Evaluation and Diagnosis of Concurrency Architectures

by

W. Craig Scratchley, B.A.Sc.

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

July 2000

©2000, W. Craig Scratchley

The undersigned recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis,

"Evaluation and Diagnosis of Concurrency Architectures"

submitted by

W. Craig Scratchley, B.A.Sc.

in partial fulfillment of the requirements for

the degree of Doctor of Philosophy

Chair, Department of Systems and Computer Engineering

Professor C. Murray Woodside, Thesis Supervisor

Dr. Connie U. Smith, External Examiner

Carleton University

July 2000

Abstract

Because it affects such things as the degree to which logical concurrency can be realized by physically concurrent devices, the concurrency architecture of software can have a large impact on performance. Architectural decisions such as concurrency are typically made early in the development of software, and are difficult to change once the software has begun to be fleshed out. It is therefore useful to be able to evaluate the feasibility of software concurrency architectures proposed for a set of scenarios and a set of response-time requirements which, in part, define a particular software project. Such evaluation, however, is difficult because it is hard to predict the effects of pre-emption, the amount of concurrency-related overhead generated, and which portions of scenarios will actually be performed in parallel and which will for some reason end up serialized. So that one can discover possible ways to improve a concurrency architecture, it is also useful to diagnose an architecture for causes of performance shortfalls.

This thesis describes an approach (and a tool) called PERFECT which performs such evaluations and diagnoses by automatically constructing and simulating an instrumented virtual implementation which conforms to a behaviour and concurrency architecture easily specified by supplying a Use Case Map supplemented with performance information. This work focuses on software for servers.

An evaluation reports, for each type of response, the fraction of responses which finish within the specified delay time. Also reported is the utilization of each device. Diagnoses include measurements of the cumulative effect of priority inversion at each point where messages are sent to a software process, lost opportunities to concurrently use multiple devices by the most urgent work, and the amounts of different types of concurrency-related overhead such as context switching.

To demonstrate how the whole approach works, a number of cycles of proposing, evaluating, and diagnosing a concurrency architecture for a group communication server are presented. The first architecture cannot process requests as fast as they arrive. By remedying the diagnosed causes of performance shortfalls, an architecture is derived which meets all performance requirements. Adding concurrency not suggested by diagnoses causes performance to decline.

Acknowledgements

Above everyone else, I acknowledge my thesis supervisor, Professor Murray Woodside, for his dedication, time, and thoroughness.

A lot of thanks goes to the staff of the Department of Systems and Computer Engineering, who over the years have facilitated my progress

I thank the creator of PARAS0L, Professor John Neilson, and those including Greg Franks and Pat Morin who have worked on the library.

I thank Professor Ray Buhr for consulting with me on his Use Case Maps notation, and Andrew Miga for quickly responding to issues regarding UCM Navigator.

I acknowledge NeXT Software Inc., for designing a great development environment, and providing it to the University at cost. I also thank the Free Software Foundation and all its volunteers for providing so many of the tools and libraries that were used in this research.

Financial assistance was provided by the National Science and Engineering Research Council of Canada (NSERC), the Telecommunication Research Institute of Ontario (TRIO), Nortel Networks, and Carleton University.

Finally, I thank my family: Tatiana, Linda, Ted and Shirley Scratchley for their love.

Contents

1 Introduction **1**

1.1 Concurrency in single-node systems 1

1.1.1 Characteristics of applications 2

1.1.2 Specification by scenarios 3

1.1.3 Concept of concurrency within an application 3

1.1.4 How concurrency in applications is supported 4

1.2 Value of concurrent threads in an application 4

1.3 Costs of concurrent threads 5

1.4 Concurrency architectures for single-node systems 6

1.5 Evaluation and diagnosis 7

1.6 Class of systems considered 8

1.7 Contributions 8

1.8 Organization of thesis 10

2 Background **11**

2.1 Specification of system behaviour 11

2.1.1 Written text for scenarios 12

2.1.2 Message sequence charts 12

2.1.3 Extended state machines 12

2.1.4 Process algebras 13

2.1.5 Graphical specification of scenarios 13

2.2 Partitioning activities into processes 15

2.3 Scheduling in soft real-time systems 18

2.3.1 Static priority scheduling 18

2.3.2 Priority inheritance 19

2.3.3 Earliest Deadline First 19

2.3.4 Subtask deadline assignment 20

2.3.5 Scheduling disk requests with deadlines 21

2.4 Diagnosing designs for performance problems 21

2.5 Realizing potential parallelism in massively parallel processing 22

2.6 Software architecture 23

3 Use Case Maps with Performance Annotations 24

3.1 Basic UCM notation for specifying scenarios 24

3.2 Performance extensions to the basic scenario notation 26

3.2.1 Tokens and workload intensity 27

3.2.2 Activities, their references to data objects, and their service requirements 27

3.2.3 Branch selection criteria at OR forks 29

3.2.4 Timestamp points and response-time requirements 30

3.3 Partitioning of path elements 31

3.4 Incorporation of performance annotations into UCM Navigator 33

4 Virtual Implementation to Evaluate Architecture 36

4.1 Processes, threads, and mailboxes 37

4.2 Thread behaviour in the virtual implementation 37

4.2.1 Behaviour at an activity 38
4.2.2 Behaviour when forking 38
4.2.3 Behaviour when a token leaves a process 39
4.2.4 Behaviour at an AND join 41
4.3 Scheduling policy 42
4.4 How the scheduling policy influences desired thread behaviour 42
4.4.1 Defining deadlines for a token 43
4.4.2 AND fork 43
4.4.3 Current priority of a thread 44
4.4.4 AND join 45
4.4.5 Sending messages 46
4.4.6 Scheduling influences at OR forks 46
4.4.7 Summary of when a thread's deadline may need to change 49
4.5 Scheduling of devices other than central processor(s) 50
4.6 Details of the simulator 50
4.7 Simple validation of the simulator 54
4.7.1 A M/M/1 queueing system 54
4.7.2 A simple product-form queueing network 56
4.7.3 Validating pre-emption and overhead 58

5 Diagnostic Metrics 60

5.1 Problem 1: inversion due to inadequate concurrency 60
5.2 Problem 2: inefficient scheduling of devices due to inadequate concurrency . 63
5.3 Problem 3: excessive overhead due to inappropriate concurrency 65
5.4 Formal metrics for detecting problems 66
5.4.1 Inversion metric 66
5.4.2 External device metric 70

5.4.3 Internal device metric 73
5.5 Using the metrics to guide design improvement 75
5.5.1 Inversion examples 76
5.5.2 External device metric examples 80
5.5.3 Internal device metric examples 82
5.5.4 Example of metric for a path segment returning to a process 85
5.6 Discussion of the metrics 90

6 Case Study 92

6.1 Description of the Group Communication Server application 92
6.1.1 Scenarios 1 and 2: updating and submitting new documents 94
6.1.2 Scenario 3: subscribing to a document 96
6.1.3 Scenario 5: retrieving the most recent version of a document 97
6.2 Performance numbers 97
6.3 Concurrency architectures for the Group Communication Server 98
6.3.1 Case 1: Single-thread architecture 98
6.3.2 Case 2: Six-thread architecture 103
6.3.3 Case 3: Single Multi-Threaded Process (SMTP) architecture 109
6.3.4 Case 4: Parallelism-in-updating architecture 111
6.3.5 Case 5: Maximal-parallelism architecture 114
6.4 Impact of workload change 117

7 Conclusions 126

7.1 The overall view 126
7.2 Individual contributions 127
7.2.1 Extensions to the Use Case Map notation 127
7.2.2 Virtual implementation 128

7.2.3 Metrics to detect concurrency problems 129
7.2.4 The PERFECT tool 130
7.2.5 Case study 131
7.3 Straightforward generalizations 131
7.4 Future Research 132
7.4.1 Validation on a system supporting EDF scheduling 132
7.4.2 Displaying results in the UCM Navigator 132
7.4.3 Automatic identification of efficient concurrency architectures 133

List of Figures

3.1 UCM scenarios with basic notation indicated 25

3.2 UCM scenarios with performance extensions indicated 26

3.3 A possible plugin for activity h 28

3.4 A partitioning of the example UCM, with process types labelled in red, and overhead operations labelled in blue. 32

3.5 A different partitioning of the example UCM 34

4.1 A UCM showing scenarios and a concurrency architecture 39

4.2 The UCM of Figure 4.1 with the OR fork moved 47

4.3 A UCM corresponding to an M/M/1 queuing system 55

4.4 A simple product form queueing network 57

4.5 A UCM corresponding to the product form queueing network of Figure 4.4 57

4.6 A UCM with equally spaced arrivals and a timeline of its execution. Note the pre-emption which occurs. 58

5.1 Example showing inversion 62

5.2 Excess serialization due to only one token being able to travel along a given path segment at a given time 63

5.3 Excess serialization after an AND fork 64

5.4 A very simple UCM 65

5.5 The very simple UCM above with a second single-threaded process added . 65
5.6 Suspected deadline inversion where a path returns to a process 68
5.7 Inversion metric applied to the Use Case Map of Figure 5.1 70
5.8 External device metric applied to the Use Case Map of Figure 5.2 73
5.9 Internal device metric applied to the Use Case Map of Figure 5.3 75
5.10 Case INV2 – An architecture which removes inversion 77
5.11 Case INV3 – UCM of Figure 5.7 with changed time parameters 78
5.12 Case INV4 – Removing inversion in Figure 5.11 makes things worse 79
5.13 Case EXT2 – An architecture which increases concurrent use of devices compared with the architecture of Figure 5.8 80
5.14 Case EXT3 – An architecture which further increases concurrent use of devices 82
5.15 Case INT2 – An architecture which allows concurrent processing of AND fork branches 83
5.16 Case INT3 – An improvement to the architecture of the previous figure . . 84
5.17 Case INT4 – Results with an added intermediate deadline to tune the scheduling 85
5.18 Case RET1 – A UCM with tokens returning to a process 86
5.19 Case RET2 – The external device metric reports a problem on segment between *c*2 and *c*3 87
5.20 Case RET3 – An architecture which eliminates PPS device need where tokens return to process *stp1* 88
5.21 Case RET4 – The external inversion metric reports a problem on the segment between *c*2 and *c*3 89
5.22 Case RET5 – An architecture which eliminates inversion when tokens return to process *stp1* 90

6.1 The scenarios in the GCS application 93

6.2 Use Case Map for Group Communication Server with a single-threaded process (Case 1) 99
6.3 Evaluation of Single-thread architecture 100
6.4 Use Case Map showing Six-thread architecture (Case 2) 104
6.5 Evaluation of Six-thread architecture 105
6.6 Use Case Map showing architecture with single multithreaded process and its evaluation (Case 3a) 110
6.7 Use Case Map showing Parallelism-in-updating architecture and its evaluation (Case 4a) 112
6.8 Evaluation of Maximal-parallelism architecture (Case 5a) 115
6.9 Frequency distribution of measured response times for *Rsub* in Case 5a . . . 116
6.10 Evaluation of SMTP design with doubled subscribers (Case 3b) 118
6.11 Evaluation of Parallelism-in-updating architecture with doubled subscribers (Case 4b) 119
6.12 Evaluation of SMTP architecture with doubled subscribers and two processors (Case 3c) 123
6.13 Evaluation of SMTP architecture with doubled subscribers and double-speed processor (Case 3d) 124
6.14 Evaluation of Parallelism-in-updating architecture with doubled subscribers and double-speed processor (Case 4d) 125

List of Tables

4.1 Processor overhead for kernel primitives using Solaris on a 143MHz ultrasparc 53

4.2 Comparisons between analysis and results of PERFECT simulation for the UCM of Figure 4.3 55

4.3 Comparisons between analysis and results of PERFECT simulation for the UCM of Figure 4.5 57

6.1 Output reported by PERFECT tool for GCS architectures 102

6.2 Output reported by PERFECT tool for sub-cases of SMTP architecture . . 121

Chapter 1

Introduction

This thesis describes a way to evaluate and diagnose the performance of concurrency architectures for software running on what we call single-node server systems. The class of single node systems includes many kinds of network servers, such as media servers, web servers, intelligent network (IN) servers, and corporate data servers. This chapter starts by defining what is meant by concurrency in single-node systems, and then goes on to define concurrency architectures and introduce how they can be evaluated and diagnosed.

1.1 Concurrency in single-node systems

For purposes of this thesis, a single-node system will be defined as a computer with one memory space. The computer will likely have multiple devices which can access the memory. These devices will definitely include one or more general-purpose processors, and other devices could include special-purpose processors such as digital signal processors (DSPs) and input/output (I/O) devices such as disks.

In these systems, each of the devices can potentially be working in parallel. A disk can be retrieving data while a DSP is compressing audio and while a general-purpose processor is encrypting a data packet. This is termed physical concurrency.

1.1.1 Characteristics of applications

There are several classes of applications which run on single-node systems. The class of applications focussed on in this thesis is server applications, such as a web server. In such applications there may be different types of requests arriving from many clients. Each request will result in one or more system responses. It is assumed that each type of response has a specified delay within which nearly all responses of that type should be completed. The goal is that each response should finish within its specified delay, and hence it is not necessary that each response be treated “fairly”. For example, a web server might accept requests which result in a voice response starting to stream or a large graphic bitmap being sent. It may be more urgent to begin streaming a voice response than to finish sending a large bit-map which was started much earlier.

Other applications may also benefit from the methods described in this thesis. A second class of applications is interactive applications – those where a single user interacts with the application. These include applications such as word processors and clients which access database servers. These applications have typically made limited internal use of concurrency.

A third class of applications is data processing applications, often amenable to batch processing. With these applications, the goal is to complete a number of usually large computational jobs in a reasonable amount of time. The time it takes to complete one part of one of the jobs is not directly important. In these systems it is often possible to run multiple applications simultaneously and hence get concurrency between applications, and so concurrency within an application is often not so important. While one application is using one device such as a disk, another application can be using another device such as the CPU. This physical concurrency between logically concurrent applications reduces the amount of time it takes to complete all the jobs.

1.1.2 Specification by scenarios

Applications will be specified by describing a set of scenarios. Each scenario describes the processing that needs to occur following the occurrence of a particular type of external event and under certain conditions. Examples of external events that may trigger a scenario include the arrival of a request message or a timer expiring. An example of a condition that may have to be satisfied for a certain scenario is that a certain buffer must be full. The processing for a scenario is described by linking activities, each of which defines a chunk of processing. Examples of activities are retrieving a certain record from a database, processing a multimedia sample, and logging some report in a file. More complex paths may fork, join and synchronize.

A response-time requirement can be specified between any two points on a scenario, and consists of a minimum percentage of responses which must complete within a specified delay value.

1.1.3 Concept of concurrency within an application

Concurrency can occur within an application in two forms: concurrency within a single execution of a scenario, and concurrency between executions of one or more scenarios (there can be concurrency between multiple executions of a single scenario). Concurrency at the scenario level is an example of logical concurrency. Under certain conditions, this logical concurrency can be realized by physically concurrent devices.

To assist in discussing logical concurrency, the notion of tokens will be introduced. When an external event occurs, something called a token will be created which will traverse the path and trigger the activities in sequence. When an activity finishes executing and there is no activity in sequence after it, the token at that activity is destroyed. That is, a token serves to mark the progress of execution along a scenario instance.

Where a scenario path forks to initiate activities in parallel, extra tokens will be created

to traverse the parallel paths.

Sometimes before the processing of one external event has finished, one or more other external events occur. In many cases, from the point of view of the application, these events can be processed concurrently. In terms of tokens, one or more tokens resulting from one external event are still active when another external event generates another token. In principle, all the tokens could be executing simultaneously.[1]

At some instant, the collection of tokens for an executing application indicates all the potential concurrency at that instant.

1.1.4 How concurrency in applications is supported

It was explained above that an application can have logical concurrency, as can be expressed by tokens. How is this logical concurrency supported when the application is executed?

To support logical concurrency, our server application will employ multiple kernel threads during its execution. Each kernel thread is a virtual machine, and the kernel threads queue for devices. Multiple devices can be active concurrently, and hence multiple kernel threads can be making progress concurrently.

In some situations, a single kernel thread may implement the execution paths of several tokens in some way. That is, the execution of tokens can (if desired) be interleaved by the kernel thread. The decision about where to provide separate kernel threads determines to a large part the concurrency architecture.

1.2 Value of concurrent threads in an application

Using concurrent threads gives both advantages and disadvantages for performance. Among the advantages are:

[1] Unless one token in a critical section blocks one or more other tokens.

- It allows logical concurrency to be realized as physical concurrency, enabling a set of responses to finish more quickly. While one thread is using one device, another thread can be using another device. Also, when a thread using a processor becomes blocked for some reason, the processor can still continue being productive by executing another thread.

- It enables pre-emption based on urgency. Because each thread is scheduled separately by the kernel, an urgent thread when enabled can easily pre-empt a less-urgent thread. This provides a mechanism for concentrating system resources on achieving response deadlines that are in jeopardy.

1.3 Costs of concurrent threads

In choosing where to use concurrency in a design, the costs of concurrency must be considered. These costs include:

- Context switching. There is a processing overhead when switching from one thread to another.

- Messaging. When the flow of processing activities along a scenario instance passes from one thread to another, a message will have to be sent. The message provides a pointer to the token representing the processing.

- Protecting shared data objects. With pre-emption enabled, a thread may be pre-empted while it is in the midst of accessing a shared data object. If the activities which access the data object are not all executed by a single thread, and if at least one of those accesses can modify the state of the data object, then some form of protection will be needed so that the object will always be left in a consistent state and values obtained from it will be sensible. Any way of ensuring such consistency, such as using

semaphores in a pessimistic scheme, will have processing overhead associate with it.

- Memory. For each thread that is created, a chunk of memory is allocated to store information such as the thread's stack. Communication structures such as mailboxes and objects such as semaphores used for protecting shared data objects also consume memory. Memory usage can especially be a concern on legacy embedded systems where memory is often limited.

1.4 Concurrency architectures for single-node systems

This thesis will define a concurrency architecture for single-node systems by three characteristics:

- A relation between activities[2] and threads.
- A policy for maintaining the consistency of each shared data object.
- The style of messaging for every place where a path crosses between threads.

The method used in this thesis to define the relation between activities and threads is to first divide the activities in the application into groups and then choose how many threads can be created for each group. The thread or collection of threads available for a group will comprise a process, single-threaded or multi-threaded respectively.

Policies for guaranteeing that operations on a shared data object return sensible values and leave the object in a consistent state can either be optimistic, pessimistic, or a combination of the two. In this thesis we analyze pessimistic methods. Pessimistic methods can either allow multiple readers to access the data object simultaneously, or insist that strictly one operation be allowed to proceed at a time.

[2]And other elements along paths which will be defined in Chapter 3

The third area defined by the concurrency architecture is style of messaging. Possible styles include synchronous messaging where a message is stored with the sending thread until it is ready to be received, and asynchronous messaging where mailboxes accessible to the receiving thread are created to store messages. We will restrict attention to asynchronous messaging in this thesis. More complex interactions can be built up from asynchronous messages.

1.5 Evaluation and diagnosis

This thesis has three goals: first, to define the concepts of concurrency architecture in an operational way, second to evaluate a concurrency architecture for its performance, and third to diagnose causes of performance shortfalls and suggest improvements.

In order to choose a concurrency architecture which balances the costs and benefits of concurrency it is necessary to be able to quantitatively compare different concurrency architectures. In this thesis, we present a method for making such comparisons. For instance, we may compare the change in overhead caused by an additional process against the benefit of additional concurrency.

A concurrency architecture is ultimately judged by the degree to which the specified response-time requirements are met on the target hardware platform, so measuring the fraction of each type of response which meets the specified delay is important. The utilization of each device is also useful, as it helps us understand whether the target hardware platform comprises adequate resources.

Diagnosis identifies defects in concurrency when performance goals are not met. In this work, diagnosis is based on measurements made on an execution of each architecture, including:

1. The amounts of different types of overhead are measured: context switching, messaging, priority adjustment, and data protection.

2. Deadline inversion is identified at every point where a path crosses into a process, and a cumulative measure of its effect is computed.

3. Lost opportunities to concurrently use multiple devices by the most urgent tokens are also measured.

To evaluate and diagnose a concurrency architecture for a set of scenarios, a simulation is automatically constructed and performed.

1.6 Class of systems considered

To summarize the class of software systems that is analyzed in this research, it has

- a single processing node, which may be a multiprocessor
- the defining characteristic is a single memory address space
- soft deadlines - a specified percentage of responses in a certain class must complete within the specified delay for that class. Hard deadline systems may be analyzed by specifying 100% success for responses, but this research makes no guarantees for hard deadline systems.
- pre-emptive scheduling
- open arrivals and departures (although the ideas should carry over to closed systems as well).

1.7 Contributions

The original contributions of this work are:

- A number of extensions to the Use Case Map notation for scenarios, which allow or assist performance information to be annotated (see Chapter 3). Use Case Maps is a notation for graphically describing scenarios of a software system and the software components which implement the scenarios.

- A virtual implementation which can be constructed from a set of scenarios and a proposed concurrency architecture (see Chapter 4). The execution of the virtual implementation includes:

  - A set of rules which controls the creation, movement, cloning and destruction of tokens.

  - A set of rules which ensures that at any given moment each thread is scheduled with an intelligent deadline.

- A translation of the VI into a simulation model that captures performance measures (see Chapter 4).

- Three diagnostic metrics which can identify concurrency problems in an architecture and discussion of improvements (see Chapter 5)

- A tool named PERFECT (PERFormance Evaluation by Construction Tool) which implements the methods for generating a virtual implementation, running the simulation to necessary levels of accuracy, and calculates the diagnostic metrics.

- Tutorial examples (see Chapter 5) and a substantial case study (see Chapter 6) which demonstrate the way in which serious and diverse problems can be resolved using the methods of this thesis.

1.8 Organization of thesis

This thesis is organized as follows. Chapter 2 gives background on partitioning activities into processes, specifying system behaviour, scheduling in soft real-time systems, and diagnosing designs. Chapter 3 introduces Use Case Maps, which are used in this thesis for specifying the scenarios of an application and showing a concurrency architecture. Chapter 4 explains how a virtual implementation can be constructed and simulated. Chapter 5 defines metrics used to diagnose a concurrency architecture. Chapter 6 presents a case study on which the methods described in this thesis are used. Finally, Chapter 7 contains conclusions.

Chapter 2

Background

This chapter describes various avenues of research that are important background for the present research.

First we review how the intended behaviour of concurrent systems can be specified.

In Chapter 1 we introduced the notion of an activity, some identified chunk of processing that occurs along a scenario. We also stated that we could decide where to provide separate kernel entities to manage the execution of these chunks of processing. Described in this chapter are several author's work on how to partition activities into kernel entities, here called processes.

Because of its impact on performance, scheduling of kernel entities is very important and so this chapter reviews relevant topics in scheduling.

Finally, this chapter reviews other author's work on diagnosing designs.

2.1 Specification of system behaviour

Software architects need a suitable specification of the intended behaviour of a concurrent system. This section describes a number of techniques that are used to specify behaviour.

2.1.1 Written text for scenarios

In the Merriam-Webster dictionary, one of the definitions of scenario is: "an account or synopsis of a possible course of action or events". This is the definition which is applicable when we use the word scenario from the software engineering perspective. For us, a scenario describes the course of actions that occurs for a certain sequence of events.

Development methods typically begin by expressing each scenario with written text. This is true of Jacobson's Object-Oriented Software Engineering method [36], Real-Time Object-Oriented Modeling [59], the Fusion method [24], the Object Modeling Technique [55], and Booch's Object-Oriented Design [12]. These descriptions are not detailed enough to support performance evaluation, so this approach was not followed for the present work.

2.1.2 Message sequence charts

In many methods the activities in the scenarios are then allocated to components and the scenarios are formalized with notations such as Message Sequence Charts (MSCs) [3] or Sequence Diagrams in UML [56]. A scenario may start as a message to one component, which will perform some activities, and then the scenario can be handed off to another component in a message.

2.1.3 Extended state machines

Extended state machines provide a way of describing the behaviour of a system in terms of the behaviour of the components comprising the system. Each component can be in one of a number of states. A transition between states is triggered by receiving a message,[1] and leads to actions which may include performing activities or sending messages. If the behaviour of each component has been described accurately and the components have been joined correctly, then the behaviour of the system will have been successfully specified.

[1] "Receiving a message" can be interpreted broadly as receiving notification that some event has occurred.

Extended state machines in the forms of Estelle [66] and SDL [66, 2] were introduced to model communication protocols and spread rapidly to communication software and to real-time systems in the form of Statecharts [30, 31]. ROOM [59] uses a notation derived from Statecharts. UML [56] also uses an object variant of statecharts.

Note that a predetermined partitioning of activities among concurrent components is needed in order to describe system behaviour using extended state machines. Thus this approach to behaviour does not help us carry out the initial partitioning.

2.1.4 Process algebras

A process algebra allows behaviour (or scenarios) to be specified formally with a language. Parts of scenarios can be described separately and can be composed.

LOTOS is a language which is often used for specification of communication services and protocols [66]. LOTOS combines a behavioural model derived from the process algebras CCS [48] and CSP [33] together with an abstract data typing language based on Act One [66].

Stochastic Process Algebras [1] have been used for performance evaluation, and this is an approach which might be useful for evaluating concurrency. However the processes in the algebra and the assumed synchronous communications between them appear to impede the introduction of concurrency partitioning which does not follow the predetermined boundaries in the specification. Thus, like extended state machines, this does not really help us carry out the initial partitioning.

2.1.5 Graphical specification of scenarios

A good way to express system scenarios and their interactions is with a graphical notation. This is very successfully done with Use Case Maps [15], but can also be done with other graphical notations such as Activity Sequence Networks [16], the UCLA Graph Model of

Behaviour [67, 22, 23], and Petri Nets [52, 62, 28]

Use Case Maps, which are used in this research, were developed by observing software designers describing their work. In the pictorial version of a UCM, a scenario follows a path (which can be drawn as a line with forks and joins) with responsibilities indicated along it. Abstraction is provided via subdiagrams, and details are often suppressed. The binding of components to scenario elements can be indicated graphically, and can also be adjusted graphically, which supports the present desire to create and adjust the initial partitioning into processes. A formal graph structure also exists for Use Case Maps. Use Case Maps are described further in Chapter 3.

Use Case Maps can in some ways be thought of as a graphical depiction of a process algebra. In [7], Amyot et al. demonstrate UCMs being translated into LOTOS. After translation into LOTOS, tools can be used to analyze and validate the specification. These include LOLA [54] for analysis and design testing, and LMC [27] for checking temporal logic properties.

Use Case Maps could in theory be analyzed as well, but to this date they have been used primarily to reason about systems in a somewhat informal way and such analysis and validation tools have not yet been developed directly for them.

Petri Nets can be used similarly to UCM paths to capture system scenarios. A feature of petri nets is tokens. These can be used to mark the progress of scenario instances through a scenario captured as a petri net. Some Petri Net formalisms, such as PROTOB [9], also allow abstraction. Petri Nets can be readily analyzed.

To be useful in describing a concurrency architecture, a Petri Net notation would have to allow some way of indicating a partitioning into processes of the Petri Net places and transitions describing the scenarios.

2.2 Partitioning activities into processes

In design methodologies for concurrent software, an important step is the partitioning of the specified activities into processes. In [19], deChampeau, Lea, and Faure refer to such partitioning as clustering, and state: “Because clustering remains something of a black art, it is very convenient to use a prototyping tool to assist in the evaluation of clusterings.”

They list 17 criteria which one can consider while partitioning a set of objects into processes . Some of the more interesting criteria are:

- Forced partitioning: Identify processes with objects as mandated in non-functional requirements documents.
- Functional partitioning: Identify processes with coarse-grained objects identified in analysis.
- Structural partitioning: Identify processes with objects that are easy to isolate. For example, transaction loggers and other “message sinks” which consume events generated by a large number of other objects without communicating back to them.
- Service-based partitioning: Isolate objects that perform well-known, generic services in their own processes.
- Path-segment-based partitioning: Combine all of those objects involved in a particular sequential path segment into a process.[2] This implies, but does not explicitly specify, that it is wise to put parallel path segments in separate processes.
- Link-based partitioning: Partition into processes so that as many object links as possible point to objects residing in the same process. This avoids fragmentation, in which objects include some components situated in one process and some in another.

[2] They actually call this task-based partitioning as they call a (sequential) path segment a “single-threaded task”.

- Communication-based partitioning: Allocate heavily interacting objects to the same process.
- Recovery-based partitioning: Isolate failure-prone objects (e.g. those interacting with unreliable hardware) in their own processes to facilitate restarts, etc.
- Maintenance-based partitioning: Isolate objects of classes that are most likely to change in the future.

deChampeau et al. recognize that the criteria in their list may overlap or be incompatible. They also recognize that partitioning must be performed using heuristic approaches, and that performance must be balanced against other design factors, such as maintainability, that argue for the use of functional and structural criteria in addition to resource concerns.

Gomaa discusses partitioning in [29]. His ideas that do not appear in Lea's list include:

- Temporal cohesion: group into one process activities which need to be executed in response to the same event.
- Task priority criteria: create a process for a time-critical activity.

Mok proposed three strategies for partitioning activities in hard real-time systems [49].

- Decomposition by critical timing constraints: create one process to perform the computation associated with each timing constraint.
- Decomposition by centralizing "concurrency control": create one periodic process for each equivalence class of compatible periodic timing constraints and one sporadic process for each asynchronous timing constraint. Concurrency control in this context means enforcing critical sections.

- Decomposition by distributing "concurrency control": a process is created for each activity.

Mok explains that decomposition by distributing concurrency control allows a resulting design to be easily distributed across multiple processors, but exacts a higher cost for interprocess communication and concurrency control. He also notes that changing the period of an input or deadline of a response can require modifying the scheduling attributes of many processes. Regarding overhead, Gomaa agrees with Mok stating that too many tasks can unnecessarily increase overhead due to communication and synchronization. deChampeau et al., on a similar note, state that reasonable choices about the number and size of processes depend on the efficiency of the underlying operating system scheduling and interprocess communication mechanisms.

Buhr shows examples of process partitioning for a protocol-processing subsystem [15] that are similar to Mok's strategies.

- Buhr's first partitioning has one process for the path that sends packets and one process for the path that receives packets. This would correspond to Mok's decomposition by critical timing constraints, assuming a timing constraint maps to one of Buhr's paths.

- Buhr's second partitioning has one process for every activity and corresponds to Mok's decomposition by distributing concurrency control.

- Buhr's third concurrency architecture has a single process which controls multiple (assumed-) equally-urgent paths.

- Buhr adds a class of concurrency architectures that is not mentioned by Mok: having a single multithreaded-process which serves one or more paths. In Buhr's case study this allows multiple transmissions and/or receptions to proceed concurrently.

Thus software architects are given plenty of issues to think about when they are developing the concurrency architecture for software. They need help to know which issues are important for a particular project. For a given issue there may be lots of options, but there is currently little guidance to help select between the options. This is the starting point for this present research.

2.3 Scheduling in soft real-time systems

Because of their impact on performance, scheduling issues are very important in the design of all concurrent systems including soft real-time systems. There is a large literature on scheduling: a good survey of scheduling in real-time systems is given by Mercer [44].

