D1* handles all the activities which access *disk1* and a process *D2* handles all the activities which access *disk2*. In Figure 3 the same process D1 has been shown in three separate locations, to lie behind activities in different sub-paths. D2 is treated the same way. Process *notify* handles the matching AND-Fork/AND-Join elements and all non-disk-related activities after the AND-Fork. The AND-Fork and AND-Join must be kept in the same process in order to join matching tokens. After *notify* sends a message to either *D1* or *D2*, it waits at its *return* mailbox for a message to return from *D1* or *D2*. Once waiting, no other work can be done by the waiting thread until a message arrives at its *return* mailbox. This is why *notify* was created. Had the elements executed by *notify* been left in process *main*, the *main* thread would have been unable to

Case num	Case name	Utilizations (%)							Response times (mean time in ms / % meeting requirement)					
		Disk1	Disk2	CPU					Rupd (dv=150)	Rnot (dv=10)	Rnew (dv=150)	Rsub (dv=0.35)	Runs (dv=0.50)	Rget (dv=150)
				Total	ctxsw oh	msg oh	adj dl oh	sem4 oh						
1	Single Thread	38	37	24.8	0	1.3	0.63	0	***infinite/0***	***infinite/0***	***infinite/0***	***infinite/0***	***infinite/0***	***infinite/0***
2	Thr per Disk	48	49	38.4	3.3	3.0	2.8	0.20	***166/55***	***88/35***	***96/86***	***6./54***	***6./57***	14/99.2
3a	SMTP	49	49	35.9	2.8	1.46	2.2	0.41	93/90.3	2.3/99.3	91/90	0.314/95	0.347/95	7.5/99.3
3b	- double subscribers	49	49	70.9	6.76	2.80	4.3	0.77	***98/88***	4.5/90	94/90	***0.34/89***	0.38/90	8.3/99.5
4a	Parallelism in Updating	49	49	36.3	2.98	1.55	2.27	0.41	92.6/90.7	2.4/99.0	91/91	0.315/94.4	0.346/95	7.5/99.3
4b	- /w interm. deadline	49	48	36.3	3.03	1.54	2.3	0.41	89/91.6	3.4/98.6	90/92	0.319/93	0.348/95	7.4/99.3
4c	- double subscribers	49	49	71.2	6.96	2.88	4.4	0.77	91/90.9	***5.7/86***	94/90	***0.35/87***	0.39/90	8.3/99.5
5a	Maximum parallelism	49	49	41.3	5.5	3.2	3.1	0.41	91/90.4	***14.6/40***	94/90	0.58/90	0.6/93	8.9/99.2
5b	- with 2 processors	49	49	28.6+11.9	4.8 total	3.2 total	3.1 total	0.41 total	89/91.2	***6.0/84***	89/92	0.33/98.9	0.34/99.5	7.0/99.3
5c	- with 4 processors	49	49	26.3+9.3 +3.3+1.4	4.6 total	3.2 total	3.1 total	0.41 total	89.3/91.0	4.0/93.8	8.9/92	0.306/99.6	0.327/100.0	6.8/99.3

Table 1. Output reported by PERFECT tool for GCS designs

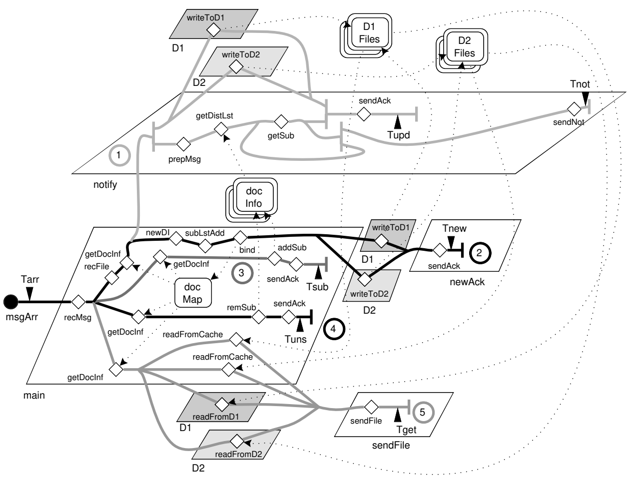

Figure 3. Use Case Map showing Thread-per-disk design (Case 2)

process other arriving client messages while waiting for a message from *D1* or *D2*. Processes *newAck* and *sendFile* were created for the same reason.