Earliest-Deadline First (EDF) is the scheduling policy used in the research described in this thesis. The following sections introduce and provide background on EDF. Static priority scheduling, a very simple policy, is described first. Then a benefit of allowing priorities to change, that is of dynamic priorities, is described in a section on priority inheritance. Of policies with dynamic priorities, EDF has a number of benefits and is described in section 2.3.3.

2.3.1 Static priority scheduling

A simple, low-overhead scheduling policy is to assign a single priority to the thread (or threads) of a process when created, and never change that priority. Typically a higher-priority thread can pre-empt a lower-priority thread.

Those hard-real-time systems for which each class of external events has a minimum non-zero spacing in time and for which activities have a finite maximum processor demand often use one of the rate monotonic family of scheduling policies [42]. These policies use static priority scheduling with pre-emption.

Daigle et al. describe a static priority scheduling policy for communication processing

systems where a scenario is split up into a sequence of activities[3] [18]. They discussed a packet switching software example where the processing of an incoming packet was divided into 13 activities which were handled by 10 processes. Daigle found that to get fast response times, that processes should be given priorities such that the later a process is in the sequence, the higher its priority should be.

2.3.2 Priority inheritance

It is often useful to change the priority of a thread, even if only temporarily. Consider for instance the problem of unbounded priority inversion [60], where a high-priority thread is waiting for a low-priority thread to release a resource (such as a semaphore) but one or more medium priority threads keep the low-priority thread from running. Such a situation can be avoided if the low-priority thread "inherits" the priority of the high-priority thread while the high-priority thread is waiting [60].

Unbounded priority inversion can be caused by messaging in addition to synchronization [45, 65]. To avoid unbounded priority inversion when messaging and synchronization are nested, priority inversion must be managed using an integrated method such as the Integrated Real-Time Resource Management Model [41].

2.3.3 Earliest Deadline First

If the priority of a thread can change, a good scheduling policy for soft real-time systems is Earliest Deadline First (EDF) [43]. A deadline is some absolute time in the future when a response should be finished.

A number of authors have found EDF to be a good policy for real-time database systems, especially at low to moderate loads (eg. [5, 68, 32, 34]).

[3]Using the terminology of Daigle et al., a job is split up into a sequence of tasks.

Abbot and Garcia-Molina in [5] report for systems with exponentially distributed inter-arrival times that EDF scheduling worked best at lower loads, and Least Slack, statically evaluated, performed better at higher loads. For statically evaluated least slack, slack is measured once when a transaction arrives and is defined by:

$slack = deadline - (arrival_time + service_time)$

where $service_time$ is the total service time required at all devices.

During overload conditions, EDF suffers from the "Domino Effect" [64, 32], 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 [32] propose algorithms called Adaptive Earliest Deadline (AED) and Hierarchical Earliest Deadline, the latter of which considers the assigned "value" of a response completing on time. In [26], Haritsa et al. use the AED in a network management system.

J. Huang et al. found on a testbed system that the CPU scheduling algorithm is very important in improving the performance of real-time transactions. They also found that overheads such as locking and message communication are non-negligible and can not be ignored in real-time transaction analysis [34].

If the EDF policy cannot meet all deadlines, it has a tendency to discriminate against longer jobs. A policy called Adaptive Earliest Virtual Deadline (AEVD) has been developed to overcome such bias [51].

2.3.4 Subtask deadline assignment

The EDF policy was developed assuming only one device (a single processor) was being used by the jobs. If the jobs can use multiple processors, then parallel subtasks can be executed in parallel. It has been found that there is a problem if all subtasks are scheduled with the deadline of the parent task: if one parallel subtask is late, then the whole task

is late. A number of strategies for assigning a deadline to each parallel subtask have been proposed [39]. Strategies have also been proposed for assigning deadlines to sequential subtasks [38].

2.3.5 Scheduling disk requests with deadlines

Disk scheduling normally does not consider deadlines or priorities of disk requests. The better conventional algorithms for scheduling disk requests instead are focussed on the location of data on a disk when scheduling the disk access [53]. Examples are Shortest-Seek-Time First (SSTF), which chooses for next service the request that is closest to the current head position, and SCAN, in which the head sweeps back and forth across the disk surface servicing all requests that lie ahead of it.

In [4], Abbot and Garcia-Molina propose a number of policies which do consider request deadlines. In their policies, they give preference to read requests over write requests as long as there are sufficient memory buffers to hold outstanding write requests. This is due to the assumption that once a transaction is committed in memory there is no deadline to write changed data back to disk. The simplest of these is EDF, which applies the same EDF policy used for the processor to disk read requests. Policies which consider both data locations and request deadlines are Earliest Deadline SCAN (D-SCAN), which modifies the traditional SCAN algorithm so that the track location of the read request with the earliest deadline is used to determine the scan direction, and Feasible Deadline SCAN (FD-SCAN), which is similar to D-SCAN except that only read requests with feasible deadlines are chosen as targets that determine the scanning direction.

2.4 Diagnosing designs for performance problems

In her pioneering work on software performance engineering, Connie Smith [17] advocates a loop of performance evaluation and lifecycle concept improvement at every stage of a

computer application's lifecycle. She discusses ways to improve designs without pushing any particular detailed development strategy. She lists a number of performance principles. An example is the parallel processing principle, which says: execute processing in parallel (only) when the processing speedup offsets communication overhead and resource contention delays. In a case study, she reduces the best-case response time from 1026 seconds to 4.3 seconds by applying these principles.

John Neilson et al. show that software bottlenecks may exist in concurrent software [37]. A software bottleneck occurs when a task exhibits a high utilization which is also high relative to the utilizations of each of its servers, either direct or indirect. They describe how software bottlenecks may be alleviated by multi-threading or "cloning" .

Gomaa describes how a concept called task inversion can be used in situations where there are concerns about high tasking overhead [29]. Task inversion is essentially the process of combining the functionality of multiple tasks into one task.

Hesham El-Sayed proposes a design-improvement process which generates and solves an LQN Model for a design, and then either increases the priority of the task deemed most suitable or reshapes the design by re-allocating or splitting that most suitable task [21]. The most suitable task is chosen from among those tasks involved in performing non-compliant scenarios. A task is split by putting the entry which suffers most from waiting time into a separate task. The process then iterates by returning to the step of generating and solving an LQN Model, this time for the modified design.

2.5 Realizing potential parallelism in massively parallel processing

This thesis is concerned with software concurrency architectures for soft-real-time server applications. A somewhat related area is finding and realizing potential parallelism for

massively parallel processing – used for example for scientific computing (see for example [25, 6]). Both areas try to exploit parallelism to increase performance. For scientific computing there is typically only one response – the end of the long parallel program – and typically these programs would not pre-empt each other. Soft-real-time server applications, on the other hand, typically deal with a number of classes of relatively short responses, and often have multiple responses competing with each other to finish.

2.6 Software architecture

Software Concurrency Architecture is one aspect of the general field of Software Architecture [61, 40], which deals with all large-scale issues of software design.

In [40], the software architecture of a program or computing system is defined as:

"the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."

The book [61] is a good introduction to the concerns of software architecture, which go far beyond performance. Evaluation of architectures is addressed in [40], and at greater length by by Bass, Clements and Kazman in [10]. They identify goals in terms of qualities such as performance, security, availability, reliability, modifiability, and they relate these to architecture features and styles. They describe architecture evaluation by review, using a qualitative scoring approach to assessment, and scenarios to define challenges to each particular quality.

The present concern for evaluation for performance can easily be embedded in the broader approach of [10].

Chapter 3

Use Case Maps with Performance Annotations

This chapter describes the scenario notation called Use Case Maps, and how it has been specially modified to support performance analysis. Use Case Maps were defined by Buhr [13] to specify scenarios and their relationships to software components.

3.1 Basic UCM notation for specifying scenarios

Figure 3.1 shows example UCM scenarios and the basic UCM scenario elements have been named.

The scenarios are indicated by smooth solid lines starting at filled circles called *path starts*, and ending at short perpendicular bars called *path ends*. The thicker bars along a scenario indicate AND forks and AND joins and allow possible concurrency in a scenario to be expressed. When a path contains one or more OR forks or OR joins, it implies that multiple scenarios are sharing portions of the path. The path segment before an OR fork is shared by multiple scenarios. The path segment after an OR join is also shared by multiple

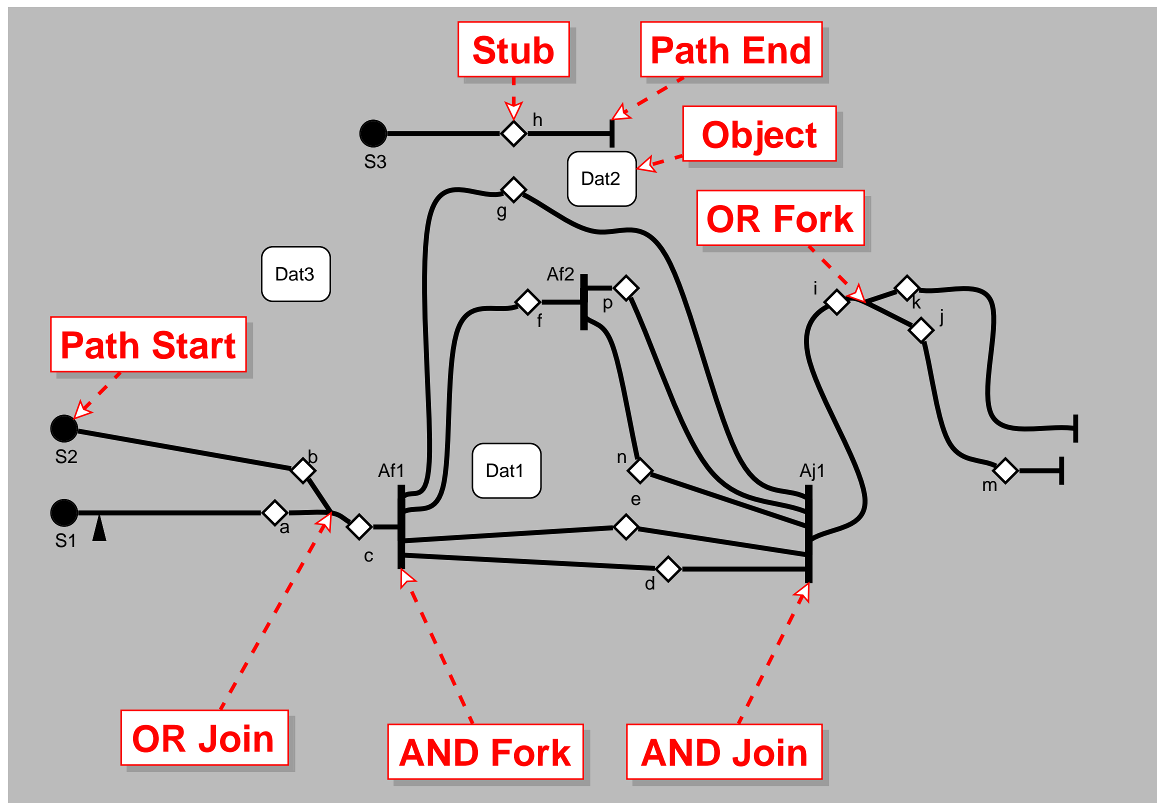

Figure 3.1: UCM scenarios with basic notation indicated

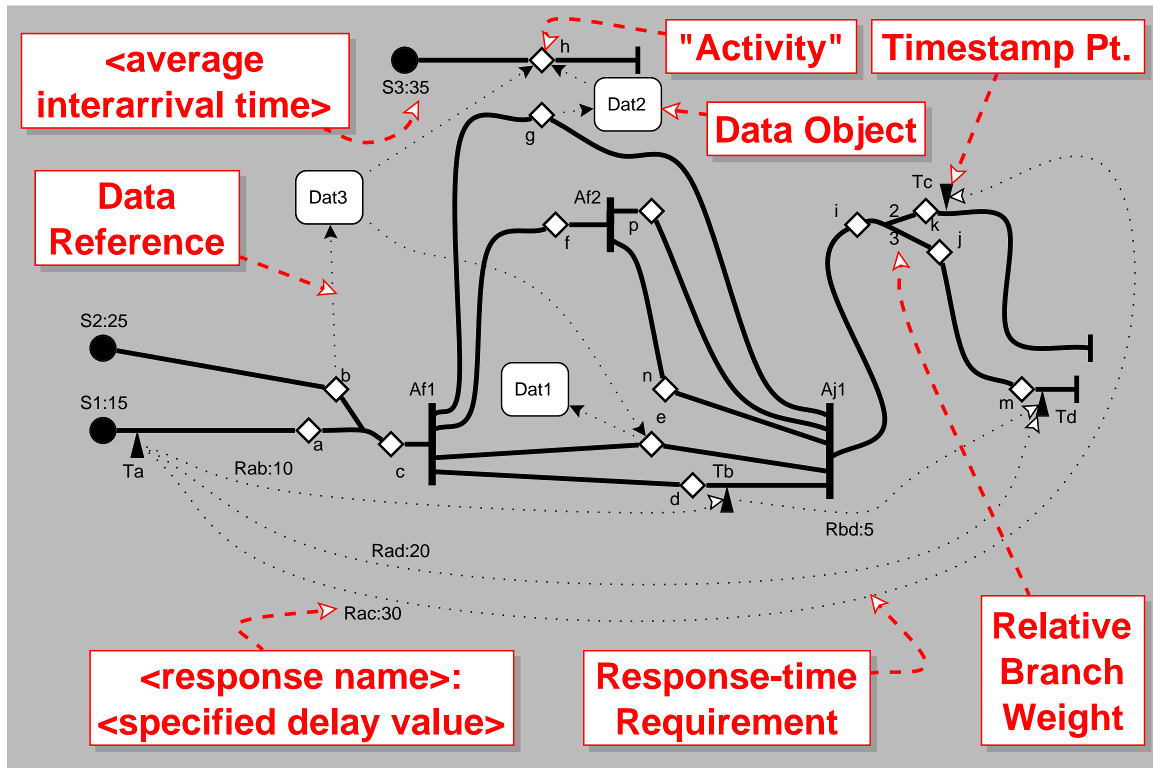

Figure 3.2: UCM scenarios with performance extensions indicated

scenarios. A *stub* is represented by a small diamond shape and indicates a portion of a scenario which is defined elsewhere in a “plugin” diagram.

3.2 Performance extensions to the basic scenario notation

The basic UCM notation has been extended to include needed performance information such as arrival processes, branch selection criteria for OR forks, and response-time requirements. These and other extensions are indicated in Figure 3.2 and are described in the following subsections.

3.2.1 Tokens and workload intensity

A scenario is started by a relevant event occurring in the environment of the system being described. An example event is the arrival of a request to the system. When such an event occurs, think of a *token* being created at a path start and starting to travel along the path of a scenario. This token is similar in many ways to a Petri Net token. To get performance predictions we need estimates of the workload intensity, and so for each path start we require that an arrival process for events be specified. Unless specified otherwise, the examples in this thesis have Poisson arrivals at a path start. The average interarrival time for a path start can be indicated on a UCM diagram beside the name of the path start.

3.2.2 Activities, their references to data objects, and their service requirements

In this thesis, we define as an *activity* a stub whose plugin is single-input/single-output, and has no AND forks or AND joins. The stub may cross passive components such as passive data objects. Such an activity will require service from a sequence of declared devices when executed. We make this definition because a stub is a more general concept than we need, for example allowing multiple inputs and multiple outputs.

The plugins which define activities in this thesis will be relatively simple: when an activity is is executed, think of a method or procedure being invoked which may in turn invoke other methods and procedures thus forming a call graph. To show that an activity's plugin will invoke one or more methods of an anchored[1] data object, a dashed line is drawn between the activity and the data object. Typically in UCM's this connection is identified only in the plugin diagram. An arrow pointing to the data object implies that the activity will write to the data object. Conversely, an arrow pointing to the activity implies that the activity will read from the data object.

[1]An anchored component is one that is referenced in a plugin, but is defined at a higher level.

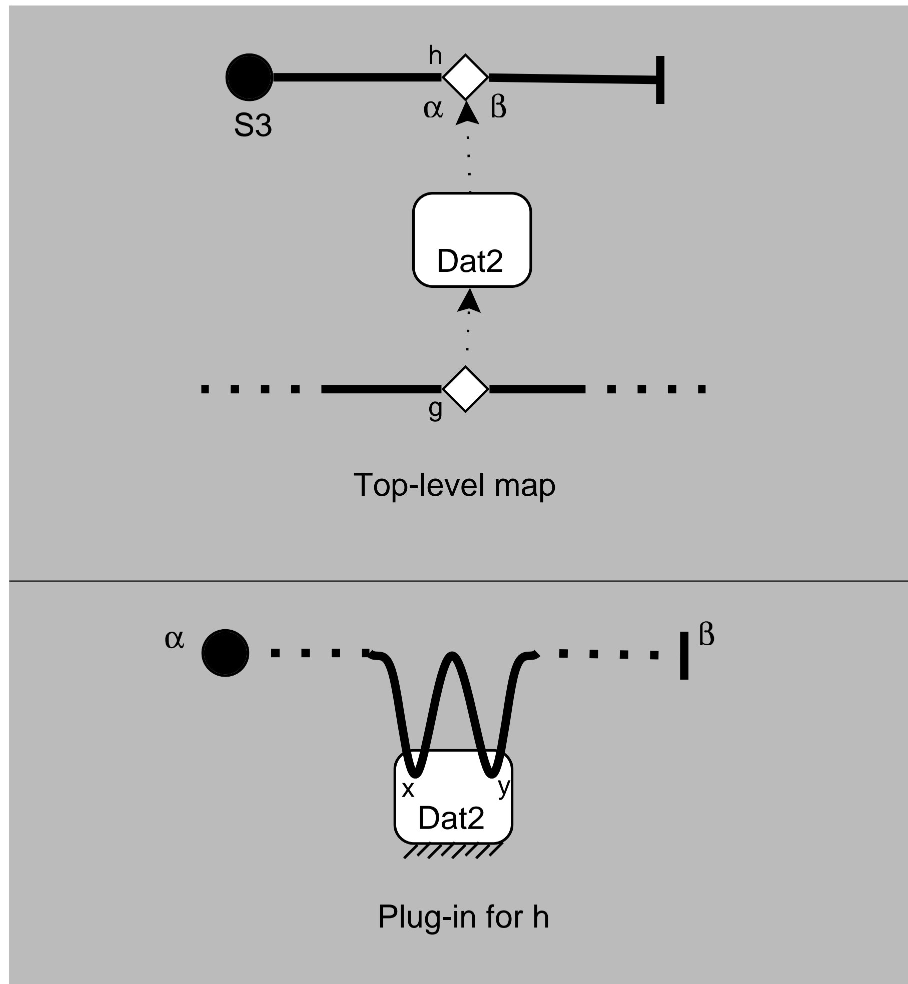

Figure 3.3: A possible plugin for activity h

Figure 3.3 shows a plugin for activity h of the example UCM. The plugin assigns two responsibilities x and y to the anchored data object *Dat2*.

When an activity is executed, service will be required from a sequence of declared devices such as the processor and disks. In the sequence, the amount of service required of a device will be specified by a distribution and required parameters. An example sequence of service requirements is:

1. CPU: A processing demand of 30ms, a deterministic quantity. Processing demands

are defined in the model for a reference processor, and then scaled according to the processor speed when evaluating a particular deployment.

2. Disk1: A number of accesses uniformly distributed between 1 and 3.

3. DSP1: An amount of time exponentially distributed with a mean of 0.5ms (as measured on a reference Digital Signal Processor)

Although not illustrated in this thesis, PERFECT also allows operations within the executing activity to be defined, so as to create, modify, or delete data associated with the token which arrived at the activity. As explained in the next subsection, this data can be used to route the token at OR forks.

Characteristics may be specified when declaring devices. For example the speed of the processor relative to the chosen reference processor must be specified.

3.2.3 Branch selection criteria at OR forks

At an OR fork, an arriving token can be routed to one of a number of branches. In the UCMs presented in this thesis the branch is chosen randomly, and a relative branch weight can be assigned to each of the branches to define the probabilities of choosing the different branches. The OR fork in Figure 3.2 has branches weighted 2 and 3, giving probabilities of 2/5 and 3/5 respectively. Where no branch weights are specified, any branch will be chosen with equal probability.

PERFECT also allows for data associated with a token to control which branch the token should follow. Specifically, if a token has a piece of numeric data called BRANCH_SELECTOR, then the token will follow the branch whose label matches the value of BRANCH_SELECTOR.

3.2.4 Timestamp points and response-time requirements

A response is the completion of some requirement following a certain begin event. The response time for a response is the time difference between the occurrences of the begin event and the end of the response. A single begin event can trigger multiple responses. For example, when a file is requested of a web server, two responses may be sending the file to the client and logging the request to a log file. Each type of response can have its own response-time requirement. In this thesis, a response-time requirement specifies a delay and the fraction of responses (usually 90%) which must complete within that delay.

On the path of a scenario, the begin and end events of a certain response are defined by two points. The points are indicated on the path by timestamp points. Timestamp points are marked by filled triangular arrowheads along a path and are used to "instrument" the UCM. A type of response can be indicated by a dotted line between two timestamp points with an arrowhead pointing to the end of the responses. The specified delay value for the response-time requirement and optionally a name for the type of response can be indicated along the dotted line.

For example, when a message arrives at the path start *S1* of Figure 3.2, the newly created token immediately passes timestamp point *Ta* where it is stamped with the current (i.e. arrival) time. The token carries its timestamp history throughout its lifetime. When a token crosses the timestamp point at the end of a response-time requirement, such as *Td*, the response time for that response is determined (provided the token has previously passed the begin point of that response). In Figure 3.2, a token starting at *S2* or one of the token's descendents may generate a response-time for a response of type *Rbd* but will not for other types such as *Rad*. The response-time requirements in this thesis all specify that 90% of responses be completed within a specified delay.

In summary, to define a type of response and its response-time requirement, the following must be specified:

- A begin event
- An end event
- A delay value
- The fraction of responses of that type which must complete within the delay.

3.3 Partitioning of path elements

In addition to specifying scenarios, UCMs allow one to associate the scenario elements with software components such as processes and modules or packages. Since this thesis is concerned with concurrency architectures, UCM's notation for active components like processes is of most interest.

The partitioning of the execution between processes is indicated by drawing in the background parallelograms representing processes. Each path element in a UCM belongs to the process drawn behind it. As shown in Figure 3.4 using red labels, a single-threaded process such as *stp1* is indicated by a parallelogram, and a multi-threaded process such as *mtp1* by a stack of parallelograms. Each process is named. The scenarios can be reshaped to ease partitioning, and if desired multiple parallelograms (or multiple stacks of parallelograms) identically named can be drawn in separate parts of the diagram to include in one process scenario-elements which are awkwardly separated on a diagram.

The partitioning defines whether a data object will be shared or not. If the data object "falls" in a multithreaded process, or is accessed by activities in different processes, then it is shared, and must be protected in some way when it is accessed.

A partitioning implicitly introduces overhead operations including messages between processes, context switches between threads, and access control for the shared data objects. These are indicated in the colour blue in Figure 3.4. Details of when these overhead operations occur will be explained more fully in the next chapter. However, it is apparent

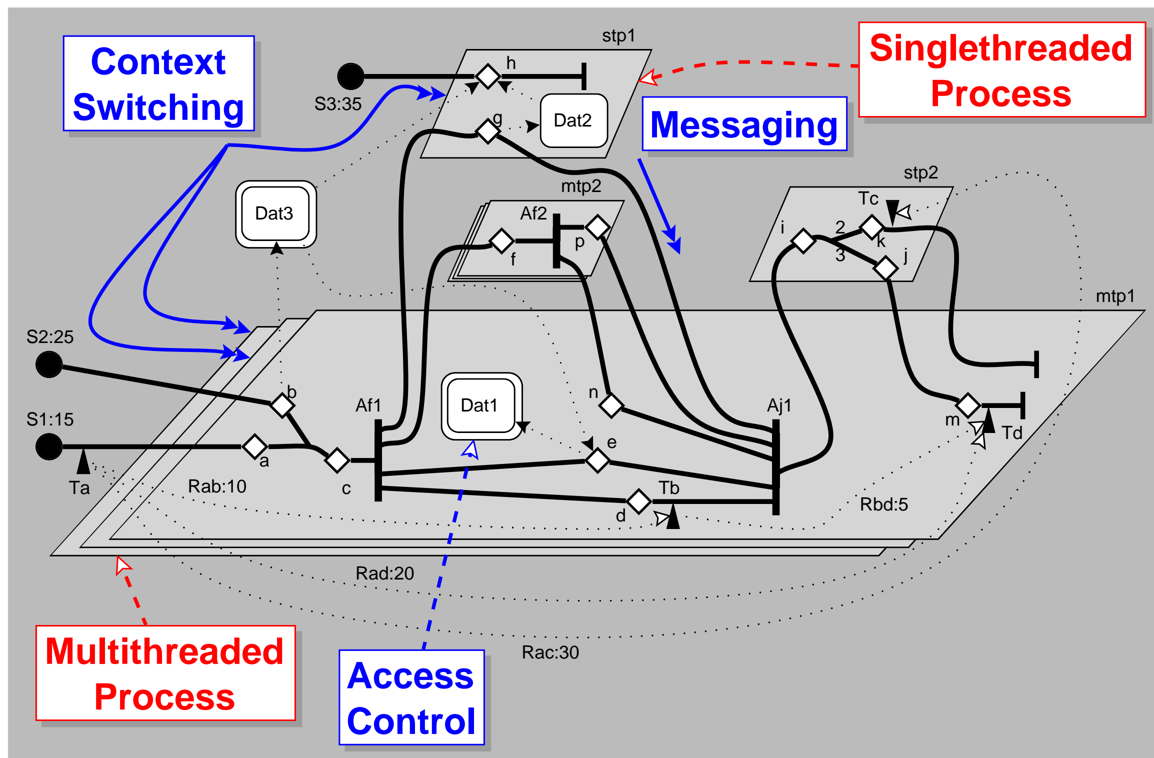

Figure 3.4: A partitioning of the example UCM, with process types labelled in red, and overhead operations labelled in blue.

that certain overhead operations occur during an execution of a scenario starting at S1 in Figure 3.4.

- There are messages from a thread of *mtp1* to processes *stp1* and *mtp2* and later to process *stp2*, and also messages from a thread of each of *stp1*, *mpt2*, and *stp2* to the originating thread of *mtp1*. Note that two messages are sent from *mtp2* to *mpt1*, and only one message from each of *stp1* and *stp2* (a message will be sent for only one of the OR-fork branches in *stp2*).
- In order for each process to do its share in processing the scenario, context switching will be necessary between a thread in each of the processes.
- Access control will be performed when activities *b* and *e* access shared data object *Dat3*, and when activity *e* accesses shared data object *Dat1*. Thus access control will be needed whether data objects are shared between processes, as is the case with *Dat3*, or are shared between threads of a single process as is the case with *Dat1*.

Figure 3.5 shows the same scenarios but with a different concurrency architecture. In this architecture fewer overhead operations will occur.

3.4 Incorporation of performance annotations into UCM Navigator

The UCM Navigator [14] is a graphical tool for editing UCMs. It allows one to draw scenarios with just a few clicks of a mouse. A component such as process is also easily drawn by just clicking and dragging the mouse.

The performance annotations described in Section 3.2 have been incorporated by Andrew Miga [47] into UCM Navigator. Not currently implemented in the tool are the dotted lines drawn

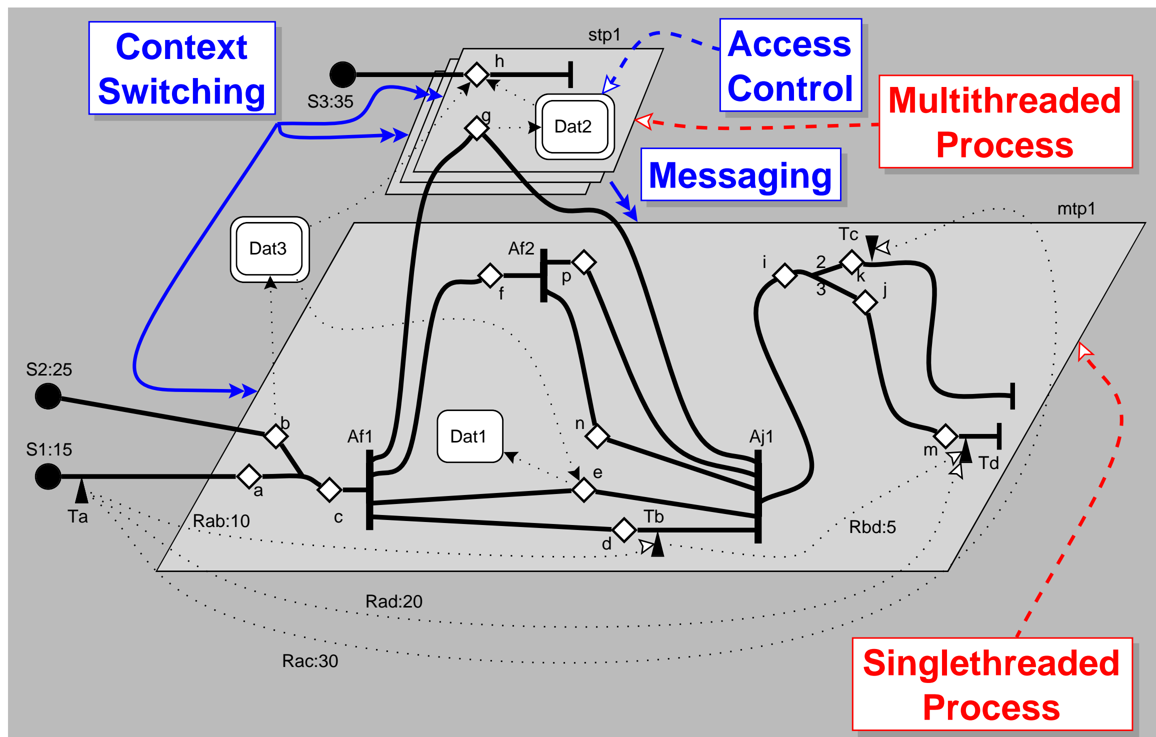

Figure 3.5: A different partitioning of the example UCM

- from a stub to a data object to show that the stub's plugin references the data object (see Section 3.2.2)
- from one timestamp point to another to indicate a response (see Section 3.2.4).

UCM Navigator saves entered UCMs in a defined XML format [8].

Chapter 4

Virtual Implementation to Evaluate Architecture

This chapter explains how a virtual implementation is automatically constructed and its execution is simulated to provide quick evaluation of a concurrency architecture. The goal of the virtual implementation is to allow the full performance potential of the architecture to be exploited. In constructing the virtual implementation, certain assumptions are made regarding kernel scheduling and message handling, and tentative decisions are made regarding the behaviour of each thread.

It must be re-iterated that the virtual implementation is not an automated design step. It is simply constructed to make the best of the concurrency architecture expressed in the UCM, and it uses techniques which might not be usable in practice. At least, their use in practice would require further development of platform features, which is outside the scope of this research.

4.1 Processes, threads, and mailboxes

In the virtual implementation there is a process for each process in the specification. Each process has its own *process* mailbox. If the process is singlethreaded, it is a program with a single instance which can receive messages from the *process* mailbox. Alternatively, if the process is multithreaded, it has a potentially unbounded number of threads which can all receive messages from the one *process* mailbox.