Table 1 shows that this design is much better than Case 1, the Single-thread design. Now the server is able to keep up with the arriving requests. However, only response-time requirement *Rget* is being met: over 99% of *Rget* responses take less than 150ms. Response *Rnew* is close to being met with an 86% success rate, but response *Rupd* does much more poorly with only 55% of responses less than 150ms. One reason for this is that messages might have to queue at the *process* mailbox for *notify* while *notify* is waiting on its *return* mailbox. Note that responses *Rsub* and *Runs* are relatively far from the requirements. This can be explained by considering that the processing of a subscription request, for example, cannot pre-empt the execution of an activity like *readFromCache*. Compounding this, *readFromCache* is executed relatively often. This design has significantly more overhead than the single-thread design: context switching and semaphore overhead are now present, and the messaging overhead and deadline adjustment overhead have significantly increased. Thus it is certainly not true that increased overhead means a poorer-performing design. Semaphore overhead is now present because protection has been introduced so that an individual file cannot have active readers or other writers while it is being written to. Also, the management data for a particular document, including for example its list of subscribers, is protected so that reading and updating are done mutually exclusively.

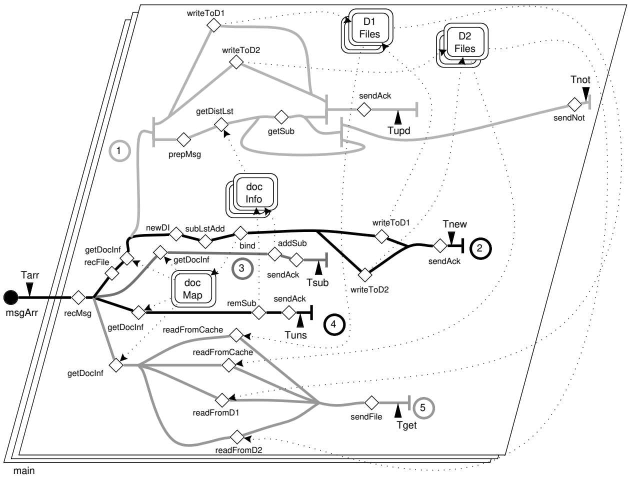

Figure 4. Use Case Map showing design with single multithreaded process (Case 3)

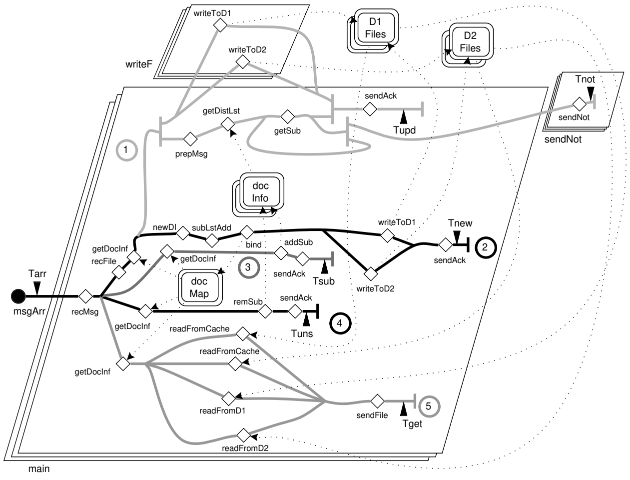

Figure 5. Use Case Map showing design with maximum parallelism (Case 5)

Case 3: Single multi-threaded process (SMTP) An alternative improvement to the single-thread design is to create a single multi-threaded process (Figure 4). This solves two problems. First, while one thread is using one device, another thread can be using another device. Second, more urgent processing can pre-empt less urgent processing. The results for this design are quite encouraging: every response-time requirement is now being met. Also, although the overhead is greater than for the single-thread design, the total overhead is less than it was for the thread-per-disk design of Figure 3. The only category of overhead which has increased beyond that for the thread-per-disk design is semaphore overhead, because now *docMap*, the dictionary which indexes the data object for each document, must also be protected.