If an execution path leaving a thread later returns, it will arrive as a message to a *return* mailbox private to the thread.

4.2 Thread behaviour in the virtual implementation

The behaviour of a thread is governed by a controller which sequences the execution of activities and the invocation of kernel primitives. In the simulation, 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 [59] and UML [56]). The state of a thread, R, consists of a set $\mathcal{T}_R$ of tokens being processed, each of which has a record of its location on a UCM subpath, and the number, n_R, of tokens which are expected to return to the thread in messages to the return mailbox.

Each thread starts by obtaining a token from the *process* mailbox of its containing process. Stored in this token is the token's location on a UCM path (the location will be on a path segment crossing into the process). The controller moves the token to the next path element and updates the token's location attribute.

4.2.1 Behaviour at an activity

When the token arrives at an activity, it begins to execute the activity's specification by requesting service at the first of a sequence of devices. If the activity needs access to data objects shared between threads, then read or write permissions as appropriate will be requested first, before service is requested from any device. The request may cause the thread to be blocked. To avoid deadlock trying to access shared data objects, they are globally ordered in the model, and all requests are made according to that order. To reduce the occurrence of a thread holding a data object while it is blocked waiting for permission to access another, in general data objects subject to greater contention should be placed earlier in the ordering. In Figure 4.1 for example, activity *e* reads from *Dat3* and writes to *Dat1*. Assuming *Dat1* is placed in the order before *Dat3*, when a token in a thread of process *mtp1* reaches activity *e*, write permission will first be requested for *Dat1*, and then read permission will be requested for *Dat3*. Because of the global ordering, another token reaching activity *e* in any other thread of *mtp1* will have to request access to those data stores in exactly the same order. When the last in the sequence of devices has finished serving the activity for a token, the data stores will be released and the token will be moved along the path.

Note that permission to access shared data is requested at the granularity of a whole activity. In order to get access to a shared data object for only part of the duration of an activity, the activity would be subdivided into multiple activities.

4.2.2 Behaviour when forking

When a token arrives at an OR fork along a UCM path, the token will be routed along one of the outgoing branches. In the UCMs presented in this thesis, the output branch is chosen randomly according to the specified relative branch weights. The evaluation method also allows for data to control the choice of branches as was explained in section 3.2.3.

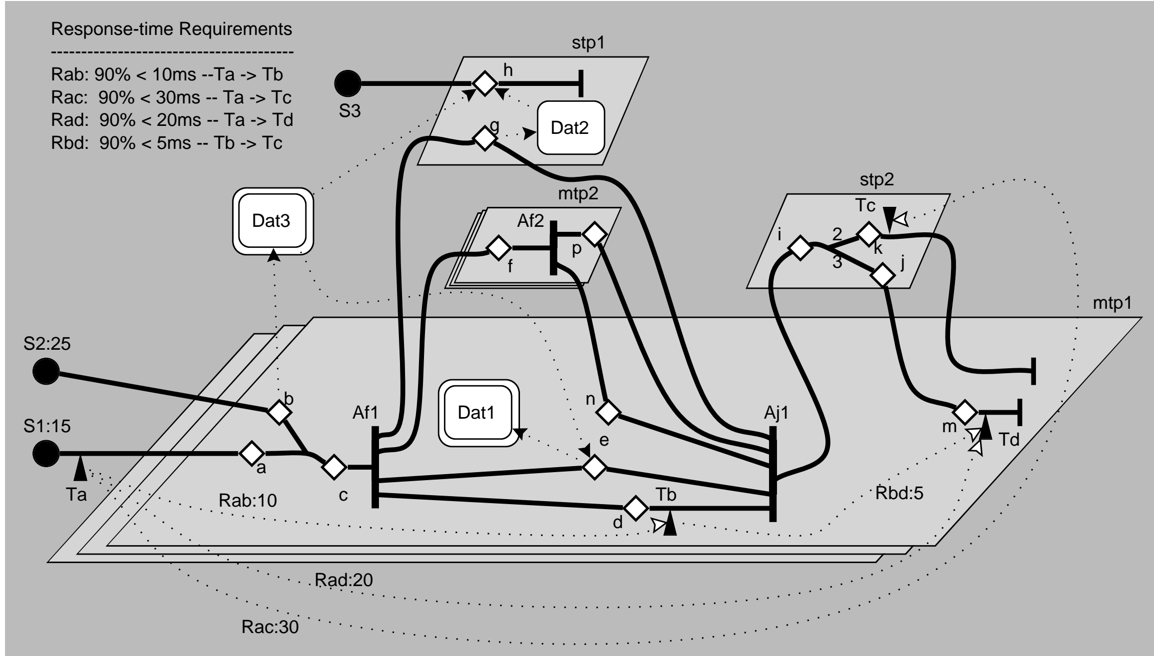

Figure 4.1: A UCM showing scenarios and a concurrency architecture

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. The cloned tokens will be referred to as *descendents* of the arriving token.

4.2.3 Behaviour when a token leaves a process

When a token T reaches a point where a path segment crosses from one process, P_s, to another, P_d, the thread, R_s, of P_s, under whose context token T has been executing, will send a message referencing T. If token T or a descendent will later return to P_s then R_s – keyed to P_s – is stored in T in dictionary $\mathcal{R}_T$ and R_s's count of expected returning tokens, n_{R_s}, will be incremented as described at the bottom of this section. If token T or an ancestor has previously visited P_d then it was handled by R_d, the thread in $\mathcal{R}_T$ keyed to P_d, so the message goes to the *return* mailbox of R_d. Otherwise, the token goes to the *process* mailbox of P_d. From the above it should be understood that if a token executed by

a thread R_s leaves a process P_s but the token or at least one descendent later returns to P_s, then all descendents which return to P_s, and the original token if it returns, will return to thread R_s via the thread's *return* mailbox.

Returning to the same thread is done mainly to ensure that proper tokens are joined with each other at an AND join. Using the thread behaviour presented in this chapter, joining proper tokens requires that each token joining at an AND join in a thread of a multi-threaded process was either the last token the thread received from the *process* mailbox, or is a descendent of the last token that the thread received from the *process* mailbox. An AND join should generally not be put in a process without this condition being met. A consequence of this requirement is that it is never simultaneously permitted in a thread that tokens are waiting to join at an AND join, the token return count is 0, and there are no tokens resulting from an AND fork which are ready to proceed.

In Figure 4.1, the token that travels through activity c in a certain thread and all the token's descendents will all arrive at that same thread to be joined. Five tokens arrive at the thread of *mtp1* to be joined, but only four of the tokens have previously visited *mtp1*. The fifth arriving token is a descendent that was created at the AND fork in process *mtp2*. Without a *return* mailbox per thread, different tokens that should join together could end up in different threads of a process containing an AND join resulting in possibly undesired behaviour and perhaps deadlock.

An added consequence of tokens returning to the same thread of a process is that intermediate result data needn't be bound to and referenced through the token, but can just remain in the context of the thread if the data will not be needed while the token is away from the process. In Figure 4.1, activity m can directly use any intermediate result data generated at an activity such as d.

After sending a message, a thread must get another token to be the active token following the steps to be listed in section 4.4.2. At this point it suffices to say that a thread, R, cannot

receive from the *process* mailbox of the process it belongs to if $n_R > 0$.

Incrementing returning-token count

When a message is sent from a thread R of a process P, n_R, the count of the number of returning tokens, should be incremented by the number of tokens which are expected to return to R as a result of the message being sent. If no tokens are to return as a result, then the increment will be 0. If tokens do return as a result, the the increment will often be 1, but if the token will be AND forked without matching AND join(s) while the token is away from process P, then the increment can be greater than 1. In Figure 4.1, when a token leaves thread R of *mtp1* bound for *mtp2*, n_R is incremented by 2.

In any case, the number of returning tokens must be deterministic: if a token is OR forked after leaving but before returning to a process, all outgoing branches of the OR fork must either OR join or must re-enter the process, even if no processing is done after re-entering the process. This is necessary so that a thread will not wait for a message which never will arrive. In the example UCM of Figure 4.1, when a token leaves thread R of *mtp1* bound for *stp2*, n_R will be incremented by 1. The branch of the OR fork in *stp2* with activity *k* will return to *mtp1* even though no processing is done there.

4.2.4 Behaviour at an AND join

When a token T controlled by a thread R arrives at an AND join, one of two things will happen. If at least one other input branch of the AND join does not have a waiting token controlled by thread R, then T must wait and R's controller must get another token to be the active token following the steps to be listed in section 4.4.2. If each other input branch has a waiting token controlled by thread R, then R's controller must merge into T the information, such as $\mathcal{R}_T$, stored in the waiting tokens and delete the waiting tokens. T then continues on.

4.3 Scheduling policy

The scheduling of the processor in the simulator was chosen to aid in the evaluation of feasibility of the response requirements. Thus, the scheduler should be "good" for systems with soft deadlines and statistically determined arrivals and execution times, yet not have an unreasonably large computational overhead. These scheduling requirements are similar to those of real-time database systems where Earliest Deadline First (EDF) scheduling has been investigated (see Section 2.3.3). All the reported studies show that EDF scheduling works well at lower loads, allowing most of the responses to complete within their deadlines. At heavy loads some other scheduling policies perform better but we are most interested in cases where most responses are completed on time.

During periods of overload, EDF suffers from the "domino effect"[64], 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 [32] propose an algorithm called Adaptive Earliest Deadline. This research has used pure EDF and it has been satisfactory.

4.4 How the scheduling policy influences desired thread behaviour

With EDF scheduling, a thread controller will have to set deadlines at certain points during its execution.

The first thing described is the way in which deadlines for a token are defined. Then, the deadlines of tokens emerging from an AND fork are described, and how the active token for a thread is chosen. Last to be described is the way in which the deadline for a thread is determined, and how AND joins, sending messages, and OR forks influence deadlines.

4.4.1 Defining deadlines for a token

A token, T, has a dictionary $\mathcal{D}_T$ of deadlines which evolves through time. At each time t, the dictionary has one deadline for each response-time requirement that T is subject to at that time. When T crosses a timestamp point which starts a certain response-time interval, rtr, a new value $D_{rtr,T}$ is added to $\mathcal{D}_T$. $D_{rtr,T}$ is defined by adding the response-time requirement's delay value, v_{rtr}, to the current simulation time, t_{sim}.

$$D_{rtr,T} = t_{sim} + v_{rtr}$$

Multiple deadlines can be defined at the same moment if a timestamp point starts multiple response-time intervals (as *Ta* does in Figure 4.1). The earliest deadline in $\mathcal{D}_T$ at any point in time, t, will be termed the deadline, $D_T(t)$, of the token at time t.

$$D_T(t) = \min D \in \mathcal{D}_T \text{ at time } t$$

If a token T is created at *S1* in Figure 4.1 and crosses *Ta* at a simulation time, t_{sim}, of 100.0ms, then token T will have 110.0ms, 120.0ms, and 130.0ms as the deadlines in its collection and 110.0ms will become its current deadline, $D_T(t = 100.0\text{ms}) = 110.0\text{ms}$.

When T crosses the timestamp point which ends rtr, $D_{rtr,T}$ is removed from $\mathcal{D}_T$.

4.4.2 AND fork

It was mentioned in section 4.2.2 that a token is cloned when it reaches an AND fork. 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. For each branch which immediately leaves a process and irrespective of the deadline of the token on that branch, the controller will immediately send a message to the appropriate mailbox of the destination process (the process mailbox or a return mailbox of one of the process's threads). These messages are

immediately sent to guarantee that another thread can begin using the cpu if and when the current thread starts using a device other than the cpu (such as a disk). The immediate sending is even more important if the computer is a multiprocessor.

Because of cloning at AND forks, a controller for a thread R might have to interleave the processing of a number of "ready" tokens. After sending any necessary messages, the controller will have to obtain a token to make active.

1. If the thread R has ready tokens, select as the active token, A_R, a ready token with the earliest deadline, D_{A_R}. Any other tokens will form the thread's "ready-to-proceed" set. $\mathcal{Q}_R(t)$ is defined as the "ready-to-proceed" set at time t.

2. Otherwise, thread R will receive from a mailbox.

   (a) If the number of returning tokens, n_R, is greater than 0 the thread will decrement n_R by 1 and receive from its *return* mailbox.

   (b) Otherwise the thread will receive from the *process* mailbox of the process it belongs to.

In Figure 4.1, after the token T described in section 4.4.1 reaches *Af1*, a message will be sent to each of *stp1* and *mtp2*. Each message will carry a token with a current deadline of 120.0ms. The token on the lowest branch with a current deadline of 110.0ms will become the active token, and the fourth token will have a current deadline of 120.0ms and will form the "ready-to-proceed" set.

4.4.3 Current priority of a thread

The current priority of a thread R will be assigned based upon the earliest deadline chosen from among the earliest deadline of R's active token, the earliest deadline of each token waiting at any AND join and served by R, the earliest deadline of each token waiting at a mailbox exclusive to R, and the deadline of each thread whose active token is waiting

for access to data being used by R. Thus deadlines can be inherited in order to reduce unbounded priority inversion [60]. Note that deadline inheritance is considered separately for messaging and synchronization. When a message is sent to a thread which is waiting to access a shared data object, or when a message is sent to the *process* mailbox of a single-threaded process whose thread is waiting for a message on its *return* mailbox, unbounded priority inversion can again result. To reduce the occurrence of unbounded priority inversion in the latter case, such a single-threaded process should be made multi-threaded. To eliminate all unbounded priority inversion, a method such as the Integrated Real-Time Resource Management Model [41] should be used[1]. If no token currently associated with a thread has a deadline, then the thread will be assigned the minimum possible priority (i.e., have an implied deadline of the end-of-time).

4.4.4 AND join

The above section stated that in choosing the current deadline of a thread, the thread's controller considers the deadlines of tokens waiting at AND joins in the thread. The necessity of this is understood by considering the situation in which a relevant[2] response-time requirement begins on one branch of an AND join and ends after the AND join. *Rbd* in Figure 4.1 is such a situation. When a token T from that branch reaches the AND join, tokens still to arrive at the AND join and join with token T will need to execute at the deadline of T to avoid the possibility of inversion (i.e. allowing the execution of an unrelated token with a deadline later than that of token T). If the AND-join branches with outstanding tokens are completely within the thread containing the AND join and have the most urgent remaining tokens of the thread, there is no problem: according to the previous section the deadline of the thread will be at least as urgent as that of the waiting token in question.

[1] In the PERFECT tool described in Section 4.6 such a method is not currently used, but it can be added.

[2] A response-time requirement is considered relevant if it can affect the scheduling of threads or a thread controller's ordering of operations.

In other cases, such as happens in Figure 4.1, concepts similar to those employed in the Integrated Real-Time Resource Management Model should be used. It is anticipated that a response-time requirement straddling an AND join in this way will rarely be encountered in real applications.

4.4.5 Sending messages

When a message is sent to a mailbox, priorities may need to change. If the message is being sent to a mailbox exclusive to a thread (any *return* mailbox, or a *process* mailbox for a single-threaded process), 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. If the message is being sent to the *process* mailbox of a multi-threaded process, the earliest deadline of the token should be assigned to an idle thread. In a virtual implementation, if a multi-threaded process needs an idle thread but none is available, an additional thread will be created.

If multiple messages are waiting in a mailbox, a message carrying a token with the earliest deadline will be the first to be received.

After sending a message a thread will have to obtain another token to make active by following the steps enumerated in section 4.4.2. The thread's deadline may change. In any case, the thread should have a deadline assigned according to the rules described in section 4.4.3.

4.4.6 Scheduling influences at OR forks

At an OR fork, deadlines which a token T is carrying in dictionary $\mathcal{D}_T$ which are specific to branches other than the one chosen can be discarded. This may result in the token's deadline, $D_T(t)$, becoming less urgent, and possibly the switching of which token is active and/or the scheduling of another thread.

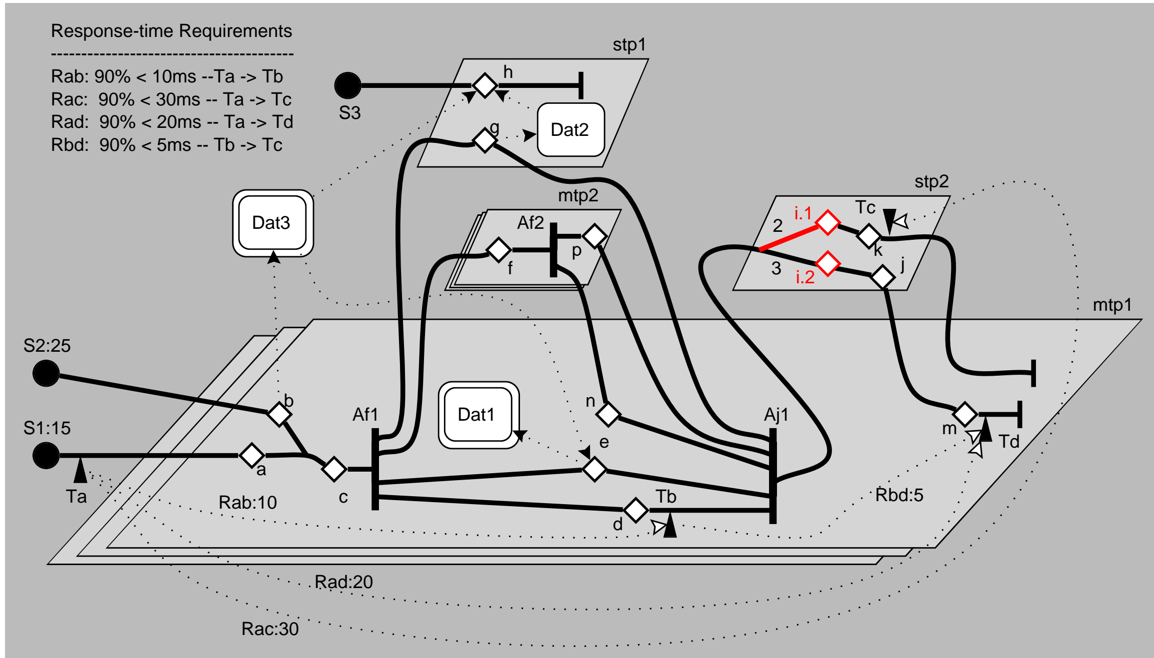

Figure 4.2: The UCM of Figure 4.1 with the OR fork moved

To illustrate this, when the token T described in Section 4.4.1 or a descendent finally reaches the OR fork in process *stp2* of Figure 4.1, then if the upper branch is chosen the current deadline of T will be discarded from $\mathcal{D}_T$ and $D_T(t)$ will change to 130.0ms.

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 one or more urgent deadlines in a token's collection to be discarded earlier, and hence allow a now more urgent token to execute earlier thereby yielding more efficient scheduling. Thus moving an OR fork more towards the start of a scenario can bind less urgent deadlines earlier, and is an application of Smith's Fixing-Point Principle [17]. For example, the OR fork in *stp2* of Figure 4.1 could be moved in front of activity i provided that a copy of activity i was placed on each branch immediately after the OR fork. These changes are shown in red in Figure 4.2. The changes allow an earlier extension of the current deadline of each token that follows the top branch.

Summary of the data stored by a token

The data stored by a token T can now be summarized, as follows:

1. The location of the token T on a path. T can be located on a path segment between two path elements, at an activity, or at an AND join. T would be located at an AND join if it was waiting for other tokens to join with. Its location is updated every time T moves along the path it is on.

2. A set of timestamps. A timestamp references a timestamp point and specifies a time value at which the token T crossed the timestamp point. The set is augmented every time T crosses a timestamp point.

3. The dictionary $\mathcal{D}_T$ of deadlines for T, keyed by response-time requirement. See Section 4.4.1. This dictionary gains one or more new deadlines when T crosses a timestamp point which begins one or more response-time intervals. The dictionary loses one or more existing elements

    - when T crosses a timestamp point which ends a response-time interval,
    - when T, possibly a newly cloned token, is placed on a branch of an AND fork for which some deadlines in $\mathcal{D}_T$ are not relevant,
    - when T is routed to a branch of an OR fork for which some deadlines in $\mathcal{D}_T$ are not relevant.

4. A dictionary $\mathcal{R}_T$, keyed by process, for which each value is a thread. See Section 4.2.3. When a token is sent to a process which is a key in $\mathcal{R}_T$, the token will be deposited in the *return* mailbox of the thread which is keyed to the process. When T leaves a process which is not already a key in the dictionary and to which T will later return, the dictionary is augmented to include as a value the thread which is actually sending T. At the time T is created, $\mathcal{R}_T$ is initialized to be empty.

5. A set of named data values. These values can be used to select which branch should be followed when T arrives at an OR fork as was discussed in section 3.2.3. These values can be added, modified, or deleted at activities.

4.4.7 Summary of when a thread's deadline may need to change

It will be helpful at this point to summarize in the following list when a thread's deadline may need to change.

1. The thread sends a token in a message to a mailbox (including at AND forks).
2. The thread's active token crosses a timestamp point which either starts or finishes a response-time requirement.
3. The thread's active token arrives at an OR fork.
4. A message arrives at a mailbox and is destined for the thread.
5. Another thread must wait when it tries to access a data object currently being accessed by the thread.

The first reason corresponds to a possible change in which token, if any, is active. This includes sending messages at an AND fork. The second and third reasons correspond to a possible change in the current deadline of the currently active token and possibly a change in which token is active. The thread's controller changes the thread's priority.

The fourth and fifth reasons typically correspond to priority inheritance. In these cases it is kernel primitives (messaging and locking) executing in the context of another thread, or perhaps an interrupt service routine at the beginning of a path, that changes the thread's priority.

4.5 Scheduling of devices other than central processor(s)

The above sections have explained the choice of EDF for the kernel's processor scheduling policy and ways in which thread behaviour co-ordinates with EDF. While a thread is using a processor, a more urgent thread can pre-empt it. Other classes of devices such as input/output (I/O) devices and special-purpose processors may not be so easily pre-empted. The UCMs presented in this thesis, and the PERFECT tool presented in the next section, use disks and digital signal processors (DSPs) to represent these other classes of devices. The scheduling of disks and DSPs in a virtual implementation is presented below.

Demand is placed on a disk as a number of accesses. Each access has a deadline equal to the current deadline of the token. The scheduler for a given disk maintains a single queue of all outstanding access requests. When one access finishes, the access with the earliest deadline is performed next. An access while in progress cannot be pre-empted.

This is not a very sophisticated scheduler. Better strategies for disk scheduling were described in Section 2.3.5.

An activity places demand on a DSP by requesting that a certain DSP procedure be initiated. Statistics on the invocation are known or estimated. Each request has a deadline equal to the current deadline of the requesting token. The scheduler for a DSP maintains a queue of outstanding requests. Once a DSP procedure has been started it cannot be pre-empted. When a scheduled DSP procedure finishes, the requested invocation with the earliest deadline is performed next.

4.6 Details of the simulator

A virtual implementation of specified scenarios is evaluated by simulation using the PERFECT tool which was constructed for the purposes of this thesis. The tool takes a UCM in the defined XML format [8], and simulates an automatically generated virtual implemen-

tation for a specified amount of time. Typically, the UCM file in XML format is generated by the UCM Navigator graphical UCM editing tool mentioned in section 3.4.

PERFECT is an object-oriented application written in Objective-C. Objects modelled include:

- The scenarios: a scenario is modelled as a directed graph of scenario-element nodes:
  - Path Starts
  - Activities
  - OR forks
  - OR joins
  - AND forks
  - AND joins
  - Path Ends

  A scenario also references other objects, representing:
  - Timestamp points
  - Response-time requirements
  - Shared data objects
- The hardware platform. This is made up of
  - A multiprocessor with a variable number of processors. Obviously choosing one processor allows a uniprocessor to be simulated. The speed of the processors can be specified.
  - One or more disks, as an example of I/O devices generally
  - One or more DSPs, as an example of special-purpose processors generally

- The concurrency architecture, made up of objects including:
  - Processes
  - Threads
  - Mailboxes
- Tokens
- Semaphores

When a concurrency architecture is determined for a set of scenarios, the scenarios record at which points certain paths cross between certain processes. Then to simulate the system, tokens are injected at path starts according to the specified stochastic arrival processes, and the entire behaviour of the system is automatically interpreted using the connections between the objects. The logic of the simulator is distributed throughout the objects. This has proven a very flexible tool architecture, allowing changes to simulation behaviour simply by replacing the definition of a limited number of objects.

An external event is modelled as being received by an interrupt service routine which sends a message to the appropriate mailbox.

A disk access is simply modelled in the simulator as taking an amount of time uniformly distributed between, for the experiments in this thesis, 1ms and 23ms. Disk accesses are non-deterministic due to the different amounts of head movement, etc. that may be necessary. The XML file format for UCMs provides a way to define devices and specify parameters for each device, so in principle disks and other devices can be modelled with as much accuracy as desired.

For the experiments in this thesis, the processor overhead for the various kernel primitives was based on results of the lmbench benchmarking suite [63]. The overhead results in Table 4.1 are for the Solaris operating system running on an ultrasparc processor operating at 143MHz.

Kernel primitive	Processor overhead in μs
Context switch	20
Send message	7
Receive message	7
Adjust priority (deadline)	7
Semaphore wait block	4
Semaphore wait noblock	1
Semaphore signal resume	4
Semaphore signal noresume	1

Table 4.1: Processor overhead for kernel primitives using Solaris on a 143MHz ultrasparc

PERFECT links with the PARASOL C-language simulation library [50], which is an execution-based library designed for prototyping and performance prediction of computer systems involving concurrent and possibly cooperating software elements. The PARASOL functions for thread management, inter-thread communication, and thread synchronization are roughly aligned with primitives offered by the Mach kernel.

Running on a 32-bit machine, the 2^{32} (C-language "int") priority levels of PARASOL's pre-emptive priority scheduler were used to implement EDF scheduling by restricting deadlines to a resolution of microseconds and mapping the deadlines directly to priorities. This allowed over 71 minutes of time to be simulated before deadlines would overflow. Maintaining microsecond resolution, running on a 64-bit machine with 64-bit "int"s would allow almost 600,000 years of time to be simulated.

For output, PERFECT reports statistics including:

- The mean response time and fraction of "successful" responses for each type of response.

- The utilization of each device instance.
- The amounts observed of different types of concurrency-related overhead: context switching, messaging, deadline adjustment, and semaphore operations used for protecting access to shared data objects.
- The metrics which are presented in the next chapter.

In order to estimate confidence intervals for each simulation run, PERFECT uses PARASOL's *ps_block_stats* call, which blocks the simulation run into quasi-independent blocks and applies a Student-t analysis. In this thesis, each simulation is run for a duration such that valid comparisons can be made between architectures. Where confidence intervals are not explicitly stated, all digits for a reported statistic are significant according to a 95% confidence interval. For example, a result 8.544 with a 95% confidence interval of 0.02 will be reported as 8.54.

4.7 Simple validation of the simulator

This section presents some of the experiments which were performed to validate the evaluation capability of PERFECT. In these experiments, PERFECT's simulation results are compared to results obtained from analysis.

4.7.1 A M/M/1 queueing system

A simple system which can be thoroughly analyzed [11] is a M/M/1 queueing system with First-Come, First-Served (FCFS) scheduling. In an M/M/1 queueing system, Poisson arrivals are processed by a single server which has an exponentially distributed service time. Figure 4.3 shows a UCM which corresponds to a M/M/1 queueing system. External events arrive at an average rate of $\lambda = 0.3$ arrivals/second (the average interarrival time is 3.333

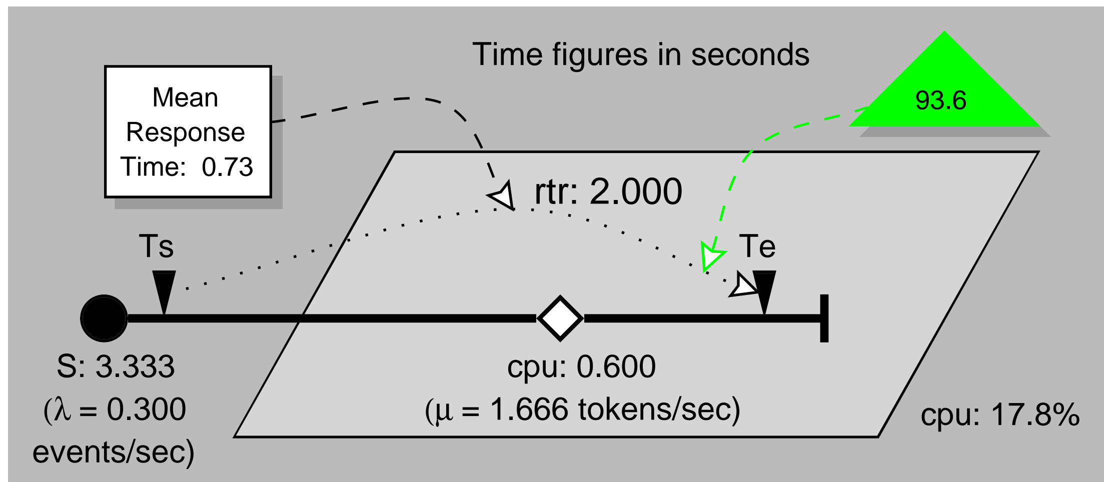

Figure 4.3: A UCM corresponding to an M/M/1 queuing system

	PERFECT simulation with 95% confidence	Analysis
Mean Response Time	(0.73 ± 0.01) secs	$^{1}/_{(\mu-\lambda)} = 0.7317$ secs
Responses completing within 2secs	$93.6\% \pm 0.6\%$	$F(x = 2\text{secs}) = 1 - e^{-(\mu-\lambda)x} = 93.50\%$
cpu Utilization	$17.8\% \pm 0.5\%$	$^{\lambda}/_{\mu} = 18.00\%$

Table 4.2: Comparisons between analysis and results of PERFECT simulation for the UCM of Figure 4.3

seconds), and each token requires an average of 0.6 seconds to be served (the average service rate is $\mu = 1.666$ tokens/second). Because all responses have the same response-time requirement, PERFECT's EDF scheduling will produce the same schedule as an FCFS scheduler. One type of response, with a specified delay value of 2.000 seconds, is drawn on the figure, and also indicated on the figure in a green triangle is the simulation result that 93.6% of responses finish within the specified delay. The use of triangular symbols to display such results is discussed further in Section 5.1.

For the UCM of Figure 4.3, Table 4.2 compares the results obtained by analysis with the results of PERFECT simulation for the following quantities:

- The mean response time.
- The fraction of responses expected to complete within 2 seconds. This is simply obtained from the response-time distribution.
- The cpu utilization.

Note that the analytical results fall nicely within the 95% confidence interval of the simulation results.

4.7.2 A simple product-form queueing network

In addition to measuring the fraction of responses which complete within the specified delay for a given response-time requirement, PERFECT can measure the mean response time as was demonstrated above for the M/M/1 queueing system. We can easily get analytical results, including the mean response time, for product-form queueing networks [20] such as the one shown in Figure 4.4.

Figure 4.5 shows a UCM which corresponds to the queueing network of Figure 4.4. For this UCM, a simulation by PERFECT reported a mean system response time of 8.8 seconds with a 95% confidence interval of 0.2 seconds. Analysis of the queueing network gives a

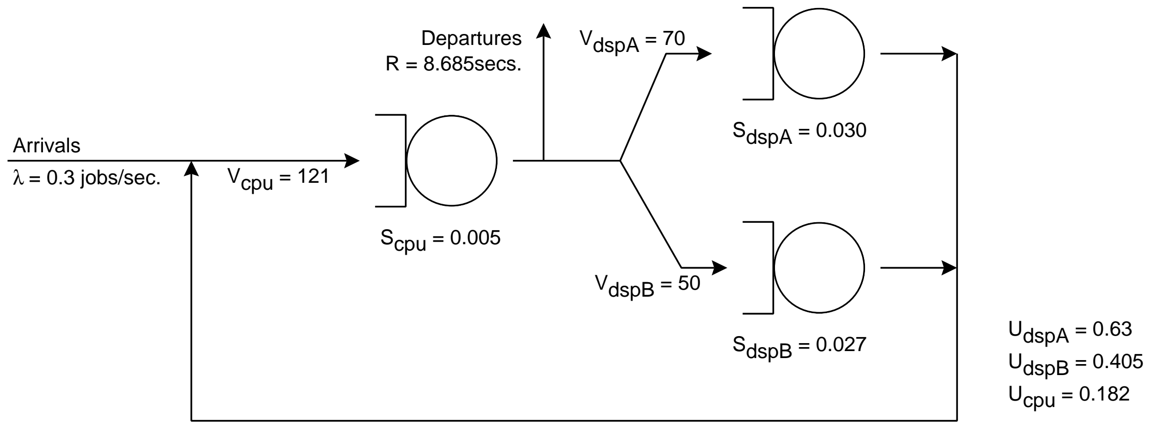

Figure 4.4: A simple product form queueing network

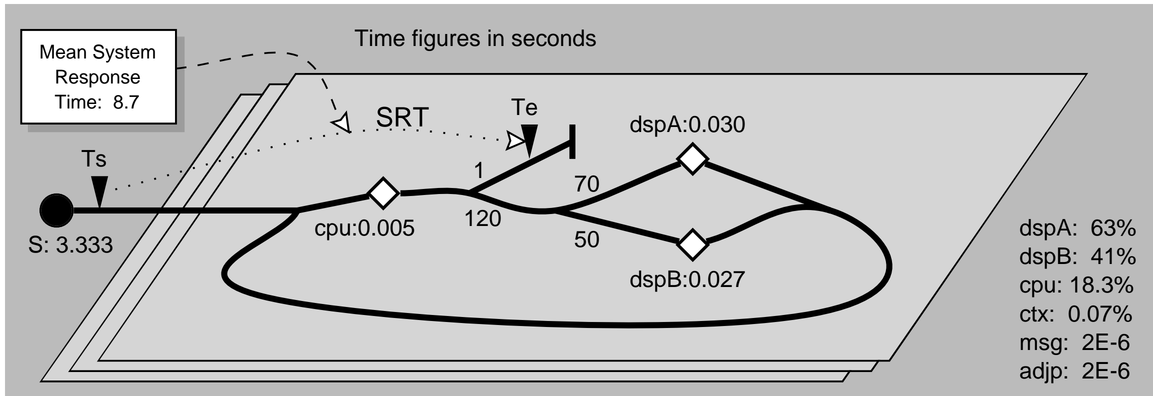

Figure 4.5: A UCM corresponding to the product form queueing network of Figure 4.4

	PERFECT simulation with 95% confidence interval	Analysis
Mean System Response Time	(8.8 ± 0.2) secs	8.69 secs
dspA Utilization	63.1% ± 0.9%	63.00%
dspB Utilization	40.6% ± 0.5%	40.50%
cpu Utilization	18.3% ± 0.3%	18.15%

Table 4.3: Comparisons between analysis and results of PERFECT simulation for the UCM of Figure 4.5

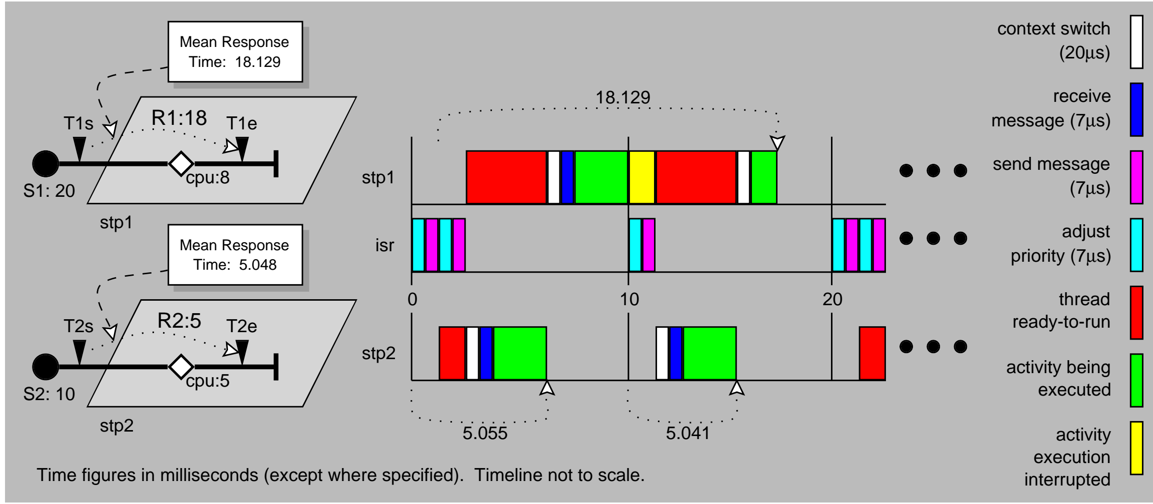

Figure 4.6: A UCM with equally spaced arrivals and a timeline of its execution. Note the pre-emption which occurs.

mean system response time of 8.69 seconds, well within the simulation confidence interval. When the virtual implementation is simulated, the cpu must execute overhead operations, such as messaging, in addition to the cpu-intensive activity. However, the resource demands of activities were made large enough to dwarf the contribution of the overhead operations.

In Table 4.3, the mean system response time and utilization figures obtained by analysis are compared to those obtained by PERFECT simulation.

4.7.3 Validating pre-emption and overhead

Consider the UCM on the left side of Figure 4.6. The "shorter", more urgent scenario starting at *S2* is triggered twice as often as the longer scenario starting at *S1* and should cause pre-emption to occur. All interarrival times are deterministic.

By following the rules of virtual implementation, a timeline of expected execution is constructed on the right side of the figure. When an event is recognized by an (assumed) interrupt service routine (ISR), the ISR will set the deadline of the destination thread, and then send a message to that thread. Assuming that the destination thread is not

already running, as is always the case for this UCM, the processor will eventually switch to the context of the destination thread and the message will be received. For half of the invocations at *S2*, the thread of *stp1* will be running and will be pre-empted.

By adding the time spent executing the activities to the various overhead times, the expected response times have been calculated and indicated on the timeline. Because of the cyclical nature of the example, response times for response *R2* will alternate between 5.055ms and 5.041ms. The mean response time reported by PERFECT for response *R1* is 18.1289ms, exactly as it should be. The mean response time reported for response *R2* is 5.048ms, the mean of the two possible response times for that response.

Chapter 5

Diagnostic Metrics

This chapter describes three problems which can degrade the performance of a concurrency architecture. For each problem, an example which illuminates the problem is presented, and a metric to measure the extent and identify the locations of the problem is proposed. Combining a knowledge of which responses are determined to be late (using the evaluation technique of the previous chapter), and information on concurrency problems which are present using the diagnosis techniques of this chapter, the designer can expose the limitations of a concurrency architecture and propose improvements. This is demonstrated for each of the examples.

5.1 Problem 1: inversion due to inadequate concurrency

The term inversion here will be applied more generally than priority inversion but it has the same significance. Inversion occurs when, due to inadequate concurrency, more urgent work must wait for less urgent work [60]. Inversion can occur in a virtual implementation under the following circumstances:

- Synchronization – An urgent token might have to wait for a less-urgent token which is accessing a protected data object needed by the urgent token.

- Messaging – An urgent token might have to wait to be received by a thread which is processing a less urgent token.

As was explained in section 4.4.3, priority inheritance is used in a virtual implementation to reduce unbounded inversion but some bounded inversion still remains under the above circumstances. In a simple case, the bound on the inversion is then the amount of time it takes the less-urgent token to finish with the resource in contention (a datastore or a thread). It is difficult to eliminate this remaining bounded inversion when it occurs during synchronization, but there are often avenues for eliminating bounded inversion when it occurs during messaging.

For the messaging used in the virtual implementations, inversion can occur either at a primary mailbox of a single-threaded process, or at any return mailbox. An example of inversion at a primary mailbox is shown in Figure 5.1. Here, there are three paths of processing. The first two paths have a cpu demand of 50μs, are invoked according to a Poisson process with an average interarrival time of 500μs, share a data object $ds1$, and require that 90% of responses complete in less than 300μs. The third path has a cpu demand of 10μs, an average interarrival time of 500μs, and requires that 90% of responses complete in less than 50μs. With all these paths assigned to one single-threaded process, we get very high success ratios for paths 1 and 2, but only a 76% success ratio for path number 3. The "success ratio" for a type of response is the ratio of responses which complete within the specified delay to all responses of that type. Success ratios are indicated on UCM diagrams in triangular symbols with a dashed arrow pointing to the type of response. Success ratios which exceed the required percentage (90% in this thesis) are shaded green, while success ratios which fall below the requirement are shaded red. Success ratios which come close to meeting the required percentage are shaded pink.

The poor success ratio for path number 3 could be due to

- its combination of average interarrival time, cpu demand, and specified delay value

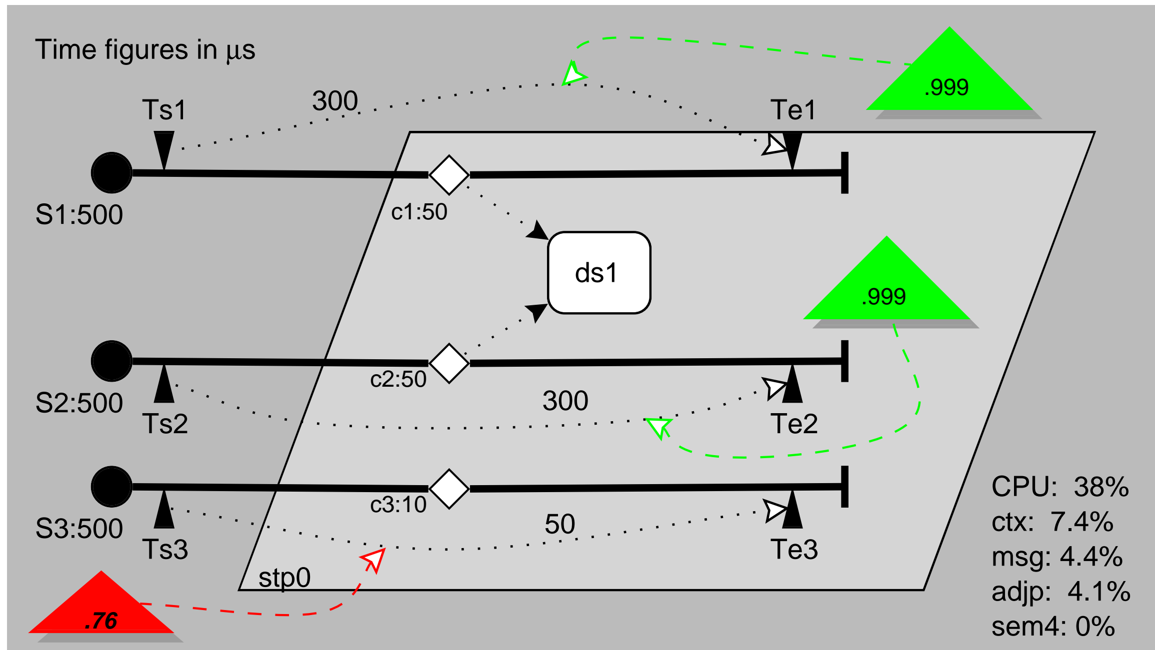

Figure 5.1: Example showing inversion

(too small average interarrival time, too small specified delay value and/or too large cpu demand) – i.e. an infeasible specification

- relatively urgent tokens arriving at *S3* too often having to wait for the thread to finish processing a less-urgent token on one of the top two paths – i.e. inversion. The thread will not receive the relatively urgent token until it has finished its current processing,

Section 5.4.1 describes a metric that measures the amount of inversion in this architecture. If inversion is shown to be a significant problem, then a separate process can be created for the third path. The thread of the separate process could pre-empt the processor when its deadline is the most urgent. An alternative improvement would be to multithread the process.

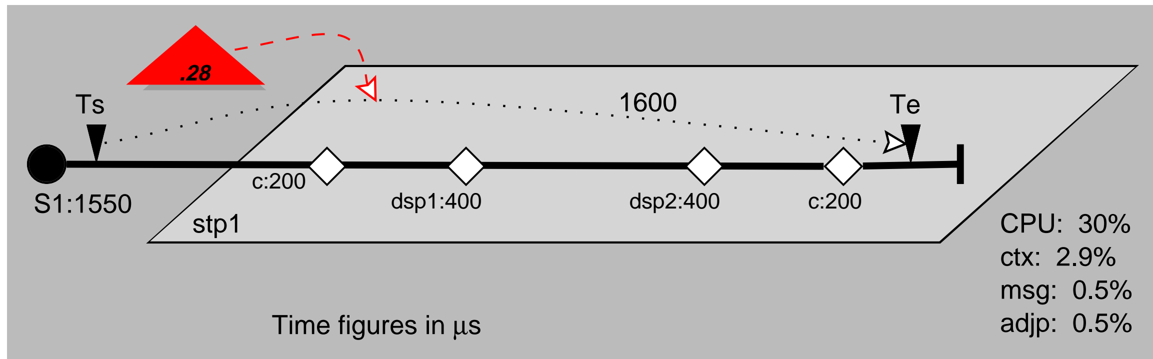

Figure 5.2: Excess serialization due to only one token being able to travel along a given path segment at a given time

5.2 Problem 2: inefficient scheduling of devices due to inadequate concurrency

If the scenarios make demands on multiple devices, then it should be possible for multiple tokens to make progress simultaneously. Appropriately placed concurrency is required in the software architecture in order to exploit this potential. In addition to having tokens on different scenarios concurrently making progress, it is also possible for related or unrelated tokens on the same scenario to make concurrent progress. Different external events triggering a scenario will result in unrelated tokens, and one or more AND forks on a scenario will result in related tokens. If there is inadequate concurrency, then the progress of different tokens will end up being serialized.

An example of excessive serialization due to only one token being able to travel along a given path segment at a given time is shown in Figure 5.2. Here, the activities along a path use three devices: one cpu and two DSPs. A response is defined over the length of the path, and when the path is assigned to one single-threaded process, the response achieves a 28% success ratio. A likely problem with this architecture is inefficient use of devices due to serialized executions of the path. Because there is only one thread, only one device can be active at any one time.

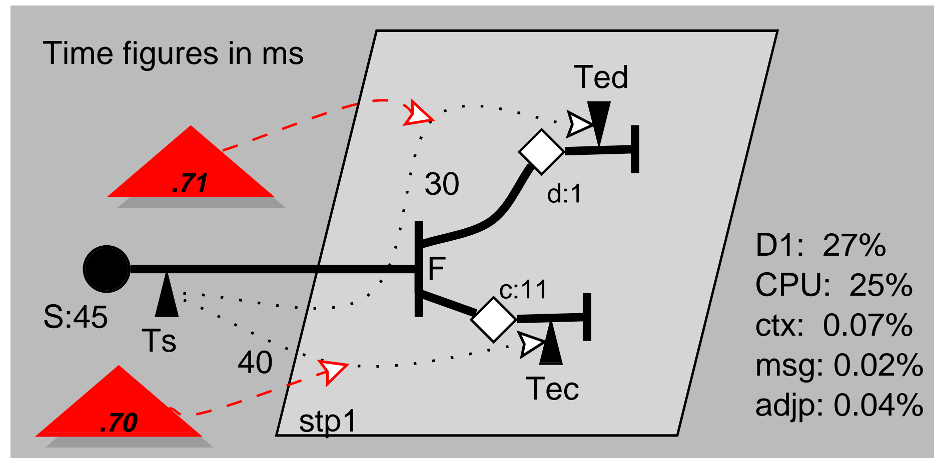

Figure 5.3: Excess serialization after an AND fork

An example of excessive serialization after an AND fork is shown in Figure 5.3. Here, a single-threaded process is responsible for executing both branches of the AND fork: one which uses a disk and one which uses the cpu intensively. There are two responses, each starting when an external event occurs: one ending after the activity which makes one disk access and another ending after the activity which uses 11ms of processor time. Although the cpu is lightly loaded, evaluation reveals a success ratio of 70% for the response ending after the cpu-intensive activity. In this example it can be readily determined by inspection that serialization of the branches is limiting the performance of the architecture. However, with more complicated UCMs it is not always true that such serialization significantly degrades performance because there may be many other tokens competing for devices. In sections 5.4.2 and 5.4.3 metrics are proposed which will help to gauge the degree to which the efficient scheduling of devices is being limited. The metrics may suggest that an architectural change be considered.

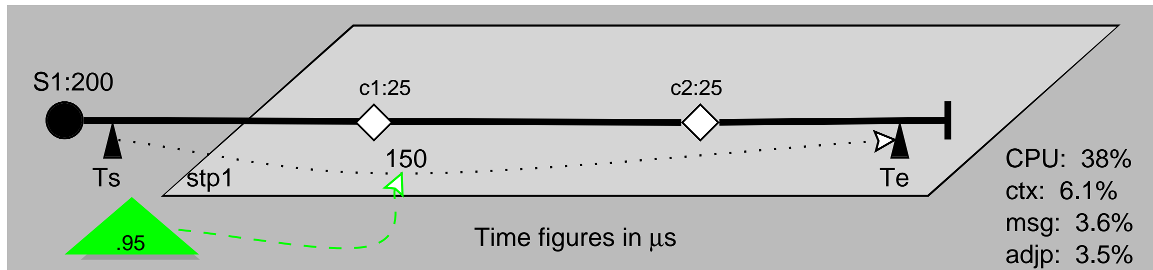

Figure 5.4: A very simple UCM

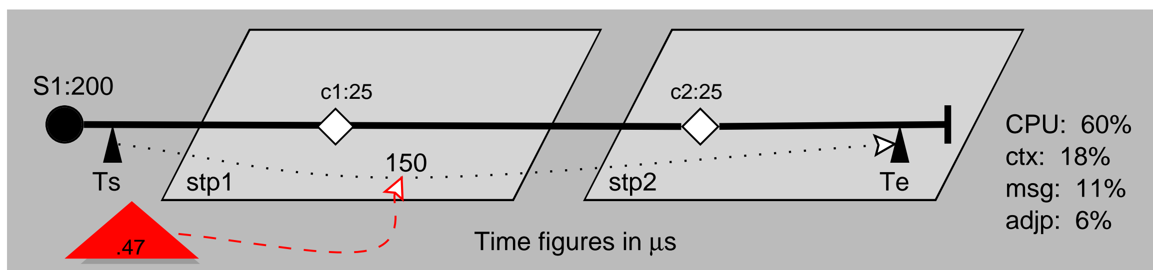

Figure 5.5: The very simple UCM above with a second single-threaded process added

5.3 Problem 3: excessive overhead due to inappropriate concurrency

Too much concurrency can cause undue overhead, thus slowing down responses. Figure 5.4 shows a simple Use Case Map where 95% of responses complete within the specified delay. If we change to a two-process architecture the success rate drops to 47%. See Figure 5.5. It is easy to see why the two-process architecture gives poorer performance. Both processes use only the cpu and hence only one process can be "useful" at a time. Because only one path uses the processes, the processes will always execute in alternation. Thus for every response, two context switches are necessary as well as an extra message transfer and an extra priority adjustment. In the single-process architecture, no context switches may be necessary when a burst of arrivals is being processed.

To help to balance the overhead introduced by concurrency with the benefits of concur-

rency, the fraction of cpu cycles used for different categories of concurrency-related overhead is reported in the UCM diagrams in the lower right-hand corner.

- "ctx" refers to context switching
- "msg" refers to message passing
- "adjp" refers to adjusting thread priority
- "sem4" refers to semaphore operations used in protecting access to data objects.

Device utilizations are also listed.

5.4 Formal metrics for detecting problems

As explained in the previous section, we are measuring the different types of overhead, and therefore can have some indication of when there is excessive overhead, but how can we detect when there are problems in the architecture leading to inversion or inefficient use of devices? To detect these problems we introduce some metrics which indicate the presence of the problems.

5.4.1 Inversion metric

Figure 5.1 shows a UCM where we suspect inversion is a problem. That is, a relatively urgent token created at *S3* might have to wait for the thread to finish processing a less urgent token on one of the top two paths. We want a metric to measure the extent to which this actually happens. To achieve this we can create a set of metrics which measures the time-averaged number of tokens which are deadline-inverted at each point where a path crosses into a process. To help to interpret the results we will report the measurement at a point, s, as a fraction of the time-averaged number of tokens waiting at s.

The metric is thus specific to a point where a path crosses into a process. The point is identified by naming the neighbouring path elements between which the point is located, as illustrated in Figure 5.6. There are three cases. In the first case a path crosses for the first time into a single-threaded process. Figure 5.1 shows examples of this case. In the second case a path crosses for the first time into a multi-threaded process. If there is no limit to the number of threads, as is the case in this work, then deadline inversion should not occur because an urgent token would get its own thread which could pre-empt threads doing less urgent work. Thus in this work we will not define the inversion metric for this case.

In the third case a path returns to a process. In this case a token returning to the process will be received by a particular thread. If the process is multi-threaded, then the token will be sent to exactly one of those threads. The path segment between *c2* and *c3* in Figure 5.6 shows the case of inversion at a path returning to a process. If a token arrives at *mtp1* along the path from activity *c2* to activity *c3* before the sibling token arrives at the AND join then the token will be deadline inverted.

The inversion metric will be measured in the first and third cases where a token will wait to be received by a particular thread (the only thread for the first case, and the known thread in the third case).

1. Definition: deadline inverted token.

Consider a token T waiting at time t to be received by a thread R which has an active token and is in process P.

- As defined in section 4.4.1, let $D_T(t)$ be, at time t, the deadline for token T.
- Let $D_{A_R}(t)$ be the deadline of the active token of thread R at time t.

T is deadline inverted at time t if $D_T(t) < D_{A_R}(t)$

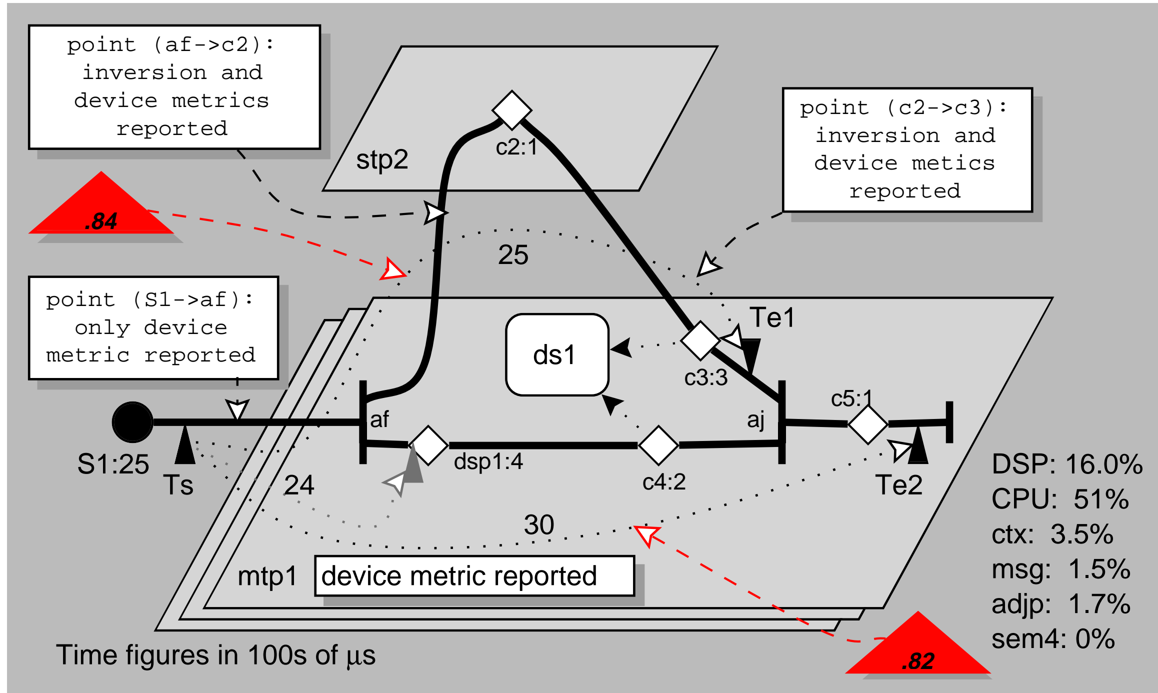

Figure 5.6: Suspected deadline inversion where a path returns to a process

2. Inversion measures at a point s

Consider a point s at which a path crosses into a process.

- Let $\mathcal{I}_{s,R}(t)$ be the set of deadline-inverted tokens waiting at a point s and at time t to be received by thread R.
- Let $\mathcal{W}_{s,R}(t)$ be the set of tokens waiting at a point s and at time t to be received by thread R.

Then, at time t, the total number of deadline-inverted tokens waiting at point s to be received by any thread is:

$$I_s(t) = \sum_{\text{all} R \text{in} P} |\mathcal{I}_{s,R}(t)|$$

The total number of tokens waiting there and at that time is:

$$W_s(t) = \sum_{\text{all} R \text{in} P} |\mathcal{W}_{s,R}(t)|$$

Over a period of observation of length L and starting at time t_1, the time-averaged number of tokens waiting at point s is:

$$W_s(t_1, L) = \frac{1}{L} \int_{t_1}^{t_1+L} W_s(t)dt$$

The fraction of those tokens which are deadline inverted is:

$$I_s^* = \frac{1/L \int_{t_1}^{t_1+L} I_s(t)dt}{W_s(t_1, L)} = \frac{\int_{t_1}^{t_1+L} I_s(t)dt}{\int_{t_1}^{t_1+L} W_s(t)dt} \leq 1$$

The measures were estimated over a simulation run, taking care to discard initial transients and to obtain sufficient accuracy of estimates of the measures. Confidence intervals were measured, and runs were extended to make the comparisons of measure values significant. Reported values will be referred to as W_s and I_s^* or, if the point on a path is understood, simply W and I^*. A value of $I^* = 0$ at a point means that there is no inversion observed there, while a value of 1 or 100% means that all tokens at that point were inverted all the time.

The result of applying the inversion metric to the Use Case Map of Figure 5.1 is shown in Figure 5.7. The metric value is shown in white rectangles for the three points where paths cross into the process. For the path with the urgent requirement, there is a time-average of 0.033 tokens waiting to be received ($W_{(S3 \rightarrow c3)} = 0.033$). Of those, 48% are deadline-inverted ($I^*_{(S3 \rightarrow c3)} = 48\%$). Note that for the two paths with the less-urgent requirement that no waiting tokens are deadline inverted.

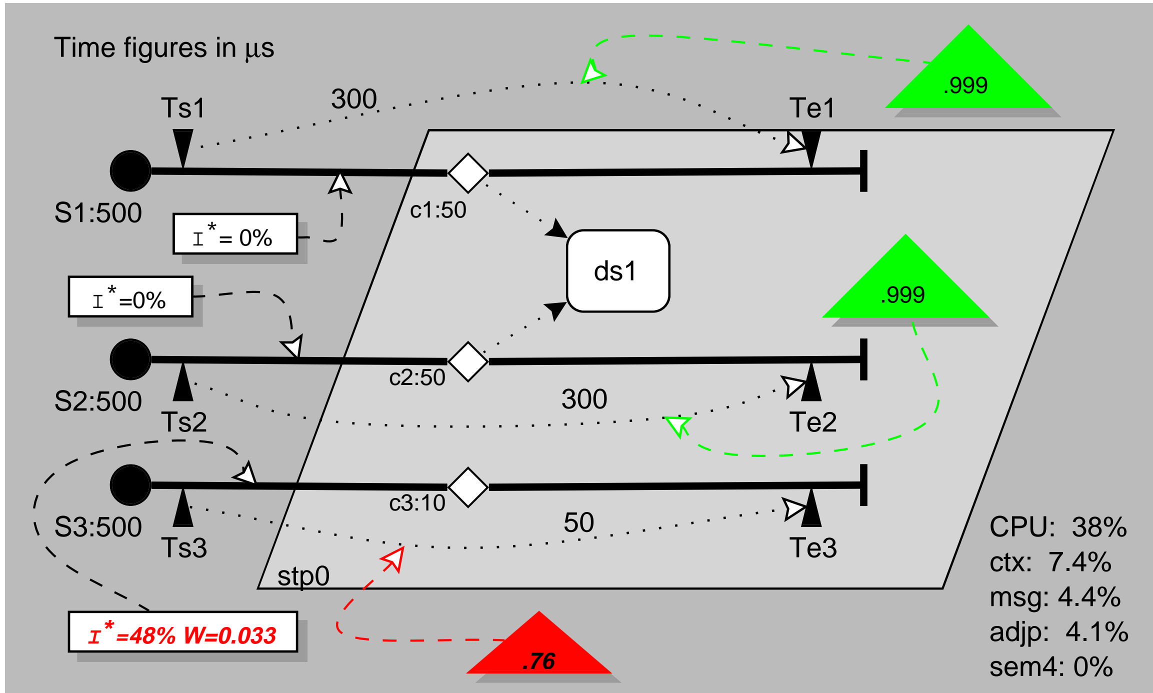

Figure 5.7: Inversion metric applied to the Use Case Map of Figure 5.1

5.4.2 External device metric

Figure 5.2 showed a UCM where we suspected a problem of inefficient use of devices due to serialization of execution. We need metrics to measure the degree to which any token is waiting (either in a process or in a mailbox external to a process) and devices it needs are idle or are doing less urgent work. This section will introduce a metric for tokens waiting external to a process. A metric for tokens waiting internal to a process will be introduced in the next section.

Consider a software system which places demand on a set $\mathcal{C}$ of classes of devices. Consider in that system a token T arriving at a process P at point s along a path. Eventually T will be received by a thread of P. T and any possible descendent tokens may need to use a set $\mathcal{N}_{T,s}$ of devices before either leaving the thread or being destroyed at path-ends. If OR forks are present then the token and descendents may only use a subset of $\mathcal{N}_{T,s}$. Some

of these devices will be used at the earliest deadline of the token, but other devices will not be used until a later deadline of the token is applicable (because the token will have already finished the response associated with its current earliest deadline).

- Let $\mathcal{D}_{T,s,d}$ be the computed set of deadlines at which token T waiting at point s might, while passing through the process it is entering, be scheduled on a device of class d.
- Let $D_{T,s,d}$ be the earliest deadline at which token T waiting at point s might, while passing through the process it is entering, be scheduled on a device of class d,

$$D_{T,s,d} = \min D \in \mathcal{D}_{T,s,d}$$

- Let $D_d(t)$ be the earliest deadline of all the threads (in any process) using or waiting to use device d at time t. $D_d(t)$ equals "the end of time" if one or more devices of class d are idle at time t. Note that multiple threads can be using a processing device in a multiprocessor. Also, for a non pre-emptable device, a thread waiting to use the device may have a higher priority than the thread actually using the device.