Case 4: Parallelism-in-updating design In the SMTP design above, a single thread serializes the potentially parallel subpaths when a document is updated: writing a file to disk; and generating, addressing, and sending notification messages to subscribers. In Figure 2, a second multi-threaded process was created to manage writing to the disk when a document is updated.

The results for this design are similar to the previous SMTP design, although all overhead except that for semaphores has slightly increased. We might have expected that response *Rupd* would be significantly faster because of the allowed device concurrency. However, since *Rnot* has a more urgent requirement than *Rupd*, all the processing of activities *prepMsg*, *getDistList*, *getASub*, and *sendUpd* for a particular response would have to be finished before the associated activity in *writeF* was started.

To allow a given response to use the multiple devices concurrently, an "intermediate" response-time requirement on the path in *writeF* was introduced in Case 4b, to make the deadline for *writeF* earlier. To achieve the desired effect, the delay value must be less than that for *Rnot*. However, the smaller the delay value is, the more the requesting of disk writing can negatively impact the response-times of other paths. The delay value was made smaller but as close to that for *Rnot* as possible.

With the intermediate deadline, response *Rupd* is less than 150ms a greater fraction of the time. Note also that as one might expect, the percentage of successes for *Rnot* has decreased slightly. There was no significant change in overhead following the introduction of the intermediate deadline.

Case 5: Maximal-parallelism design Is there any remaining concurrency in the GCS specification that has not yet been exploited? Only the sending of notification messages to subscribers, which is currently being serialized for a document being updated, remains as unexploited concurrency. Figure 5 shows a design based on the Parallelism-in-updating design but with a third multi-threaded process created to send out notification messages, one thread per message.

The performance results of this Maximal-parallelism design are relatively poor: the overhead has significantly increased and the response times have worsened over the previous two designs. There is no real parallelism with just one processor, and there is a lot of overhead in triggering a thread for every notification. This hurts some subscription

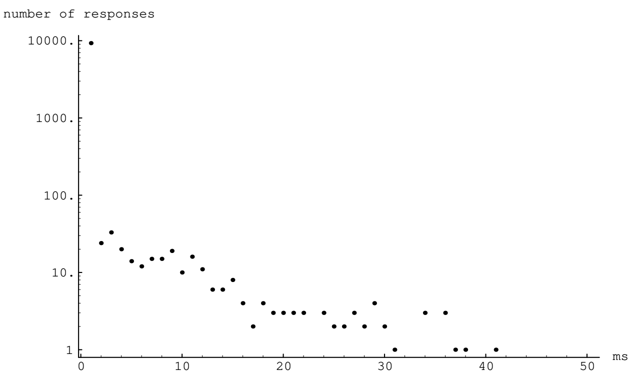

Figure 6. Frequency distribution of measured response times for *Rsub*

requests. Although *Rsub* is within its specification, it has a high variance. Notice the interesting statistics reported for response *Rsub*: the mean response time is 1.66 times the response-time requirement's time value, and yet 90% of responses are less than the time value. Figure 6 shows a frequency distribution of some measured response-times for response *Rsub* when using this design. Although the vast majority of responses take less than 1 ms, response-times up to and even exceeding 40 ms were observed. What seems to be happening is that when a document with many subscribers is updated, the sending of notification messages becomes very urgent ("I want it done yesterday!") and prevents subscription requests from being processed.

Multiple processors were considered in Cases 5b (two) and 5c (four). The response times improve as more processors are used. The response times for responses *Rsub* and *Runs* are significantly better than for any tested single-processor design, with 99% or more of the responses within the specified time value. However, the response-times of sending update messages, even with 4 processors, do not compare favourably with those achieved using the SMTP or Parallelism-in-updating designs on a single processor. This is not surprising as the thread iterating through the subscriber list incurs a greater cost in overhead by communicating with the *sendNot* process than it would simply by executing the *sendNot* activity.

Discussion of the five architectures Concurrency is found to be necessary, despite its higher overhead. The SMTP design and the parallelism-in-updating design are the only ones which meet the response-time requirements. Of these two, the parallelism-in-updating design is the better because its success rates are generally higher.

Adding an intermediate response-time requirement in Case 4b improved the performance. The best value of the intermediate deadline can be explored with experiments using PERFECT.