A device d in $\mathcal{N}_{T,s}$ is said to be potentially poorly scheduled (PPS) at time t with respect to token T if $D_d(t) > D_{T,s,d}$.

Let $\mathcal{W}_s(t)$ be the set of tokens at time t which have arrived at process P at point s but have not yet been received by the process (i.e. are waiting). A device d is said to be potentially poorly scheduled at time t with respect to $\mathcal{W}_s(t)$ if the device is potentially poorly scheduled at time t with respect to any token in that set.

Let $\mathcal{P}_s(t)$ be the set of all potentially poorly scheduled devices at time t with respect to $\mathcal{W}_s(t)$,

$$\mathcal{P}_s(t) = \{d | D_d(t) > D_{T,s,d}, T \in \mathcal{W}_s(t)\}$$

Over a period of observation of length L and starting at time t_1, the time-averaged

number of devices which may be needed by tokens which are waiting at point s is:

$$N_s(t_1, L) = \frac{1}{L} \int_{t_1}^{t_1+L} \left| \bigcup_{T \in \mathcal{W}_s(t)} \mathcal{N}_{T,s} \right| dt \leq |\mathcal{C}|$$

$N_s(t_1, L)$ will be referred to as time-averaged external device need at point s.

The fraction of that time-averaged device need which is potentially poorly scheduled is:

$$P_s^*(t_1, L) = \frac{\frac{1}{L} \int_{t_1}^{t_1+L} |\mathcal{P}_s(t)| \, dt}{N_s(t_1, L)} = \frac{\int_{t_1}^{t_1+L} |\mathcal{P}_s(t)| \, dt}{\int_{t_1}^{t_1+L} \left| \bigcup_{T \in \mathcal{W}_s(t)} \mathcal{N}_{T,s} \right| dt} \leq 1$$

Simulations are initialized with no tokens travelling along the paths. As stated earlier, t_1 should be chosen so that transient behaviour becomes insignificant, and L should be chosen to give enough precision to allow meaningful comparison of alternative architectures. Reported values will be referred to as N_s and P_s^* or, if the point on a path is understood, simply N and P^*. N_s is the average number of devices that are usable by tokens at external point s. A large value indicates a strong diversity of device demands from tokens at that point, which in turn indicates strong potential concurrency within the process, for those tokens. A value of zero for P_s^* means that no device is ever potentially poorly scheduled with respect to tokens at point s, while a value of 1 or 100% means that each device needed by tokens which wait at point s is potentially poorly scheduled with respect to at least one token waiting at point s, whenever there is at least one token waiting there.

The result of applying the inversion and external device metrics to the Use Case Map of Figure 5.2 is shown in Figure 5.8. The metrics are shown in a white rectangle. While none of the waiting tokens are deadline inverted, there is a time-averaged device need $U_{(S1 \to c)}$ of 1.6, of which $P^*_{(S1 \to c)} = 65\%$ is potentially poorly scheduled.

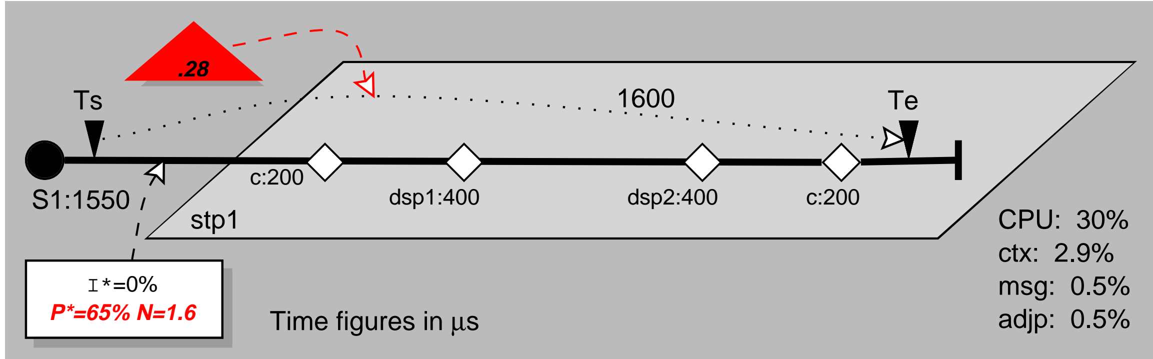

Figure 5.8: External device metric applied to the Use Case Map of Figure 5.2

5.4.3 Internal device metric

This section defines a metric similar to the external device metric, but for devices needed by tokens waiting internally to a process. The metric is measured for each process.

Consider a ready-to-proceed token T in thread R of process P and waiting at point s. See section 4.4.2 for a discussion of when tokens are waiting and ready-to-proceed. The token and any possible descendent tokens may use a set $\mathcal{U}_{T,s}$ of devices before either leaving the thread, being destroyed at path-ends, and/or joining with the currently active token of the thread or one of its descendants. Some of these devices may be used at the earliest deadline of the token, but other devices will not be used until a later deadline of the token is applicable (because the response associated with the current earliest deadline of the token will have finished).

A device d in $\mathcal{U}_{T,s}$ is said to be potentially poorly scheduled at time t with respect to token T if $D_d(t) > D_{T,s,d}$. $D_d(t)$ and $D_{T,s,d}$ were defined above in section 5.4.2.

A device d is said to be potentially poorly scheduled at time t with respect to a thread R if the device is potentially poorly scheduled at time t with respect to any token which is in R and is ready-to-proceed, i.e. in set $\mathcal{Q}_R(t)$.

Let $\mathcal{P}_R(t)$ be the set of all potentially poorly scheduled devices with respect to thread

R at time t,

$$\mathcal{P}_R(t) = \{d | D_d(t) > D_{T,s,d}, T \in \mathcal{Q}_R(t)\}$$

Let $\mathcal{U}_R(t)$ be the union of all $\mathcal{U}_{T,s}$ at time t for which T is a token in the internal ready queue of R,

$$\mathcal{U}_R(t) = \bigcup_{T \in \mathcal{Q}_R(t)} \mathcal{U}_{T,s}$$

Over a period of observation of length L and starting at time t_1, the time-averaged number of devices which may be needed by ready-to-proceed tokens waiting in threads of process P is:

$$N_P(t_1, L) = \frac{1}{L} \int_{t_1}^{t_1+L} \left| \bigcup_{\text{all} R \text{in} P} \mathcal{U}_R(t) \right| dt \leq |\mathcal{C}|$$

$N_P(t_1, L)$ will be referred to as time-averaged internal device need for process P.

The fraction of that time-averaged internal device need which is potentially poorly scheduled is:

$$P_P^*(t_1, L) = \frac{\displaystyle\int_{t_1}^{t_1+L} \left| \bigcup_{\text{all} R \text{in} P} \mathcal{P}_R(t) \right| dt}{\displaystyle\int_{t_1}^{t_1+L} \left| \bigcup_{\text{all} R \text{in} P} \mathcal{U}_R(t) \right| dt} \leq 1$$

Following the convention established for the external device metric, reported values will be referred to as N_P and P_P^* or, if the process is understood, simply N and P^*. Since a process must contain an AND fork in order to have internal device need, the metric will not be reported for any process without an AND fork.

The result of applying the metrics to the Use Case Map of Figure 5.3 is shown in Figure 5.9. The internal device metric is shown at the bottom of the process next to the process' name. There is a time-averaged internal device need N_{stp1} of 0.28, of which an impressive $P^*_{stp1} = 99.8\%$ is potentially poorly scheduled. Also note that there is a time-

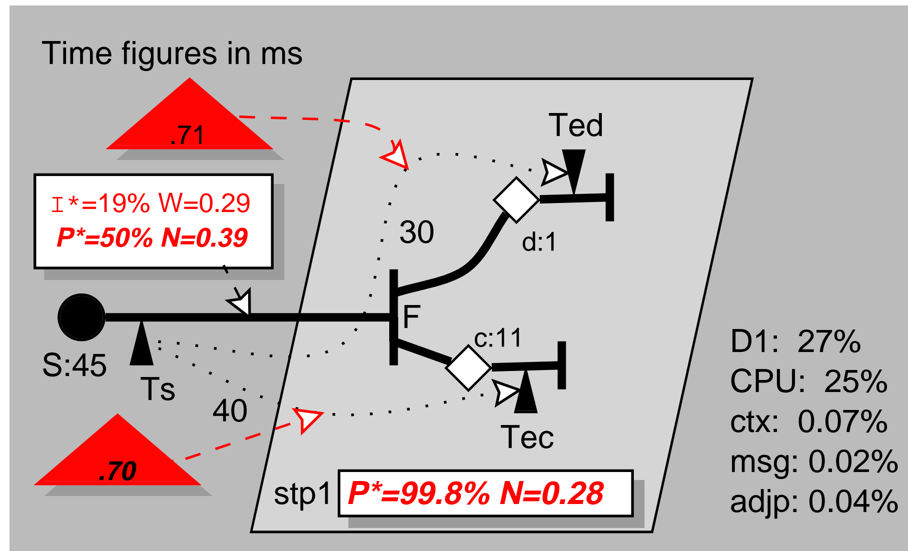

Figure 5.9: Internal device metric applied to the Use Case Map of Figure 5.3

averaged external device need $N_{(S1 \rightarrow F)}$ of 0.39, of which $P^*_{(S1 \rightarrow F)} = 50\%$ was potentially poorly scheduled (2 devices are needed by a waiting token, and 1 of those devices will be "well-scheduled" processing a previous token). Finally, of the time-average of 0.29 tokens waiting to be received by the process, 19% are deadline inverted (a token which arrives less than 10ms after the previous token will be deadline inverted when the previous token is using the cpu).

5.5 Using the metrics to guide design improvement

In section 5.4 we defined three metrics to help identify problems in architectural designs and applied the metrics to the motivating Use Case Maps from the first sections of this chapter. Considering the metric results, we will now try to improve the concurrency architectures for the examples.

5.5.1 Inversion examples

The metric results for the inversion example was shown in Figure 5.7. Call that case INV1.

Case INV2

To remedy the inversion on the third path of case INV1, an alternate architecture with the third path in a separate process is proposed. The two-process architecture and its evaluation are shown in Figure 5.10. Process *stpB* will pre-empt *stpA* if necessary when a token is created at *S3*. Now none of the tokens are deadline inverted, and the success ratio of the third response has increased from 76% to 98%. Because process *stpA* now experiences pre-emption, the success ratios for the top two responses have decreased to 99.7% from virtually 100%. The new architecture has more overhead, especially context switching which has risen from using 7.4% to 8.9% of cpu power.

Case INV3

It is important to understand that reducing inversion does not in itself guarantee that any response will have a greater success ratio. The best concurrency architecture will be obtained by balancing the benefits of greater concurrency with the overhead it introduces. To show this, Figure 5.11 restates the UCM of Figure 5.7 with different parameters that cause greater overhead. Specifically, all average interarrival times, cpu demands of activities, and specified delays are halved. This leads to a doubling of the total cpu cycles spent on messaging (the number of cpu cycles spent on a given overhead operation remains the same). Context switching and priority adjustment overhead have increased significantly. This means that for a given response, overhead now has a greater relative significance which explains the smaller success ratios for the responses.

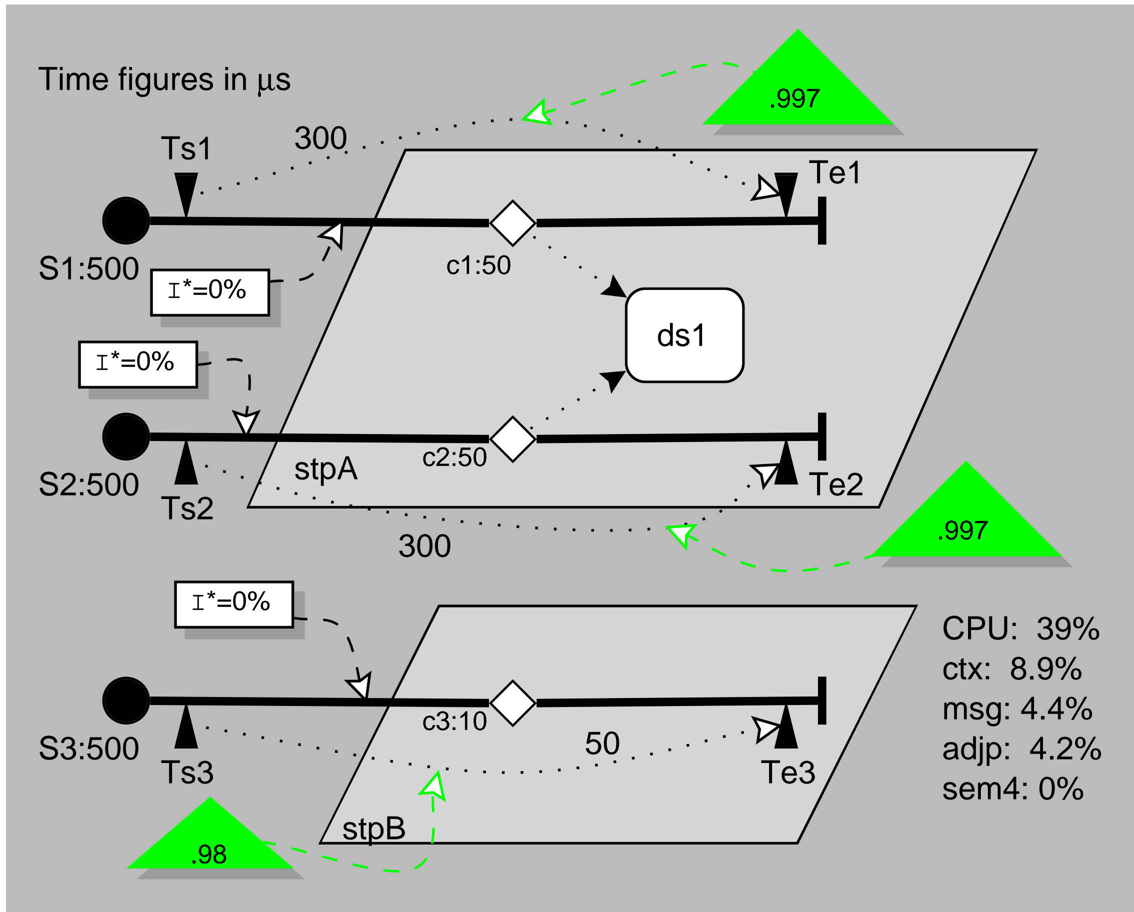

Figure 5.10: Case INV2 – An architecture which removes inversion

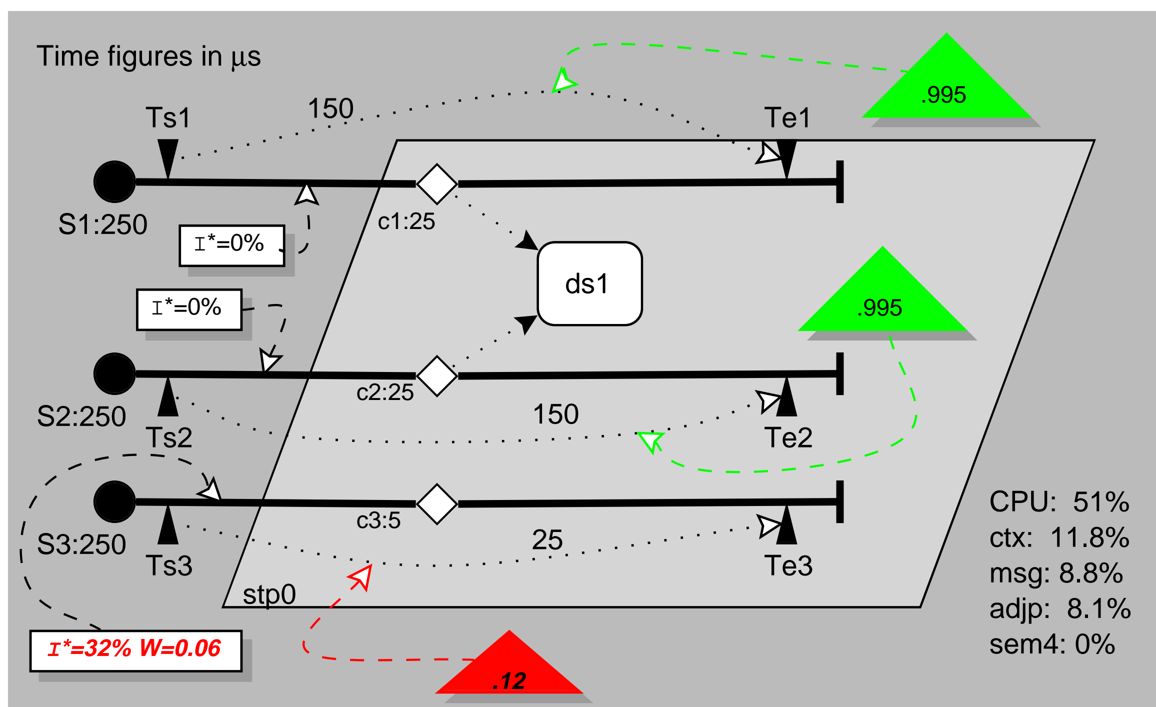

Figure 5.11: Case INV3 – UCM of Figure 5.7 with changed time parameters

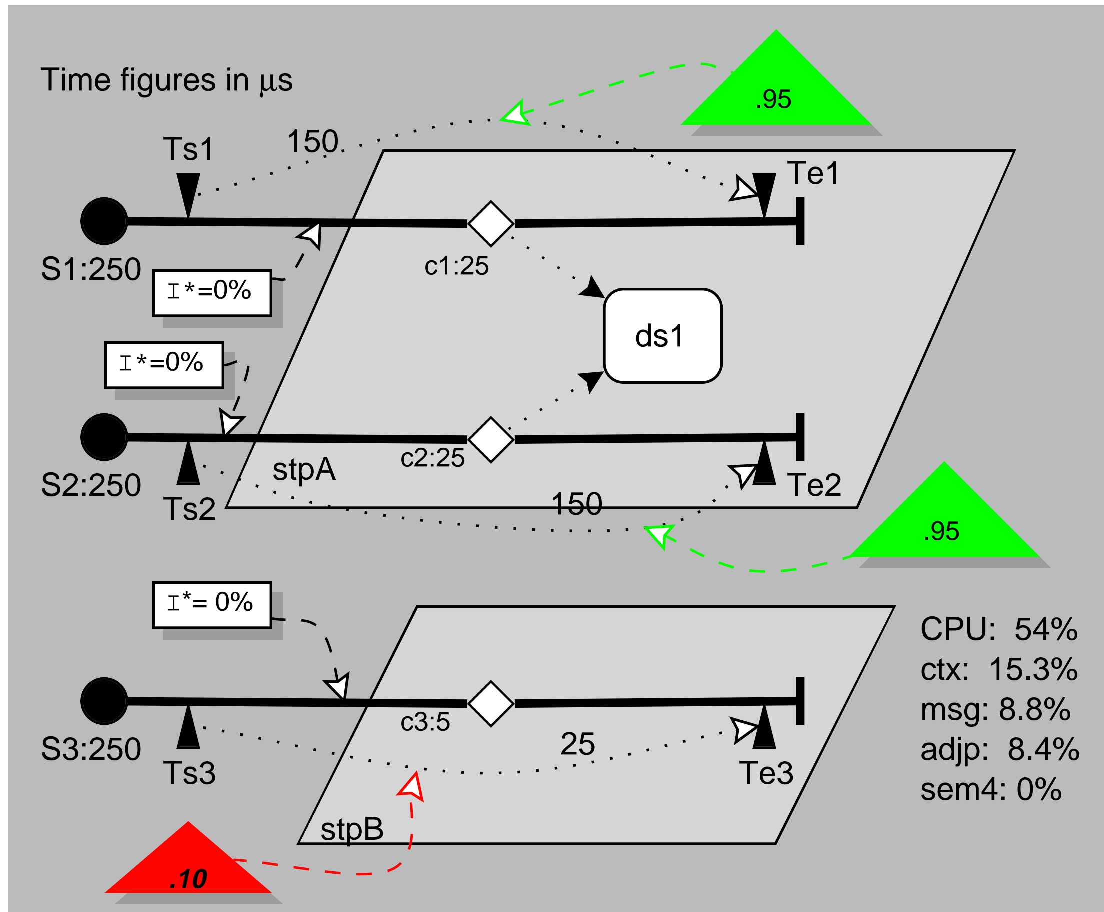

Figure 5.12: Case INV4 – Removing inversion in Figure 5.11 makes things worse

Case INV4

If we now remove the inversion as we did before by adding a second process, we find that none of the responses has a higher success ratio. See Figure 5.12. The more-significant overhead is the reason: if context must be switched to *stpB* in order to process an arrival at *S3* then there is no way that the arrival can be processed by the deadline. Recall that context switching takes 20 microseconds of cpu time. If *stpB* processes multiple arrivals one after another then it is possible that some of them will meet their deadlines.

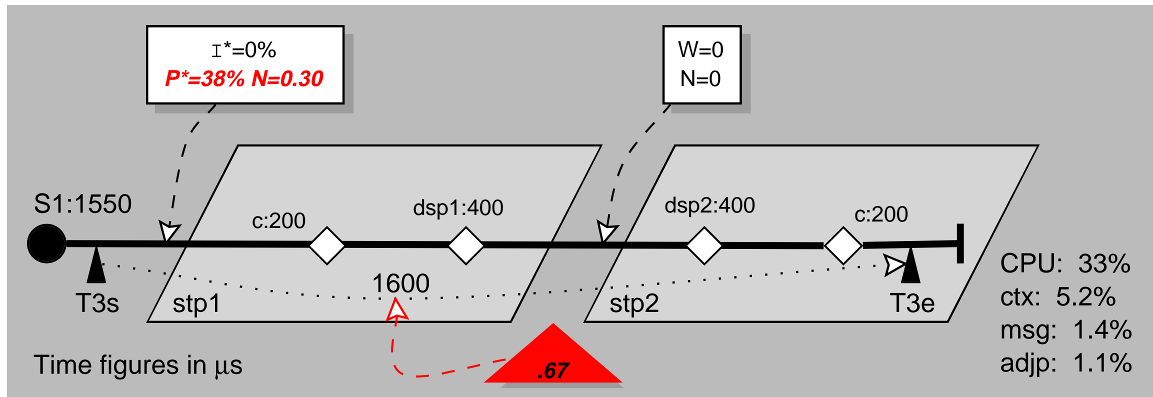

Figure 5.13: Case EXT2 – An architecture which increases concurrent use of devices compared with the architecture of Figure 5.8

5.5.2 External device metric examples

The metric results for the first external device metric example is shown in Figure 5.8. Call that Case EXT1. Remember that in the architecture shown in the figure, only one token can be making progress at a time even though needed devices may be idle.

Case EXT2

We need to propose an architecture that allows multiple tokens to progress concurrently. One way to do this is to split the activities in the original single-threaded process into two (or more) single-threaded processes. A two-process pipeline architecture and its evaluation are shown in Figure 5.13. The first two activities were grouped into one process and the last two activities into another in an attempt to give a goodly amount of time during which different devices can work concurrently: during a burst of arrivals and if there was no overhead then two devices could always be working concurrently. If the first three activities were placed in one process then multiple devices would never be able to work concurrently. If the last three activities were placed in one process then two devices could be working concurrently half of the time during a burst of arrivals.

The new architecture has two processes in front of which device need can be measured. The first point has a time-averaged external device need of 0.30, 38% of which is potentially poorly scheduled. The second point has a time-averaged external device need of 0. The original architecture had a time-averaged external device need of 1.6, 65% of which was potentially poorly scheduled. The external device metric thus seems to indicate that the two-process architecture is better. The success ratio of the response bears this out, as it has gone up to 67% from 28%.

Case EXT3

It seems however that further improvement could be made by trying an architecture with a greater number of threads. One such architecture would put each activity in a separate process, thereby potentially allowing *DSP1*, *DSP2*, and the CPU to be used concurrently. Obviously this architecture would also have an increase in overhead because of greater numbers of messages, context switches, and priority adjustments. During a burst of arrivals and ignoring overhead, all three devices could be constantly working.

By creating a single multi-threaded process, we can easily allow the situation where all three devices are working simultaneously. Such an architecture is shown in Figure 5.14. Here, the time-averaged external device need is down significantly to 0.09, 41% of which is potentially poorly scheduled. The metric cannot be expected to go to zero in this example. Consider a moment when all the threads are idle, and then a burst of arrivals occur close together. While the cpu is processing the first token, the waiting tokens will be generating device need, and the two DSPs will be recorded as potentially poorly scheduled.

Notice that we must consider both P^* and N when comparing the metric results for different architectures. The magnitude of P^* indicates the degree to which a potential scheduling problem exists in the design. The magnitude of N indicates the degree to which the potential scheduling problem will manifest itself. The rather small N in Figure 5.14

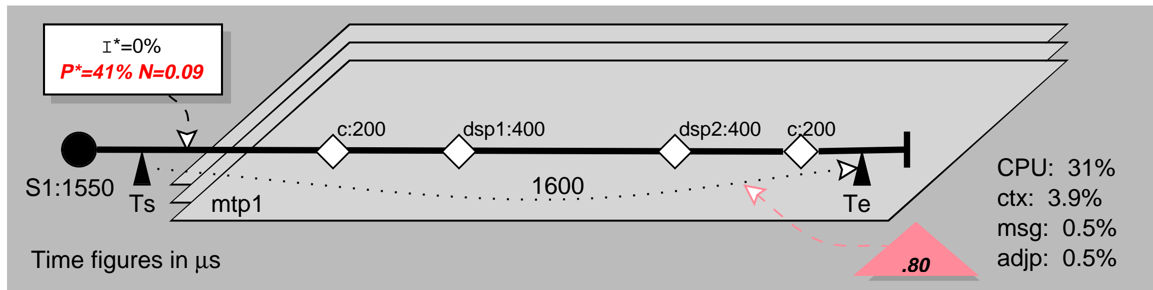

Figure 5.14: Case EXT3 – An architecture which further increases concurrent use of devices

explains why it reports a higher success ratio even though the P^* is larger than the corresponding metric in Figure 5.13. The metrics I^* and W must be considered together for similar reasons.

5.5.3 Internal device metric examples

The metric results for the internal device metric example are shown in Figure 5.9. Call this Case INT1. In this example, problems were identified by all three metric measurements: inversion metric and both internal and external device metrics. The internal device metric seems to be the most disturbing of the three, and so the first proposed architectural improvement will focus on reducing that metric. Remember that in the architecture shown in the figure, of the two tokens emerging from the AND fork following a token entering it, only one can make progress at a time even though both devices needed by the scenario instance are idle.

Case INT2

We need to propose an architecture that allows multiple tokens output from an AND fork to progress concurrently. This can be done by allowing more than one process to work on the fork's branches.

A proposed improved architecture is shown in Figure 5.15. Here, the disk-accessing

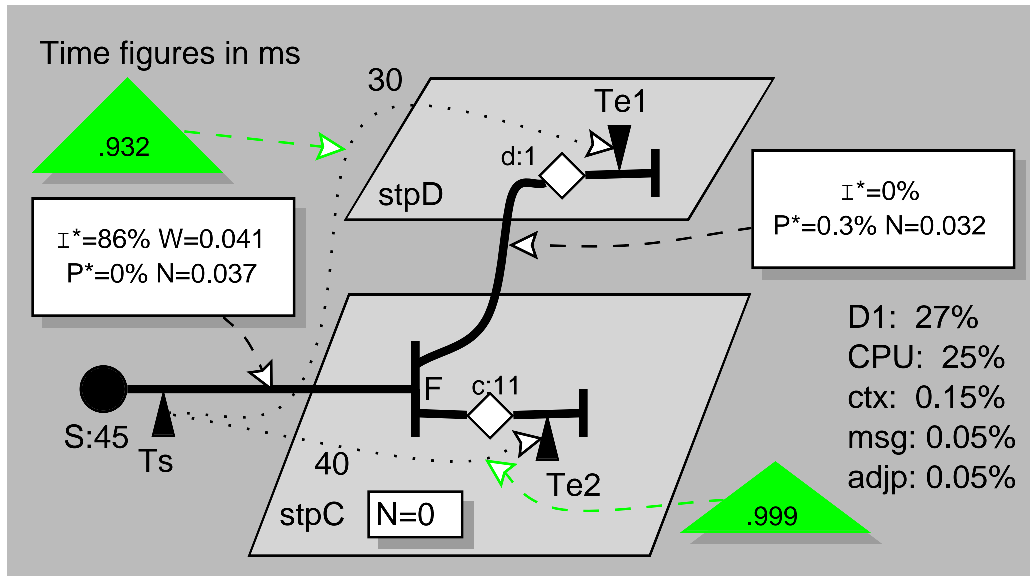

Figure 5.15: Case INT2 – An architecture which allows concurrent processing of AND fork branches

activity has been put in a separate process. This has reduced the internal device metric to zero, and now both responses are reporting better than 90% success ratios. Overhead, although negligible in this example, has increased over the one-process architecture. Is this the best concurrency architecture? Perhaps not, as the inversion metric where the path crosses into process *stpC* reports that 86% of the time-averaged number of waiting tokens is inverted. This is happening because tokens waiting at *stpC* have a deadline derived from the urgent branch of the AND fork, but must wait until the less-urgent branch finishes its current processing.

Case INT3

For this reason, it seems that a better concurrency architecture would be to put the AND fork in the same process as the more-urgent branch. This is shown in Figure 5.16. Indeed, the minimum success ratio has slightly increased from 93.2% to 93.7%. This is accompanied by a decrease in the other success ratio to 98.7% from 99.9%. Context switching happens

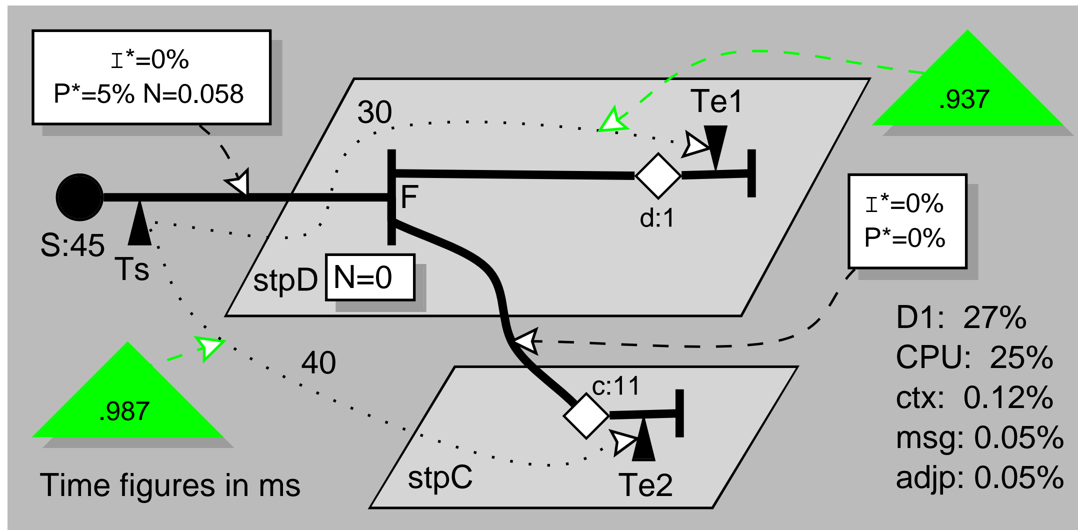

Figure 5.16: Case INT3 – An improvement to the architecture of the previous figure

slightly less often than in the previous architecture. This leaves only one remaining non-zero-valued metric: the external device metric where the path crosses into *stpD*. This metric registers non-zero under conditions such as the following. If the disk access will take less time than activity *c* for a certain token arriving at the process mailbox of *stpD*, and another token arrives at the process mailbox of *stpD* at least 10ms after the first, the second token may have to wait for activity *c* to finish even though the disk is idle.

In this small example, it would actually work well to use static priorities with process *stpD's* thread having the higher priority. In more complex examples the benefits of EDF priorities are apparent: an arriving token can cause a thread processing another token to be pre-empted or not depending upon relative timings.

Case INT4

To improve the performance of EDF in this example, we can add an intermediate response-time requirement as shown in Figure 5.17. The intention here is that the disk is scheduled

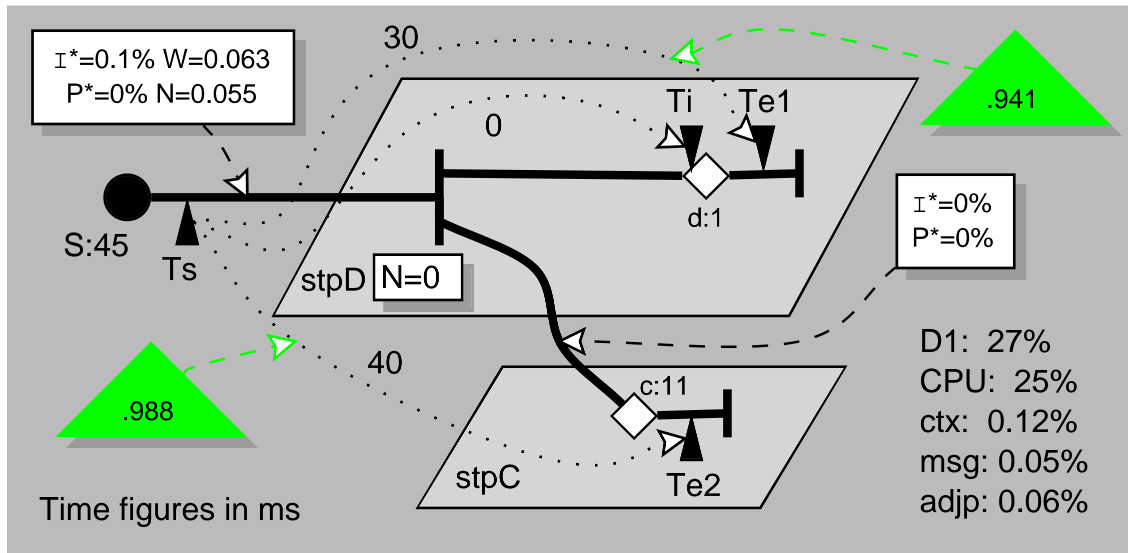

Figure 5.17: Case INT4 – Results with an added intermediate deadline to tune the scheduling

according to the deadline derived from the response-time requirement ending at *Te1*, but the cpu work of executing the AND fork and actually scheduling the disk access is operated with a deadline derived from the response-time requirement ending at *Ti*. With the intermediate deadline the minimum success ratio increases slightly again, this time to 94.1%. The other success ratio is relatively unchanged at 98.8%. Overhead for adjusting priorities has slightly increased.

5.5.4 Example of metric for a path segment returning to a process

In sections 5.5.1 and 5.5.2 we have illustrated the usefulness of the external metrics applied at a point where a path crosses into a process for the first time. In this section we will provide examples which demonstrate the use of the metrics where a path segment carries tokens returning to a process.

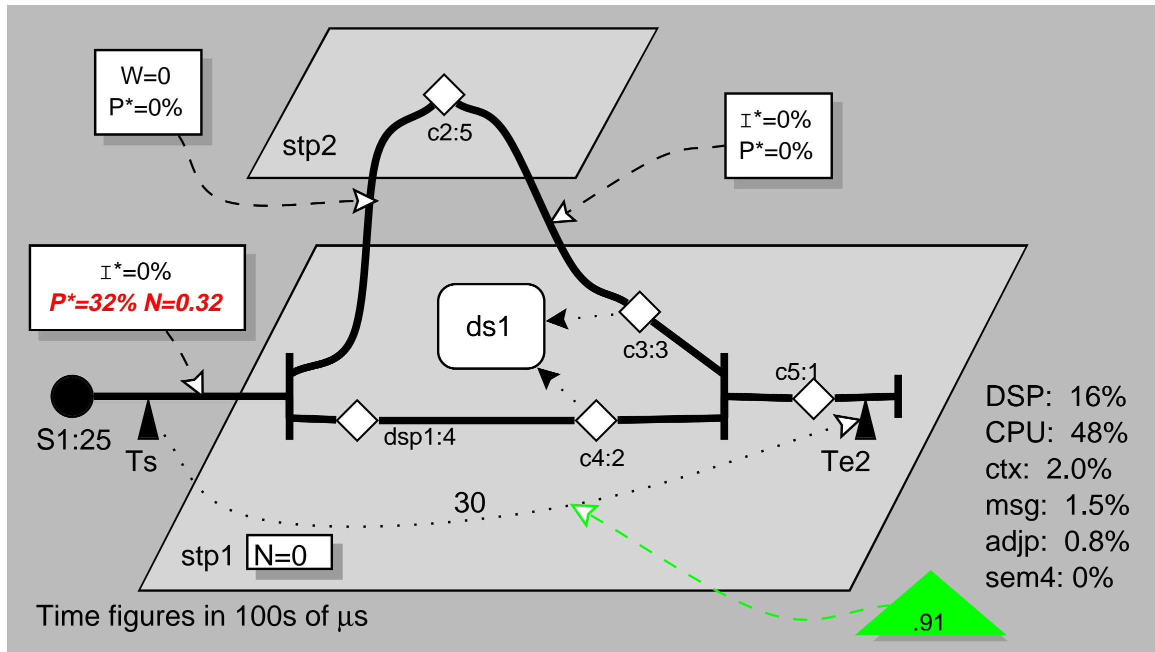

Figure 5.18: Case RET1 – A UCM with tokens returning to a process

Case RET1

Consider the UCM of Figure 5.18. The path segment between activities *c2* and *c3* carries tokens returning to *stp1*. The external metrics for this path segment indicate no problem.

Case RET2

Now suppose the cpu demand of *c2* were 100μs instead of 500μs, as shown in Figure 5.19. With this change there is now a time-averaged external device need of 0.18 at the segment returning to stp1, 53% of which is PPS. What happens is that when a token leaves *stp2*, the cpu will be idle because activity *dsp1* is still using the DSP and hence the token cannot be received.

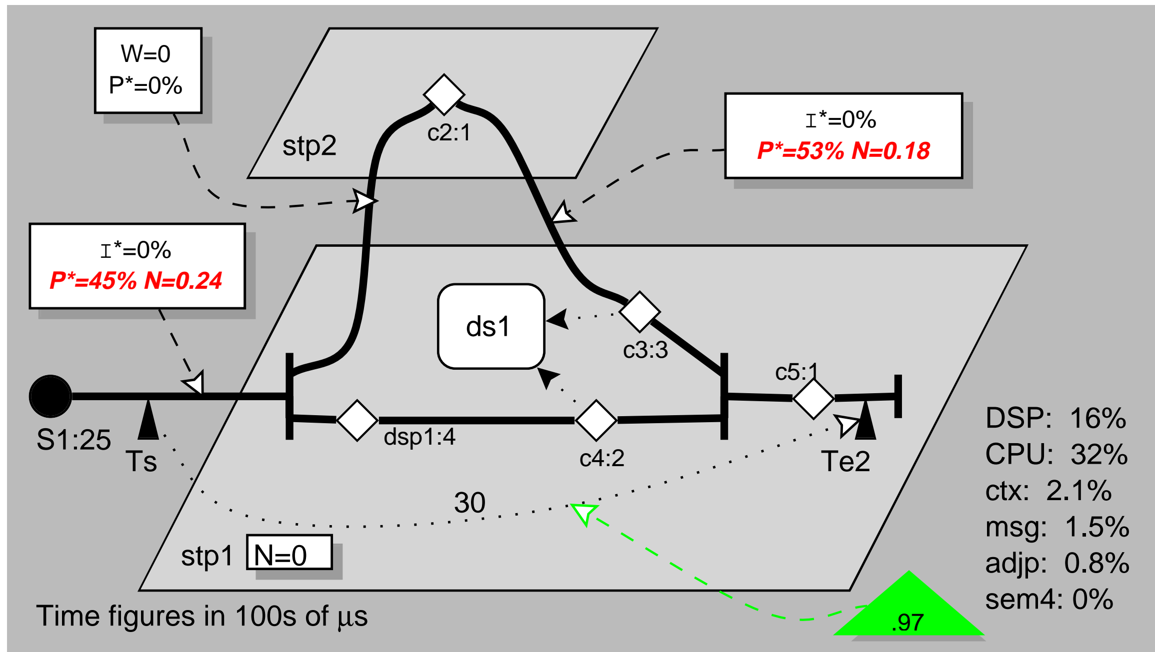

Figure 5.19: Case RET2 – The external device metric reports a problem on segment between $c2$ and $c3$

Case RET3

Although 97% of responses complete within the specified delay, we could possibly improve the performance if a token leaving *c2* does not have to wait while the cpu is idle before starting *c3*. An architecture which achieves this is shown in Figure 5.20. In the architecture, activity *c3* has been moved from *stp1* to *stp2*. This requires that *ds1* be protected from concurrent access, an additional overhead. The architecture gives better performance even though overhead has slightly increased.

Case RET4

The above examples illustrate the external device metric where tokens return to a process. By making a different change to the UCM of Figure 5.18 we can at the same path segment illustrate the occurrence of inversion. To do this, we add after *c3* a timestamp point which

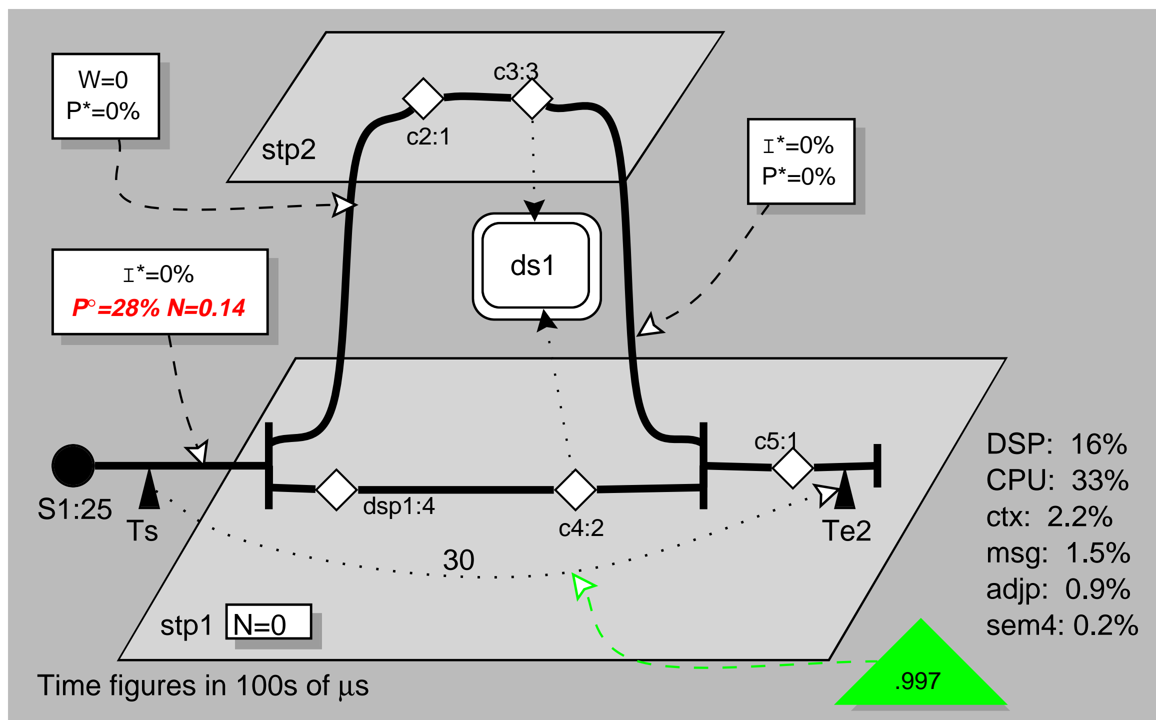

Figure 5.20: Case RET3 – An architecture which eliminates PPS device need where tokens return to process *stp1*

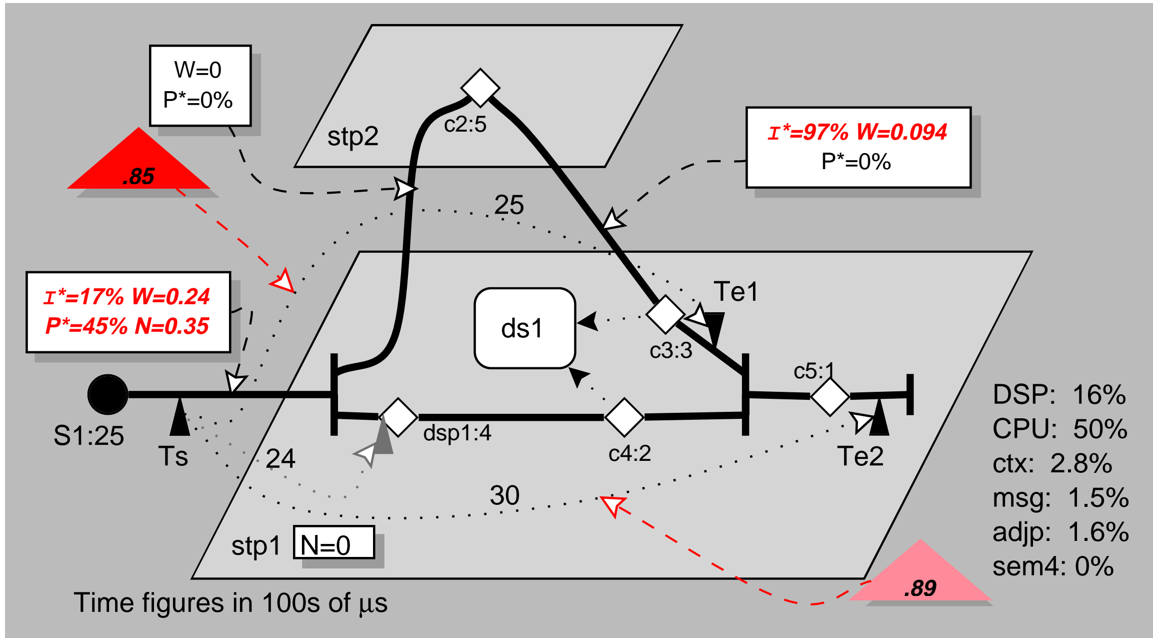

Figure 5.21: Case RET4 – The external inversion metric reports a problem on the segment between $c2$ and $c3$

is used to define a second response-time requirement. This is shown in Figure 5.21. Also added in this figure is an intermediate timestamp point at the DSP-using activity. The DSP is scheduled according to the deadline derived from the response-time requirement ending at *Te3*, but the cpu work needed to actually schedule the DSP is executed with a deadline derived from the intermediate response-time requirement of 2400us. The time-averaged number of waiting tokens at the path segment where tokens return to *stp1* is 0.094. 97% of these are deadline inverted. A token waiting on segment *c2*→*c3* will have to wait while the cpu is executing activity *c4* for a less-urgent token. Neither response-time requirement is meeting the minimum 90% success ratio, especially the response-time requirement ending at *Te1*.

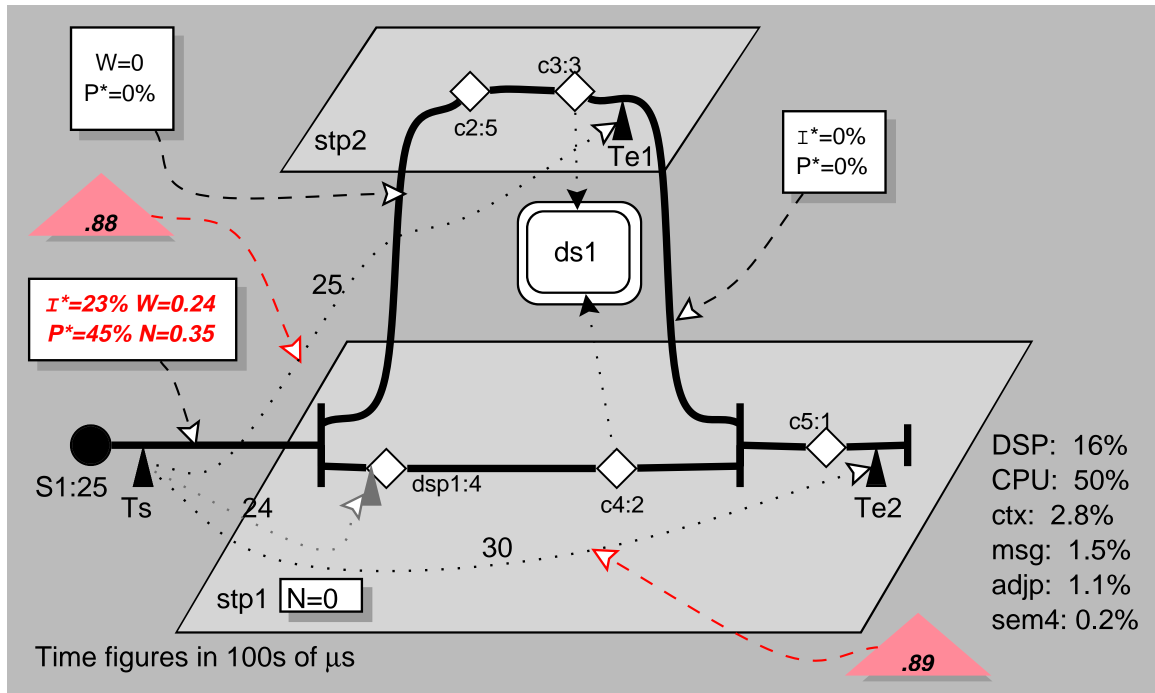

Figure 5.22: Case RET5 – An architecture which eliminates inversion when tokens return to process *stp1*

Case RET5

An architectural adjustment similar to the one used in Figure 5.20 – moving activity *c3* – should also eliminate inversion for returning tokens if applied to the UCM in Figure 5.21. Timestamp *Te1* must of course move with activity *c3* to *stp2*. This is shown in Figure 5.22. Inversion was indeed eliminated for returning tokens and the success ratio for the response-time requirement ending at *Te1* made a definite improvement from 85% to 88%.

5.6 Discussion of the metrics

In this chapter three metrics were presented for diagnosing concurrency architectures. Depending upon the situation encountered, here are some thoughts on ways in which one can react to problems identified by the metrics.

1. Inversion or PPS device need at the process mailbox of a single-threaded process – divide the process or make it multithreaded. Multithreading is often superior, but if the process can be divided such that data objects are not shared between new processes, then protecting data objects, with its inherent overhead, can be avoided.

2. Inversion or PPS device need where a path returns to a process – in this chapter success has been observed from moving activities along a path between processes. Creating an additional process may be suitable in some situations. Multithreading, an option if the process is not already multithreaded, will not help in this case.

3. PPS device need internal to a process – divide the process. Again, multithreading will not help in those situations where it is possible.

4. PPS device need at the process mailbox of a multithreaded process – if there is no PPS device need internal to the process, then the external PPS device need can be reduced by dividing the process, but this won't always improve the performance of the architecture. Sometimes the PPS device need is not caused by an insufficient number of processes, but is just the result of non-periodic arrivals of external events.

Chapter 6

Case Study

The previous chapters have described a notation for specifying the scenarios of an application, a way of building a virtual implementation to simulate and evaluate a proposed concurrency architecture, and metrics that can be used to diagnose concurrency problems in an architecture. This chapter applies those ideas to a substantial example: a Group Communication Server.

6.1 Description of the Group Communication Server application

Figure 6.1 shows the specification of a Group Communications Server (GCS) which stores a set of documents and gives users access to them. Each user can subscribe to a set of documents, submit new documents, and update documents. When a user updates a document, all subscribers of that document are notified. Users can request that the current version of a document be sent to them. There are five numbered scenarios as shown in Figure 6.1:

1. Updating a document.

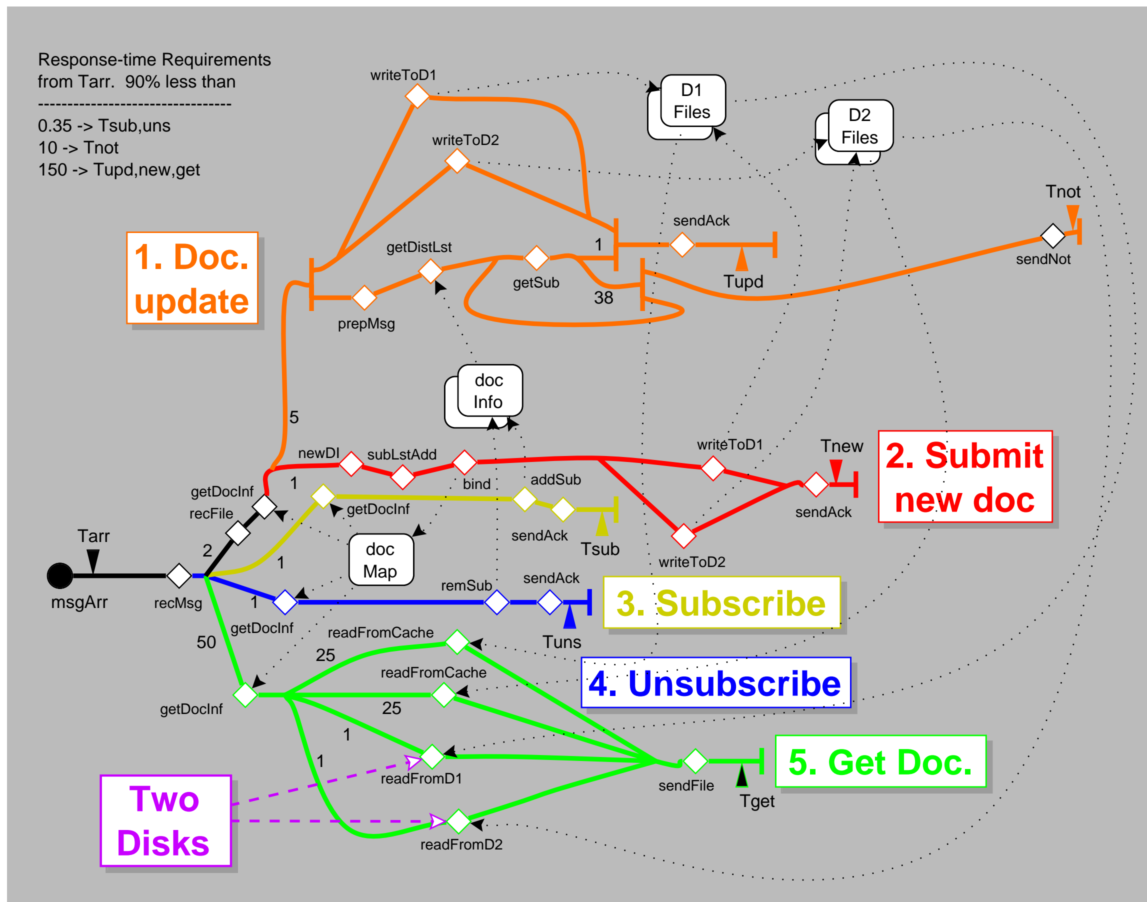

Figure 6.1: The scenarios in the GCS application

2. Submitting a new document.

3. Subscribing to a document

4. Unsubscribing from a document

5. Retrieving the most recent version of a document.

All the scenarios start at the path start named *msgArr*. The first processing on a request is specified by activity *recMsg* at which the request type and document name are extracted from the request. The request type identifies whether the user wants to store or retrieve a document or whether the user wants to subscribe to or unsubscribe from a document. Scenarios 1 and 2 are the possible results of a request to store a document.

Next on the path is an OR fork. As indicated by the numbers beside the branches, for every 2 store-document requests, on average there are 1 subscribe, 1 unsubscribe, and 50 retrieve-document requests.

6.1.1 Scenarios 1 and 2: updating and submitting new documents

This section describes the paths for the topmost two scenarios: updating a document and submitting a new document. These scenarios follow the top branch of the first OR fork. The other three scenarios follow the other branches.

At activity *recFile*, the document text is extracted from the request. Next is activity *getDocInf*, which reads from a data object: a dictionary named *docMap*. Using the document name as a key, *docMap* is queried to see if the server has already stored a previous version of the document, and if so the information on the document is returned.

Scenario 1: updating a document

If an earlier version of the document is already stored on the server, we are updating the document. The top branch of an OR fork is followed which takes us to an AND fork, which

in turn 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 (at activity *getSub*) and the path going to *sendNot* is forked for the sending of the notification message to the extracted subscriber. Forking the path in this way allows the sending of each notification message and the extraction of subscribers to be done potentially in parallel.

These sub-paths are specified as parallel because the prerequisites for executing each of these sub-paths have already been achieved before the location of the AND fork, and the activities in each sub-path have no sequential dependency on activities in the other. The sub-paths can potentially be executed in parallel. There is no restriction that they must be executed in parallel. This is identical to how a critical path model is constructed.

When the path exits from the notification loop, it joins with the disk-writing sub-path, and a "success" message is prepared for the document-sending client by activity *sendAck*. 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.

Scenario 2: submitting a new document

If a new document is being sent, the following activities are executed after the update-document path forks off:

- At activity *newDI*, a new *docInfo* object is constructed.
- At activity *subLstAdd*, a new subscriber list is created, initialized with the document sender as the initial subscriber, and inserted in the new *docInfo* object.
- At activity *bind*, the new *docInfo* object is inserted into the *docMap* dictionary. Note that prior to being inserted in *docMap*, no other activity can reference the new *docInfo* object, and hence it has not been drawn explicitly on the diagram for the new-document scenario.
- Activities *writeToD1* and *writeToD2* are for writing a document to disk 1 or disk 2 respectively. Notice that a document in the filesystem is considered a data object, and might need to be protected against concurrent access. While a document is being written to disk by one context, another context cannot write to or read from the document. The simulation will randomly choose the disk to write to.
- At activity *sendAck*, an acknowledgement is prepared for the user.

6.1.2 Scenario 3: subscribing to a document

If a subscribe request is received, the following activities are executed after activity *recMsg*.

- Activity *getDocInf*, which retrieves information on the document.
- Activity *addSub*, which adds the requesting user to the list of subscribers for the specified document.
- Activity *sendAck*, which prepares an acknowledgement for the user.

Scenario 4, which is the processing of an unsubscribe request, is very similar to the processing of a subscribe request.

6.1.3 Scenario 5: retrieving the most recent version of a document

The processing of a retrieve-document request occurs as follows after activity *recMsg*.

- Activity *getDocInf* is executed as for other scenarios
- Next, the application will read a document from the filesystem. The document will most likely be read from the filesystem cache. This incarnation of this model assumes that for every 25 documents read from the cache, only 1 will have to be obtained from the disk.
- Finally, the file is prepared for sending to the requesting user.

6.2 Performance numbers

All the requests considered together arrive according to a Poisson process with an average interarrival time of 4 ms (250 requests/second).

All response-time requirements are based on delays that start at *Tarr*. For purposes of discussion, 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*. Through this chapter these response-time requirements will be referred to as *Rupd*, *Rnew*, *Rget*, *Rnot*, *Rsub* and *Runs* respectively. Because the response-time requirements are straightforward, and to keep the UCM diagram simple, response-time requirements have not been drawn on the UCM as dotted lines, as has been done for other UCMs in this thesis. Each response-time requirement in this example specifies that 90% of responses be completed within the specified delay.

The service requirements for each activity were determined from a programmed C++ implementation [58] of the application by using the Quantify tool [35]. The C++ implementation was functionally complete and was built on top of the ACE library [57], and

measurements were made on a Sun workstation running the Solaris operating system. Two architectures were implemented, a single-thread design and a thread-per-request design. A number of scenarios were measured using an executable which had been instrumented using the Quantify tool.

6.3 Concurrency architectures for the Group Communication Server

In the following sections we propose and evaluate five concurrency architectures for the Group Communication Server application. The architectures range from one with no concurrency, to one with enough processes and threads to allow every potentially concurrent instance of an activity to have its own thread.

We start by presenting the no-concurrency architecture, identify what its concurrency problems are, and then propose another architecture to attempt to solve those problems.

6.3.1 Case 1: Single-thread architecture

The first architecture that was evaluated is the most simple: all operations are performed by a single thread (Figure 6.2). 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. It also minimizes the messaging and context switching overhead.