Sensitivity It is also useful to explore the impact of a change in the workload parameters (execution times, choices, hardware etc.) To demonstrate how the tool can help in this matter, the two best designs were tested with modified workload assumptions. The frequency of documents being sent to the server was kept constant, but the average number of subscribers per document was doubled, with a corresponding doubling of the number of get-document requests, giving Cases 3b and 4c.

The results in Table 1 show that both designs now just miss meeting the requirements. The utilization of the processor rose to over 70% in both cases. The SMTP design only achieved an 88% success ratio for *Rupd* and 89% for *Rsub*. The increased number of subscribers for documents has significantly increased the processor demand for sending notification messages to subscribers when a document is updated. This results in a reduction of the success percentage for response *Rnot*, down from 99.3% to 90%. The parallelism-in-updating design has a better success percentage for *Rupd* (90.9%), at the expense of response *Rnot* (now only 86%), and *Rsub* (now 87%).

Thus, the given system has the reserve capacity to nearly meet the response-time requirements under a significant increase in workload.

7. Conclusions

The virtual implementation technique used in PERFECT succeeds in simulating rather complex systems, as illustrated by the example. Process partitioning options can be easily generated. In principle the technique generalizes to complex specifications with hundreds of activities: UCMs support abstraction and the definition of subsystems via stubs [2, 4, 11], and the virtual implementation has no intrinsic limitations.

The evaluation made by the approach described here indicates the feasibility of the response-time requirements within a given concurrency architecture. The evaluation exposes the factors governing performance. When a demonstrably good architecture is obtained it can be used as the basis of the design of the processes. The evaluation depends on using the EDF scheduler, and if the target system uses a different scheduler then further analysis must be done later.

The technique could be used with budgeted demands [9] in place of estimated demands, to allocate execution budgets to activities, as a guideline for developers. A possible process would be:

- From the requirements, estimate demand budgets for activities not yet coded.

- Use the technique to find a feasible process partitioning with these budgets and measured demands for overhead and finished components.
- Allow developers to code the new activities trying to keep resource demands within budget.
- As the development proceeds, iterate to adjust budgets as needed while remaining feasible.

Used in this way, PERFECT would break requirements down into targets for separate groups of developers, taking into account the implementation platform and the contention delays.

References

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[2] R. Buhr. Use case maps as architectural entities for complex systems. *Transactions on Software Engineering*, 24(12):1131–1155, December 1998.

[3] R. Buhr. Making behaviour a concrete architectural concept. In *Proc. 32nd Annual Hawaii International Conference on System Sciences*, January 1999.

[4] R. Buhr and R. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice Hall, 1996. 302 pages.

[5] C.U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[6] D. deChampeaux, D. Lea, and P. Faure. *Object-Oriented System Development*. Addison-Wesley, 1993.

[7] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. The SEI Series in Software Engineering. Addison-Wesley, 1993.

[8] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *Proc. Real-Time Systems Symposium*, pages 232–242, San Antonio, Texas, Dec. 1991. IEEE Computer Society Press.

[9] M. Hasselgrave. Avoiding the software performance crisis. In *Proc. of the First International Workshop on Software and Performance*, pages 78–79, Santa Fe, New Mexico, October 1998. Association for Computing Machinery.

[10] J. Huang and J. A. Stankovic. Experimental evaluation of real-time transaction processing. In *Proc. Real-Time Systems Symposium*, pages 144–153, Santa Monica, California, Dec. 1989. IEEE Computer Society Press.

[11] A. Miga. Application of use case maps to system design with tool support. Master's thesis, Dept. Systems and Computer Engineering, Carleton University, Ottawa, CANADA, 1998.

[12] J. E. Neilson. Parasol: A simulator for distributed and/or parallel systems. Technical Report TR-192, School of computer Science, Carleton University, May 1991.

[13] H. Schwetman. CSIM17: A simulation model-building toolkit. In J. Tew, S. Manivannan, D. Sadowski, and A. Seila, editors, *Proc. 1994 Winter Simulation Conference*, pages 464–470, Orlando, 1994.

[14] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[15] J. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[16] P. S. Yu, K.-L. Wu, K.-J. Lin, and S. H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.