As shown in Figure 6.3, the PERFECT tool reported that the system was unstable. This architecture takes so long to process each request, one at a time, that it cannot keep up with the rate of arrival of requests. Inspection of detailed simulation results showed that the number of waiting tokens at the *process* mailbox was growing larger and larger as the simulation run continued. Response times were growing larger and larger as well. If we add together the 3 device utilizations shown in Case 1 of Table 6.1, the result is 100%

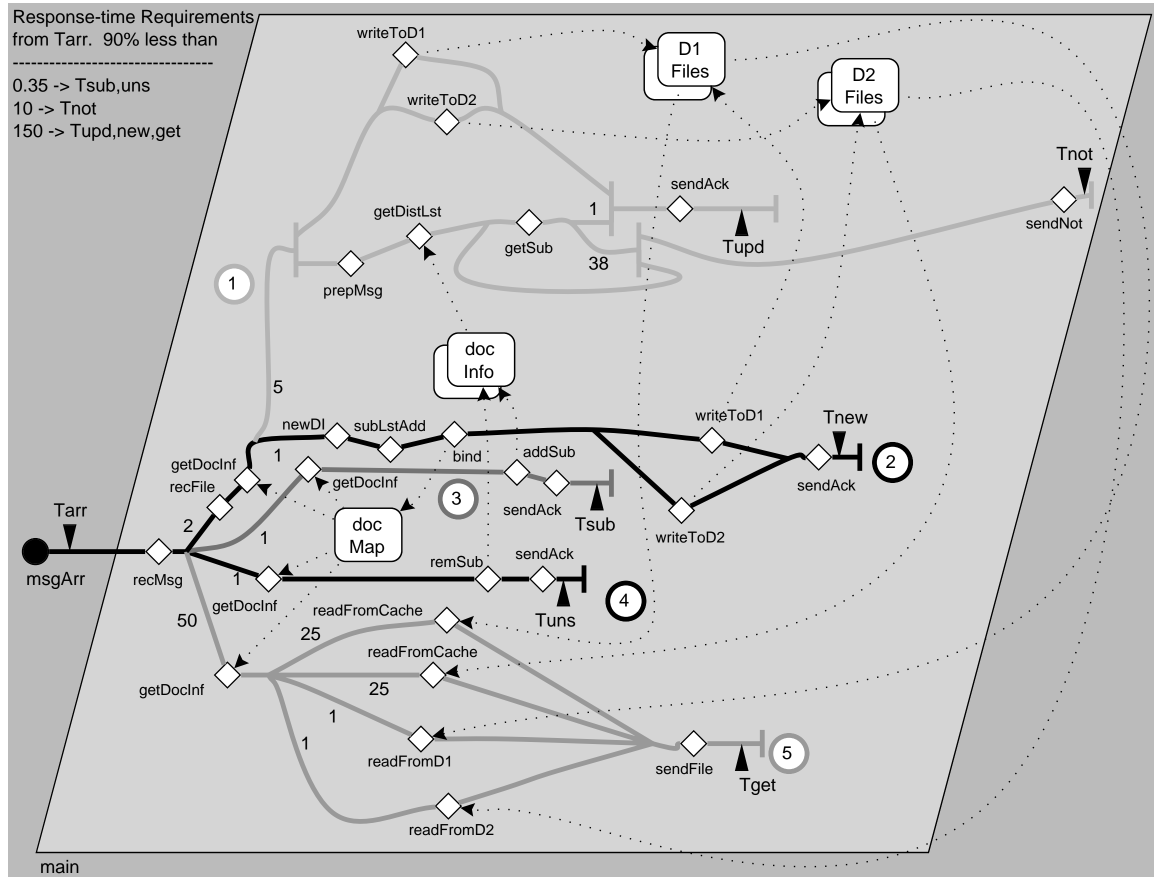

Figure 6.2: Use Case Map for Group Communication Server with a single-threaded process (Case 1)

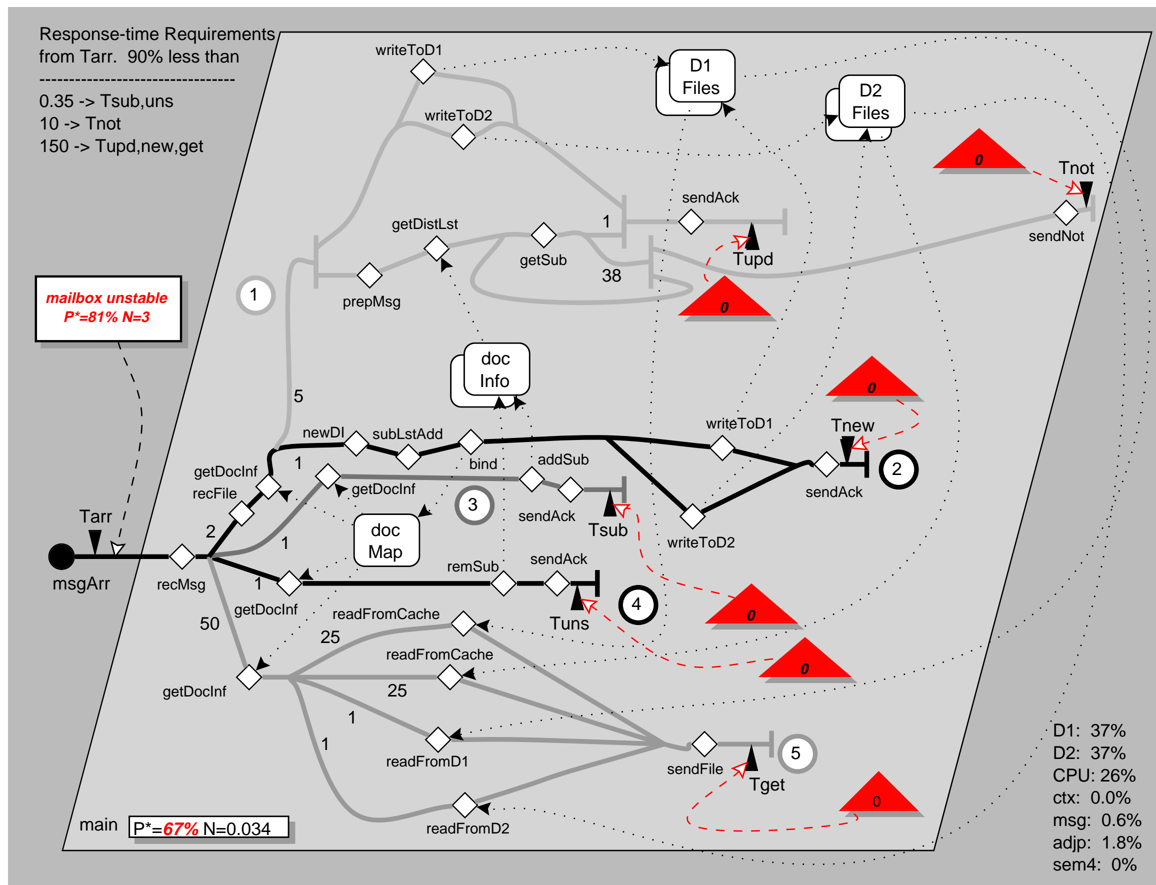

Figure 6.3: Evaluation of Single-thread architecture

from which we can deduce that the thread is never idle. The system is unstable because the average thread service time is too long.

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	37	37	26	0	0.6	1.8	0	***infinite/0***	***infinite/0***	***infinite/0***	***infinite/0***	***infinite/0***	***infinite/0***
2	6 Thread	49	49	40.0	3.9	2.3	3.8	0.20	***173/53***	***93/32***	***99/85***	***6./53***	***6./55***	15/99.2
3a	SMTP	48	49	37.6	3.5	0.7	3.1	0.41	93/90	3.6/95	90/91	0.335/93	0.369/94	7.5/99.4
3b	- double subscribers	50	50	74.4	7.6	1.4	6.0	0.78	***105/85***	***7.1/75***	***97/89***	***0.44/85***	***0.46/89***	10.1/99.4
3c	- - with 2 processors	49	49	48.5 +23.9	5.5	1.4	6.0	0.80	***96/89***	***6.2/81***	91/91	0.32/98	0.34/99.6	5.4/99.6
3d	- - 2X faster processor	50	49	36.4	5.5	1.4	6.0	0.80	92/91	3.2/96	90/91	0.17/99.8	0.19/99.8	4.8/99.6
4a	Parallelism in Updating	49	49	38.0	3.7	0.8	3.1	0.41	90/91	4.8/92	90/92	0.342/92	0.38/95	7.7/99.3
4b	- double subscribers	49	49	74.7	7.8	1.5	6.1	0.77	91/91	***8.4/70***	***97/89***	***0.44/84***	***0.49/88***	9.8/99.4
4c	-- with 2 processors	49	50	48.3 +24.5	5.7	1.5	6.1	0.80	90/91	***6.4/80***	91/91	0.32/98	0.34/99.7	5.4/99.5
4d	-- 2X faster processor	50	50	36.6	3.2	0.7	3.0	0.39	90/91	3.9/95	91/91	0.17/99.8	0.19/99.8	5.0/99.5
5a	Maximum parallelism	49	49	43.1	6.3	2.6	4.0	0.41	91/91	***16.9/31***	***95/89***	***0.65/89***	0.8/91	9.5/99.2
5b	- with 2 processors	49	49	30.2 +12.0	5.5 total	2.5 total	4.0 total	0.41 total	90/91	***6.5/81***	89/91	0.37/98	0.37/99.4	7.2/99.3
5c	- with 4 processors	49	49	26.2+8.7 +4.8+2.4	5.2 total	2.5 total	4.0 total	0.41 total	90/91	***5.0/88***	89/91	0.321/99.2	0.341/100.0	6.9/99.3

Table 6.1: Output reported by PERFECT tool for GCS architectures

The device metric at the process mailbox tells us that of the time average of 3 devices needed by tokens waiting there, 81% of that time-average is potentially poorly scheduled. Because there is only one thread, while one device is being used the other two devices are idle and are potentially poorly scheduled. While the CPU is being used, the CPU itself will sometimes be potentially poorly scheduled (it will be working with deadline X on the less-urgent portion of a less-urgent scenario while a message with a token having a deadline less than X is waiting in the process mailbox.) The inversion metric at the process mailbox is consistent with the system being unstable. The number of waiting tokens keeps growing unbounded. Eventually the deadline of any token being processed would be earlier than the current simulation time, i.e. eventually all deadlines are missed.

Clearly we need to add some concurrency so that multiple devices can be used simultaneously, thereby allowing a higher utilization of each device and thus allowing more work to be done.

6.3.2 Case 2: Six-thread architecture

To allow the disks and processor to be used simultaneously, consider the architecture shown in Figure 6.4 with six 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 6.4 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 are 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. Had the elements executed by *notify* been left in process *main*, the *main* thread would have been unable to process other arriving

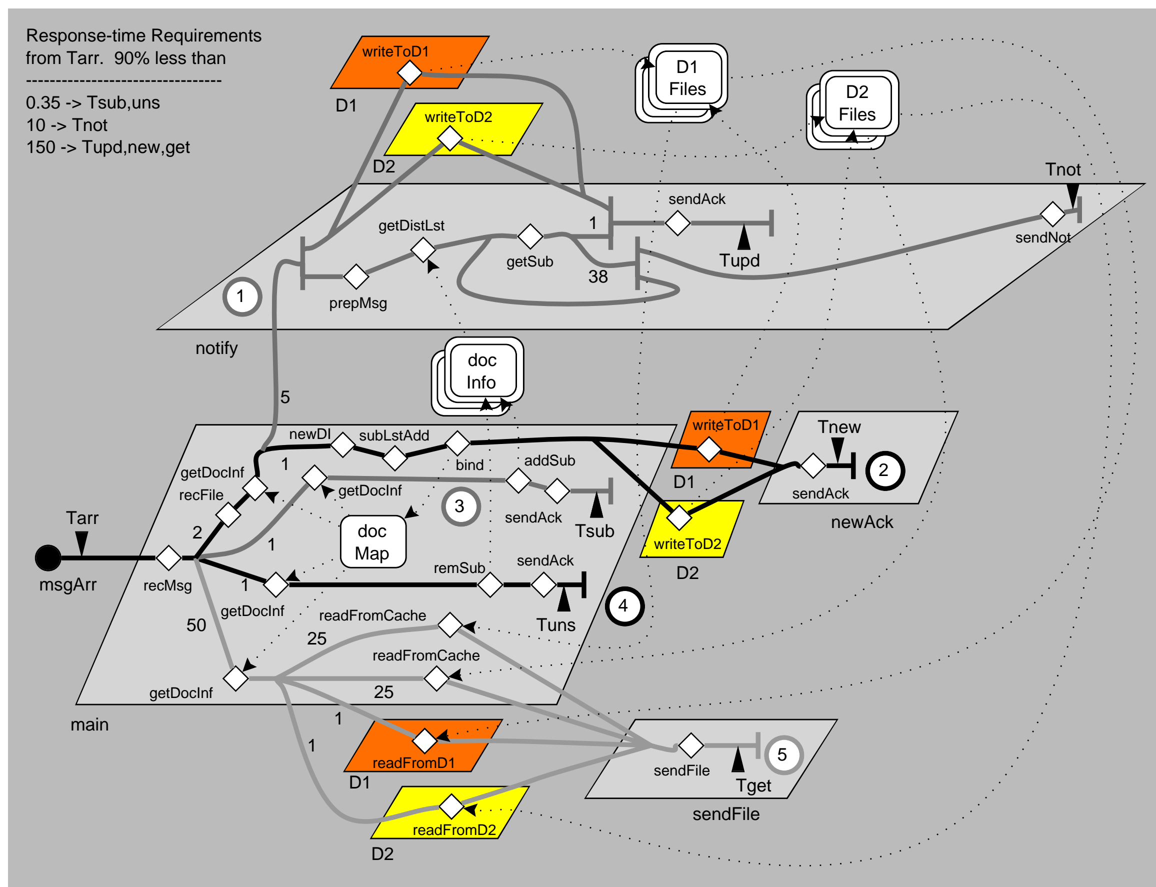

Figure 6.4: Use Case Map showing Six-thread architecture (Case 2)

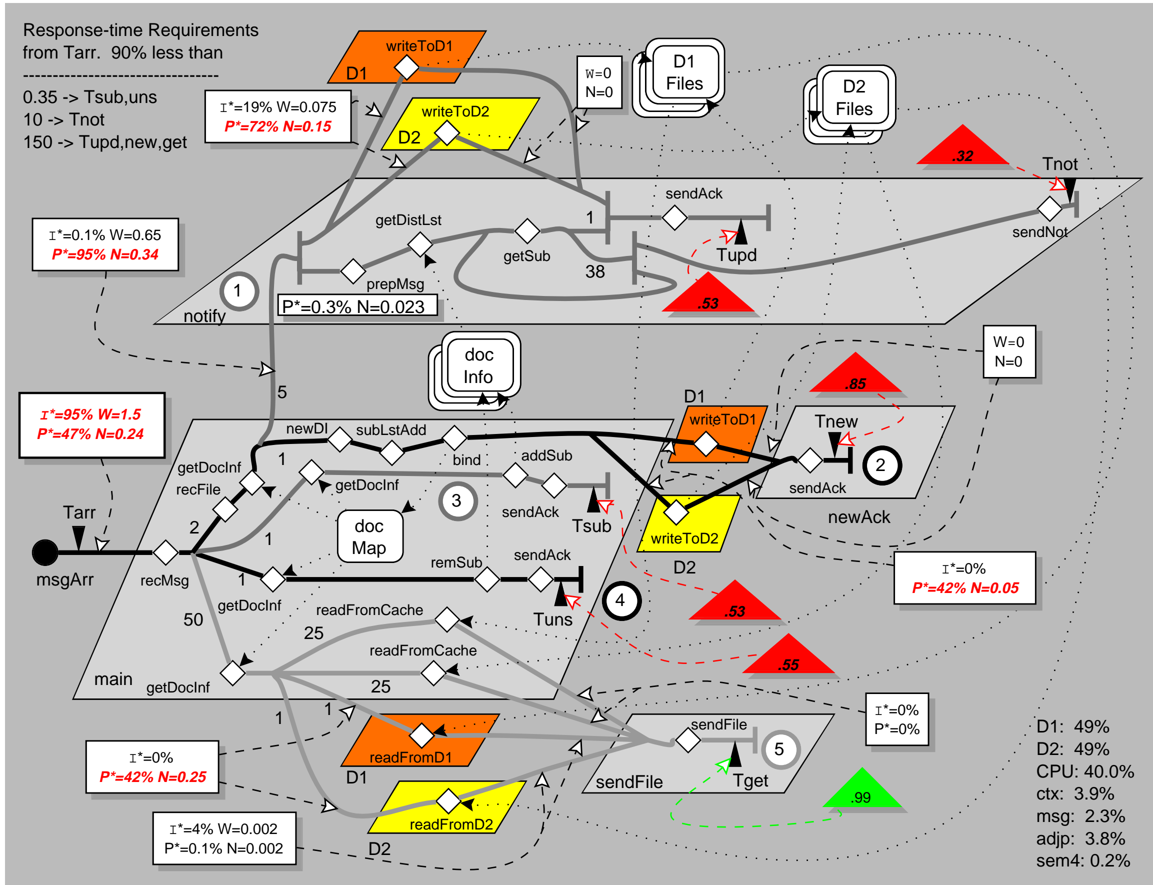

Figure 6.5: Evaluation of Six-thread architecture

client messages while waiting for a message from *D1* or *D2*. This is why *notify* was created. Processes *newAck* and *sendFile* were also created so that process *main* would not have to wait on any *return* mailbox.

Table 6.1 and Figure 6.5 show that this architecture is a great improvement over Case 1, in that 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. Requirement *Rnew* is close to being met with an 85% success rate, but requirement *Rupd* does much more poorly with only 53% of responses less than 150ms. *Rsub* and *Runs* also have low success rates: 53% and 55% respectively.

This architecture has significantly more overhead than the single-thread architecture: 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 architecture. Semaphore overhead is now present because protection has been introduced so that an individual file cannot have active readers or other writers while a thread is writing to it. Also, the management data for a particular document, including for example its list of subscribers, is protected so that reading and updating are done under mutual exclusion.

Several of the metric values in Figure 6.5 indicate problems. The inversion metric at the *process* mailbox of *main* shows that of the average of 1.5 tokens waiting at the mailbox, 95% were deadline inverted. This might explain why responses *Rsub* and *Runs* are relatively far from the requirements. For example the processing of a subscription request, relatively urgent, cannot pre-empt the execution of an activity like *readFromCache*. Compounding this, the "retrieve document" scenario occurs relatively frequently, meaning that activity *readFromCache* is executed relatively often.

The device metric at the *process* mailbox for *main* refers entirely to CPU, as this is the only device which *main* uses. How can we explain that 47% of the time-averaged device (CPU) need is potentially poorly scheduled? As it happens, the problem is with concurrent access to files. A file cannot be read from the cache while another thread is writing the file. Until the file is successfully written to disk, the cache will not be updated and no reads will be allowed. So while thread *D1* is writing a file to disk *D1*, if thread *main* tries to read that file from the cache it will end up blocked on the file. This can cause the idle thread to be scheduled, and hence the CPU will be poorly scheduled as other tokens waiting at *main*'s *process* mailbox could use the CPU.

At the *process* mailbox of *notify*, 95% of time-averaged device (CPU) need is potentially poorly scheduled. One reason for this is that messages arriving at the *process* mailbox for

notify will queue while *notify* is blocked waiting on its *return* mailbox. 0.1% of the time average of 0.65 tokens waiting at the *process* mailbox of notify are deadline inverted. The deadline inversion happens when a message, whose carried token has a deadline derived from *Rnot*, arrives before the previous execution of the update scenario has finished executing *sendAck*.

Processes *D1* and *D2* behave symmetrically, and so on the diagram we have been able to write one set of metric results for each of the corresponding paths entering and leaving *D1* and *D2*. In the middle of the diagram, the paths leaving process *main* and entering processes *D1* and *D2* show that none of the waiting tokens are inverted, but 42% of the time-averaged device need of 0.05 is potentially poorly scheduled. There are two possibilities:

1. While one token is using a disk, a token waiting between activity *bind* and activity *writeToD1* or *writeToD2* can use the CPU (albeit for only a short while) but the CPU is idle.

2. The disk that a waiting token needs is idle, but the CPU is now busy elsewhere processing a more urgent token.

The paths crossing from process *notify* into *D1* and *D2* show not only potentially poorly scheduled device use, but also inversion: 19% of waiting tokens are deadline inverted. Why is it that there is no inversion for paths leaving main and going to process *D1* or *D2*, but there is for each path leaving process *notify*? Consider one token which leaves process *main* and arrives at process *notify* while process *notify* is waiting for a message at its *return* mailbox. While process *notify* is waiting, another token leaves process *main* along scenario 2 and is received by process *D1*. Finally process *notify* receives a message on its *return* mailbox and eventually the token waiting at the *process* mailbox of *notify* is able to reach the *process* mailbox of process *D1*. This token is more urgent than the token currently being processed by *D1*, hence the inversion.

At the bottom of the diagram along scenario 5, there is the same 42% of potentially poorly scheduled device use for the paths crossing into *D1* and *D2* as we had for the paths on scenario 2.

Continuing further along scenario 5, we see some inversion for the paths crossing from *D1* and *D2* to process *sendFile*, but no inversion on the paths going from process *main* to *sendFile*. The reason for this is similar to the reason for inversion when the paths cross into *D1* and *D2* along scenario 1. A token with a less-urgent deadline which came to *sendFile* directly from process *main* may be being processed by *sendFile* when a more-urgent token which came through *D1* arrives at the mailbox of *sendFile*.

To summarize the main problems with this architecture which are observed during execution:

1. There is deadline inversion at the *process* mailbox of *main* – because it is a single-threaded process and it contains sub-paths with response-time requirements having different specified delay values. Possible solution: make *main* multi-threaded or divide into separate processes.

2. There is potentially poorly scheduled device need at the *process* mailbox of *main* – because it is a single-threaded process and can block. Possible solutions: make *main* multi-threaded or move the *readFromCache* activities to *sendFile*.

3. There are tokens waiting at the *process* mailbox of *notify* – because *notify* is single-threaded and can wait for messages at its *return* mailbox. Possible solution: multi-thread notify.

4. There is deadline inversion for tokens going from *notify* to *D1* or *D2* – because tokens may block on some paths to *D1* or *D2*. Possible solution: multi-thread *D1* and *D2*. Note that although a given disk request cannot be pre-empted, thread-switching can occur between disk requests.

5. There is potentially poorly scheduled device need at *D1* and *D2* – because each process uses both the CPU and a disk, and tokens may be waiting while one of them is busy. Possible solution if the disk being busy causes the problem – multithread the process.

6. There is significant overhead. Possible solution: reduce the total number of processes so a given response uses fewer messages.

Multithreading as a solution option appears frequently in the above list. As presented, those solutions would further raise overhead. One way to achieve multithreading and at the same time reduce the number of messages being sent is to create an architecture similar to case 1 but with a multi-threaded process.

6.3.3 Case 3: Single Multi-Threaded Process (SMTP) architecture

This case has a single multi-threaded process (Figure 6.6), with as many threads as are needed. Like the six-thread architecture, this architecture allows the two disks and the CPU to be used simultaneously. Also, more-urgent processing can now more easily preempt less-urgent processing. The results for this architecture shown in Figure 6.6 are quite encouraging: every response-time requirement is now being met. Also, although the total overhead is greater than for the single-thread architecture, that overhead is less than it was for the six-thread architecture of Figure 6.5. The only category of overhead which has increased beyond that for the six-thread architecture is semaphore overhead, because now *docMap*, the dictionary which indexes the data object for each document, must also be protected. In the six-thread architecture *docMap* does not need protection because the only access to it is from the single-threaded process *main*.

Looking at the metrics, the first thing to note is that by multi-threading the process, inversion is eliminated as a problem. For the path crossing into the process, there is a small time-averaged device need, 0.007. The device need occurs when a message arrives and has to wait because some token is executing on the CPU at a more-urgent deadline. 32% of the

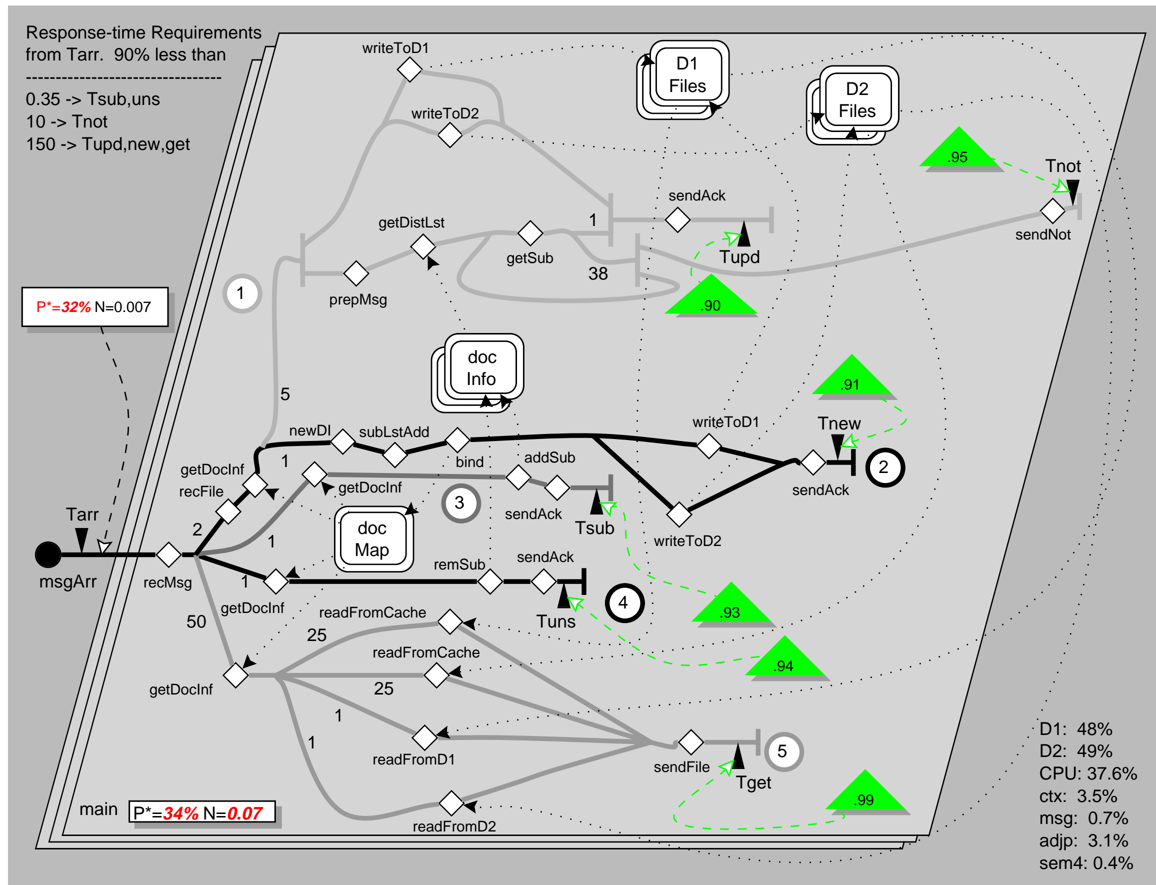

Figure 6.6: Use Case Map showing architecture with single multithreaded process and its evaluation (Case 3a)

device need is potentially poorly scheduled because one or both of the disks might be idle[1] while messages are waiting to be received.

The internal device metric reports a time-averaged device need of 0.07. Because it is the only scenario with an AND fork, we know that this device need is generated when scenario 1 is followed. 34% is potentially poorly scheduled. Three devices are potentially needed for an execution of scenario 1: the CPU and two disks. It is the disks that will be potentially poorly scheduled, as they may be idle while the CPU is generating, addressing, and sending notification messages. One might wonder whether a disk could be working on a scenario 5 retrieve-document request that arrived after a scenario 1 update-document request and is being served by another thread. This is not actually possible because, due to deadlines, the CPU would not be yielded to allow the retrieve-document request to start.

6.3.4 Case 4: Parallelism-in-updating architecture

To improve the SMTP architecture of Case 3, we could try to get rid of the 32% potentially-poorly-scheduled device need internal to the *main* process. This PPS device need occurs because an update to a single document (scenario 1) has potential parallelism which is not exploited: writing a file to disk could be parallel with generating, addressing, and sending notification messages to subscribers. To eliminate the PPS device need, we can create a separate process for one of these parallel subpaths. Figure 6.7 shows Case 4, with a separate disk process to do the disk writing when a document is updated. This is a good chance to emphasize that changing the concurrency architecture is usually very easy. In this case, the operations within UCM Navigator are simply to drag a handle on the original process to make it smaller, and then to click and drag a new process on the "canvas". After saving the file, PERFECT can be invoked again to evaluate the new architecture.

[1]The disks cannot be performing operations scheduled at a less-urgent deadline than that of any waiting message because disk-operation deadlines are taken from thread deadlines and threads when using a disk have a deadline based on the same specified delay value, 150ms.

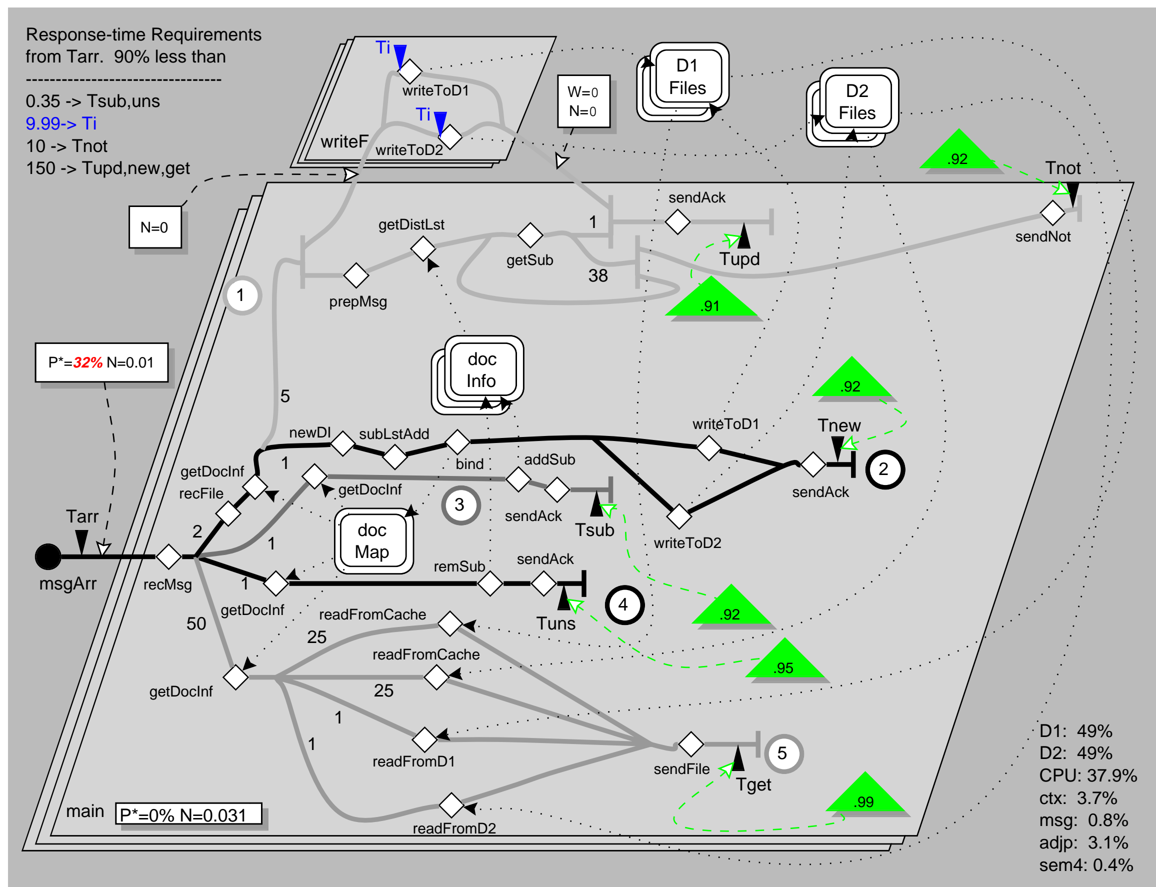

Figure 6.7: Use Case Map showing Parallelism-in-updating architecture and its evaluation (Case 4a)

Similarly to what was done in Figure 5.21 on page 89, intermediate response-time requirements were defined ending at activities *writeToD1* and *writeToD2*. The disk accesses are scheduled according to a deadline derived from the response-time requirement ending at *Tupd*, but the CPU work needed to actually schedule the accesses (receiving a message and calling the disk scheduler) is executed with a deadline derived from the intermediate response-time requirements. Why are the intermediate response-time requirements needed? Without them, the upper subpath performing the disk writing is scheduled according to a deadline derived from *Rupd*, after the lower sub-path dealing with notification messages which is scheduled according to a deadline derived from *Rnot*. This means that without the intermediate response-time requirements that all the processing of activities *prepMsg*, *getDistList*, and *getASub*, and the multiple instances of activity *sendUpd* for a particular document being updated would have to be finished before a thread in *writeF* could use the CPU to schedule disk writing, defeating the purpose of process *writeF*. Because scheduling disk operations requires minimal processing resources, the intermediate response-time requirements have an insignificant effect on the notification response times. To achieve the desired effect, the delay value of the intermediate response-time requirements must be less than that for *Rnot*. However, the smaller the delay value is, the more the disk-writing subpath can negatively impact the response times of other scenarios. The delay value was made smaller than *Rnot* but as close to it as possible.

The addition of process *writeF* has succeeded in eliminating PPS device need internal to process *main*. The success rate of response *Rupd* is no longer "on the border" at 90% but has risen to 91% (the mean response time has decreased from 93ms to 90ms). Counterbalancing this increase is a decrease in the success rate of response *Rnot*, down from 95% to 92% (the mean response time has increased from 3.6ms to 4.8ms).

The 32% PPS external device need of process *main* is similar to the SMTP design.

This is the best architecture evaluated so far for the system as specified, as it has highest

minimum rate of success for responses: 91%.

6.3.5 Case 5: Maximal-parallelism architecture

Can the Parallelism-in-updating architecture (case 4) be modified to make it better? The diagnostic metrics don't point to any improvement that can be made. The only metric which registers anything is the external device metric of process *main*, but as mentioned in Section 5.6, PPS device need external to a multi-threaded process does not always indicate that architectural improvements are possible.

There is, however, some remaining concurrency in the GCS specification which has not been exploited by the architecture. Specifically, the sending of each notification message, as well as the loop to actually retrieve the address of each subscriber, can potentially be done in parallel. Previous architectures have serialized in one thread all of this work for a given document being updated. Figure 6.8 shows an architecture with a third multi-threaded process created to send out notification messages, one thread per message. Thus a single update operation can activate as many of these threads as it can use.

The performance results of this Maximal-parallelism case are relatively poor for one processor (case 5a): the overhead has significantly increased and the response times have worsened over the previous two architectures. There is no real parallelism with just one processor, and there is a lot of overhead in triggering a thread for every notification.

The overhead hurts some subscription requests. Although *Rsub* is close to its specification, it has a high variance. Notice the interesting statistics reported for response *Rsub*: the mean response time is 1.86 times the response-time requirement's delay value, and yet 89% of responses are less than the delay value. Figure 6.9 shows a frequency distribution of some measured response-times for response *Rsub* when using this architecture. 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

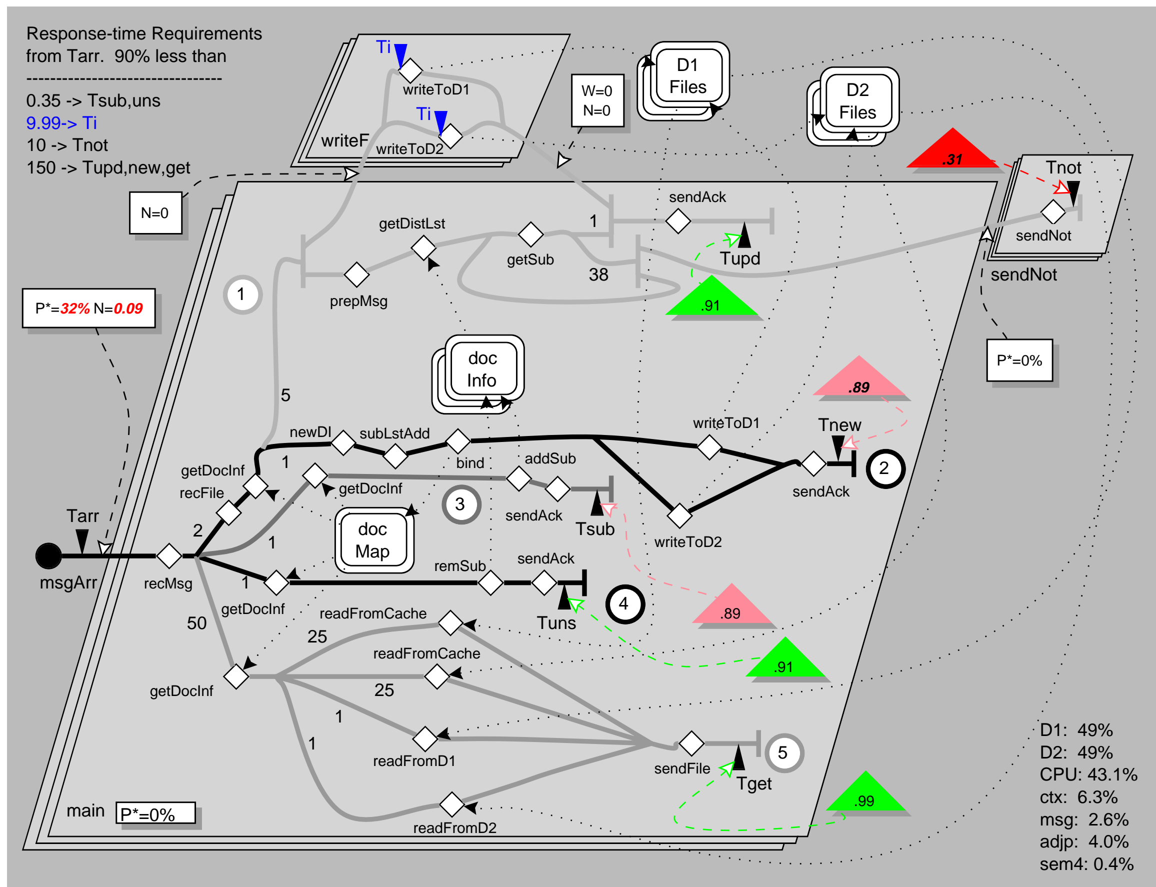

Figure 6.8: Evaluation of Maximal-parallelism architecture (Case 5a)

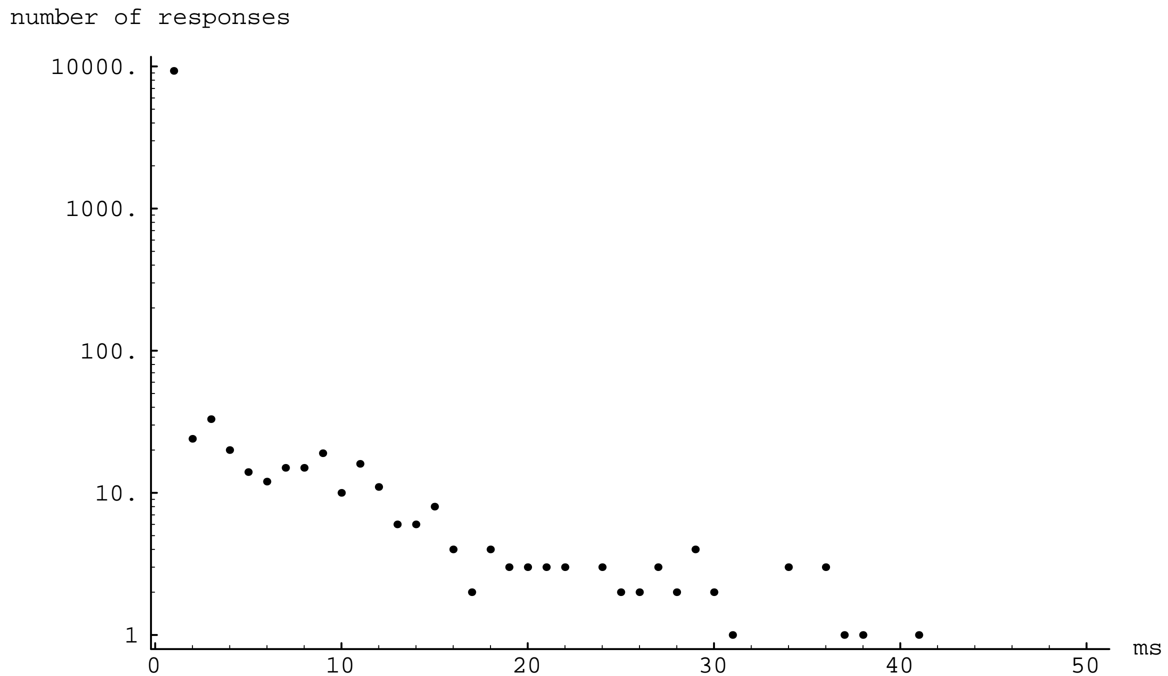

Figure 6.9: Frequency distribution of measured response times for *Rsub* in Case 5a

many subscribers is updated, the sending of notification messages becomes very urgent (in fact, significant numbers of notification responses miss their deadline) and this prevents subscription requests from being processed.

Multiple processors (cases 5b and 5c)

Multiple processors were considered in Cases 5b (two processors) and 5c (four processors). The results (in Table 6.1) show that the success rates for responses generally 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 98% or more of the responses within the specified time value. However the success rates for response-time requirement *Rnot* (sending update messages), even with 4 processors, do not compare favourably with those achieved using the SMTP or Parallelism-in-updating architectures 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.

Case 5 added concurrency to case 4 which, however, showed no indication of deficient concurrency. This Maximal-parallelism architecture indicates that it is not worthwhile to increase concurrency if the diagnostic metrics do not suggest such an increase.

6.4 Impact of workload change

It is also important to explore the impact of a change in the performance parameters on the performance of the architectures. This is because it is advantageous to choose an insensitive solution. Workload estimates often start with one set of figures, but the actual workload might be significantly different.

To demonstrate how the tool can help in this matter, the two best architectures (Cases 3 and 4) were tested with increased 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 4b in Table 6.1. The results are also shown in Figures 6.10 and 6.11. Notice the changes in the relative weights given to different branches of OR forks as compared to those in Figure 6.1. The average interarrival time has shrunken from 4ms to 2.1ms.

The results show that both architectures now miss meeting the requirements, especially for response *Rnot* which now has success ratios of only 75% for the SMTP architecture (down from 95%) and only 70% for the Parallelism-in-updating architecture (down from 93%). The reason for these changes is that the increased number of subscribers for documents has significantly increased the processor demand for sending notification messages to subscribers when a document is updated. The utilization of the processor rose to over 74% in both cases.

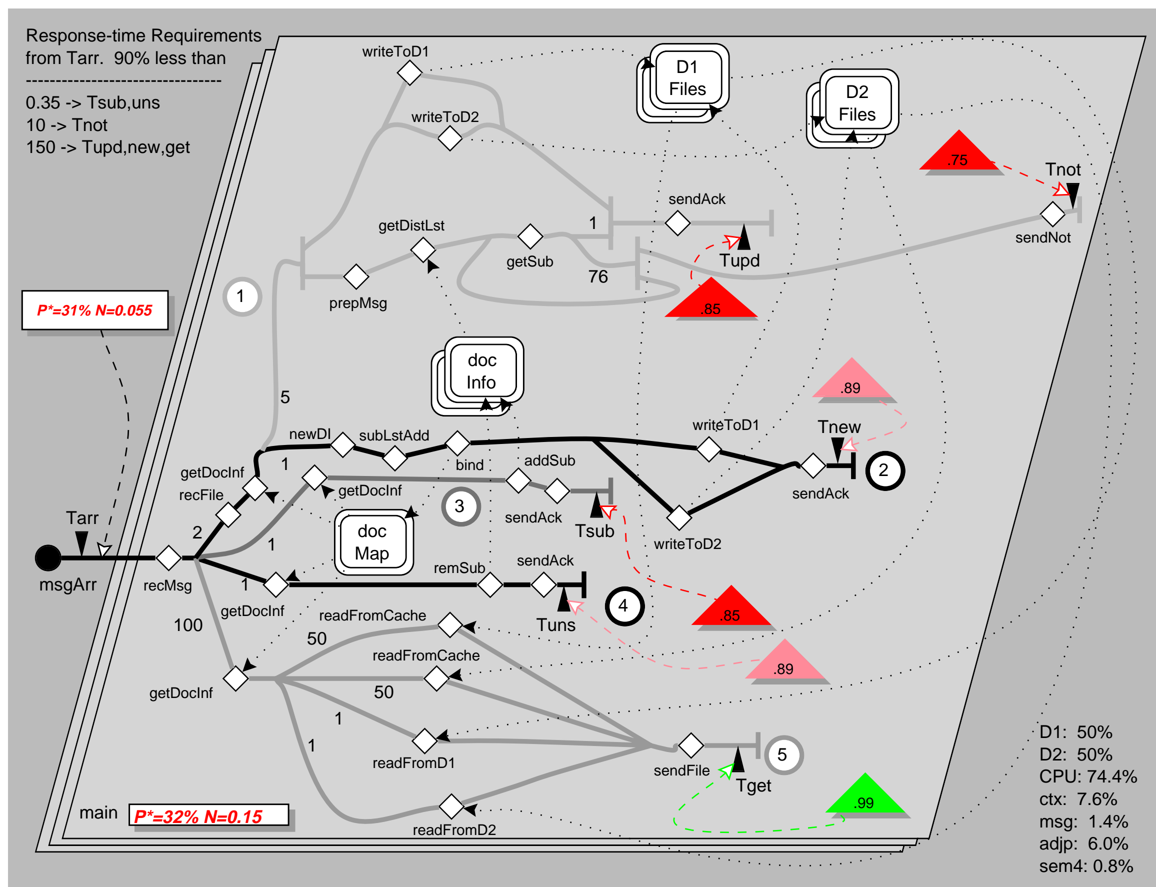

Figure 6.10: Evaluation of SMTP design with doubled subscribers (Case 3b)

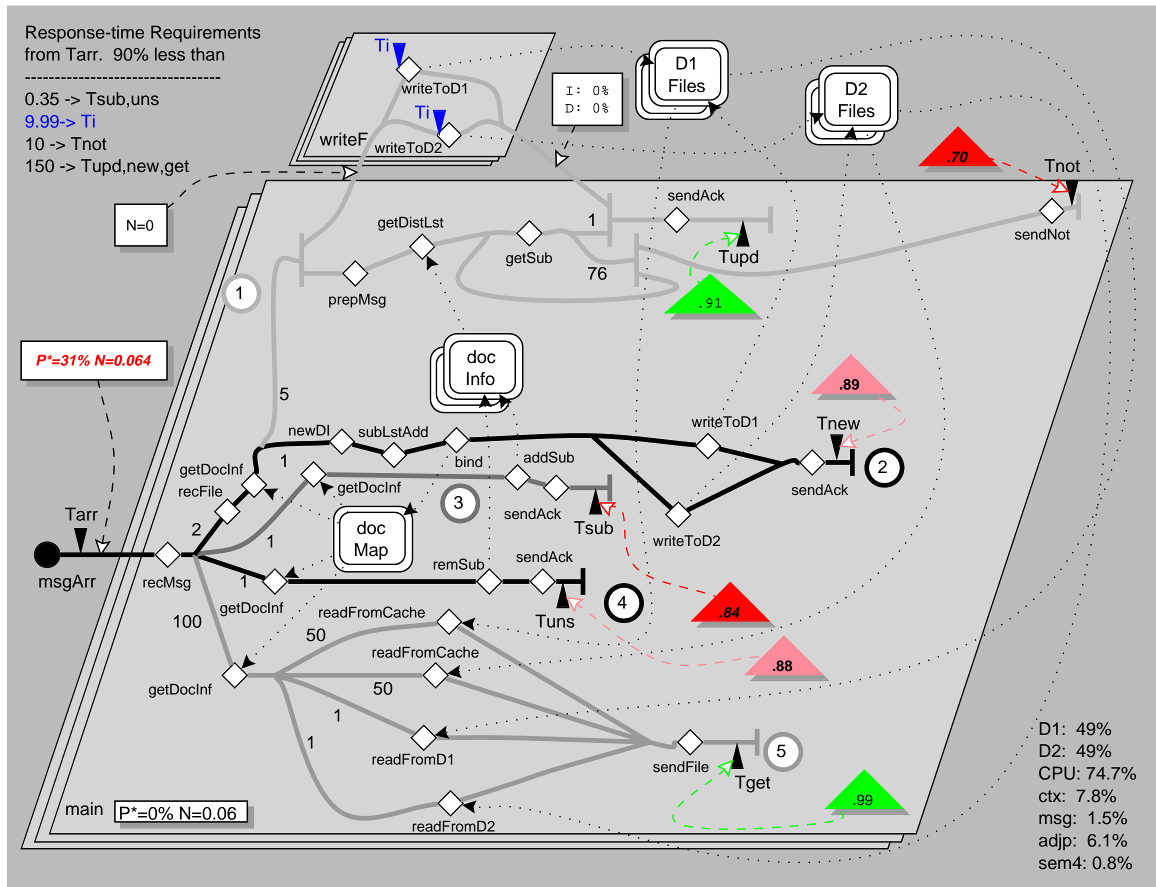

Figure 6.11: Evaluation of Parallelism-in-updating architecture with doubled subscribers (Case 4b)

The Parallelism-in-updating architecture still boosts the success ratio of response *Rupd*, up to 91% from 85% with the SMTP architecture, and this boost still comes at the expense of the success ratio of *Rnot*. This brings us to an interesting conclusion: whereas the Parallelism-in-updating architecture was deemed better with the original workload parameters, the SMTP architecture is better with the new workload parameters (better because the improvement for *Rupd* is not worth the deterioration for *Rnot*).

With respect to the metrics, in both architectures there was a significant increase in time-averaged device need. Internally to process *main* the time-averaged device need approximately doubled for both architectures. This makes sense as there are now twice as many subscribers per document. The external time-averaged device need between *msgArr* and activity *recMsg* also increased, implying messages more often waiting to be received by a thread (the new workload parameters generate more request messages.) The percentage of these device needs which is potentially poorly scheduled remained relatively constant before and after the workload parameters were changed. Table 6.2 adds metric values to other results for all simulations of the SMTP architecture.

Case num	Case name	Utilizations (%)							Response times (mean time in ms / % meeting requirement)						Device Metrics	
		Disk1	Disk2	CPU					Rupd (dv=150)	Rnot (dv=10)	Rnew (dv=150)	Rsub (dv=0.35)	Runs (dv=0.50)	Rget (dv=150)	P* (in %) : N	
				Total	ctxsw oh	msg oh	adj dl oh	sem4 oh							msgArr →	main (int.)
3a	SMTP	48	49	37.6	3.5	0.7	3.1	0.41	93/90	3.6/95	90/91	0.335/93	0.369/94	7.5/99.4	32:0.007	34:0.07
3b	- double subscribers	50	50	74.4	7.6	1.4	6.0	0.78	***105/85***	***7.1/75***	***97/89***	***0.44/85***	***0.46/89***	10.1/99.4	31:0.055	32:0.15
3c	- - with 2 processors	49	49	48.5 +23.9	5.5	1.4	6.0	0.80	***96/89***	***6.2/81***	91/91	0.32/98	0.34/99.6	5.4/99.6	32:0.001	53:0.13
3d	- - 2X faster processor	50	49	36.4	3.1	0.7	3.0	0.40	92/91	3.2/96	90/91	0.17/99.8	0.19/99.8	4.8/99.6	32:0.007	33:0.07

Table 6.2: Output reported by PERFECT tool for sub-cases of SMTP architecture

With the changed workload, we have not found a concurrency architecture which meets all of the requirements using the default hardware configuration. To satisfy the requirements, we can try scaling up the hardware configuration.

One option in this regard is to add a second processor as we had tried above with the Maximal-parallelism architecture. Rows 3c and 4c of Table 6.1, as well as Figure 6.12 show the results obtained with the added processor. Note that although *Rsub* and *Runs* now have very high success ratios, that the success ratio of *Rnot* is still inadequate. This can be explained by considering that the entire processing of the notification responses for a given "update document" request is being processed by a single thread running on one processor. If a processor is not fast enough to complete the required processing within the specified delay value, then any additional processors cannot enable the response requirement to be achieved.

The failure of the first scaling option suggests that a second scaling option, using a faster processor, be tried. To this end we have evaluated both the SMTP and Parallelism-in-updating architectures using a single processor running at twice the speed. The evaluation results are shown in rows 3d and 4d of Table 6.1 and Figures 6.13 and 6.14. Although the SMTP architecture has PPS internal device need whereas the Parallelism-in-updating architecture does not, the success ratios of the two architectures are similar and all requirements are met.

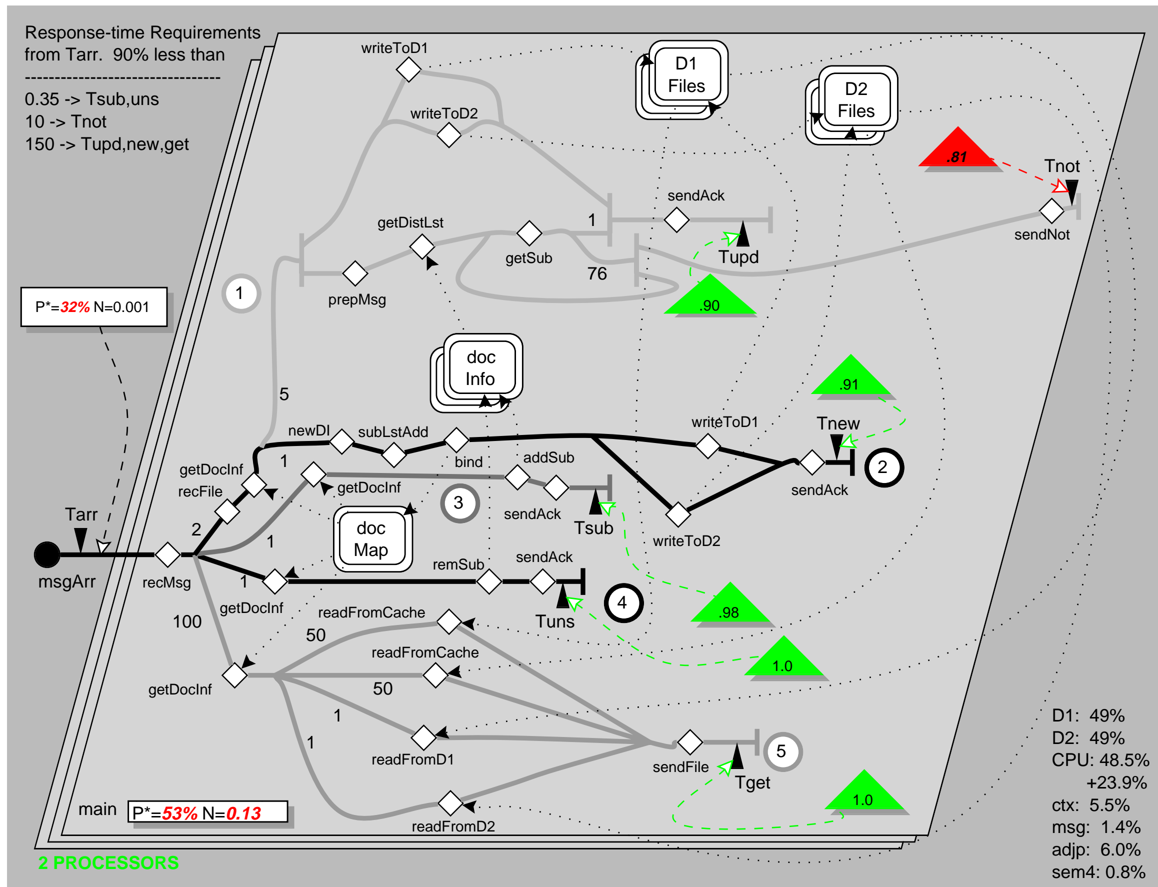

Figure 6.12: Evaluation of SMTP architecture with doubled subscribers and two processors (Case 3c)

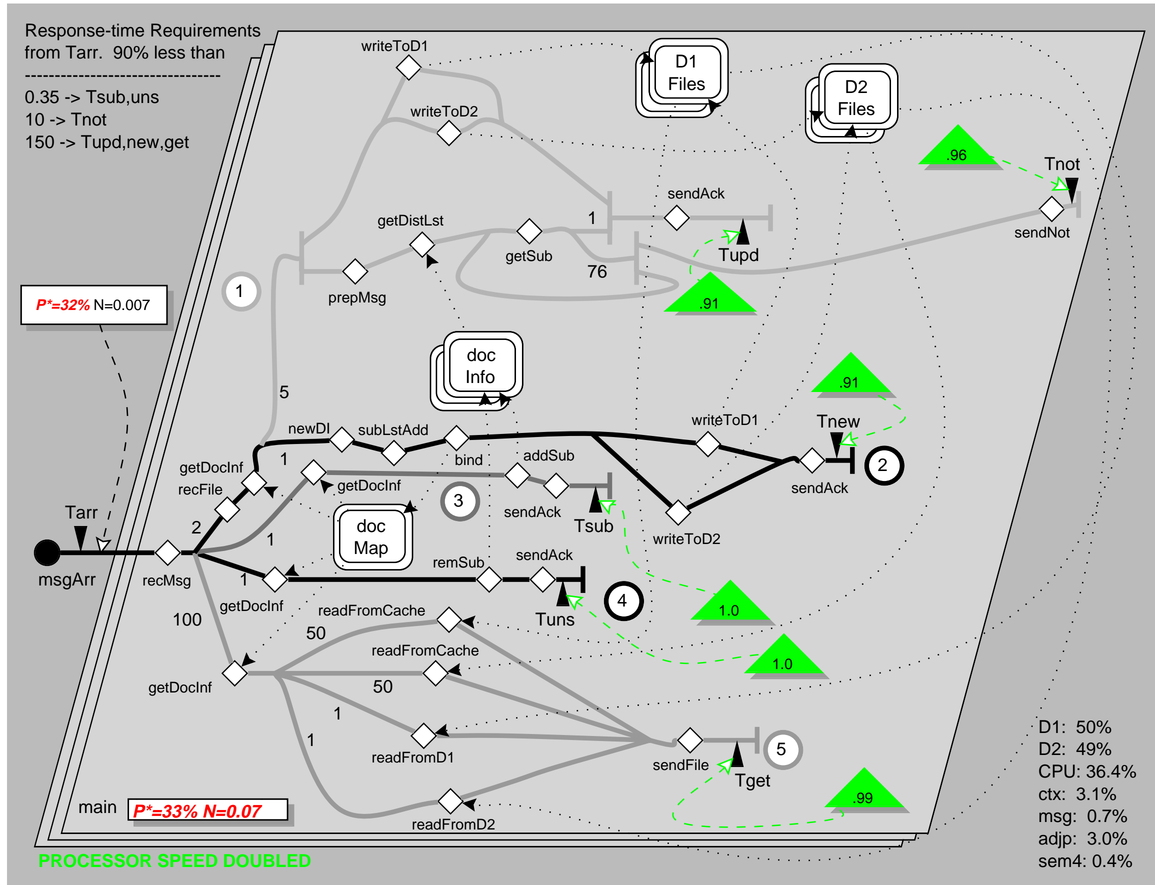

Figure 6.13: Evaluation of SMTP architecture with doubled subscribers and double-speed processor (Case 3d)

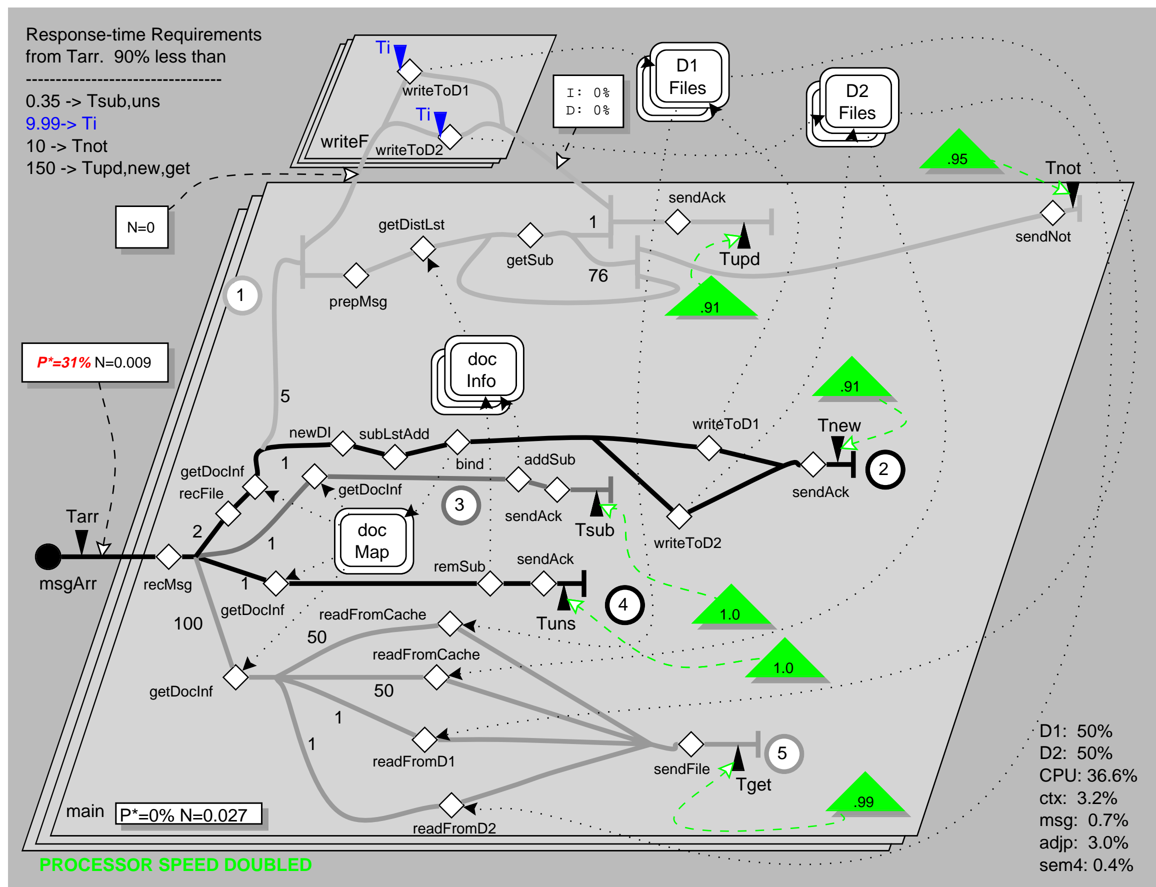

Figure 6.14: Evaluation of Parallelism-in-updating architecture with doubled subscribers and double-speed processor (Case 4d)

Chapter 7

Conclusions

7.1 The overall view

This thesis has described and demonstrated a technique for evaluating a concurrency architecture for a system that executes a given set of scenarios. The evaluation concentrates on the decision of how to partition the scenario elements among concurrent processes, and does not address the details of designing the processes. It attempts to find "very good" executions of the scenarios in the architecture, using a simulated virtual implementation and (pre-emptive) earliest-deadline first scheduling. The construction of the virtual implementation and the prioritized execution are justified heuristically; no attempt has been made to show theoretically that the executions are optimal, given the architecture.

The evaluation is based on achievement of soft deadlines, and includes the percentage of deadlines achieved, and resource utilizations and overhead costs.

As well, several special metrics were defined to detect inadequate concurrency. These metrics were also developed from heuristics and evaluated on a variety of examples. A substantial example was used to explore a wide range of issues in concurrency. It demonstrated that the metrics could indeed identify concurrency problems in an architecture, and showed

that if one adds concurrency to an architecture when such additional concurrency is not indicated by the metrics, then performance degradation may result.

7.2 Individual contributions

The contributions of this thesis are:

7.2.1 Extensions to the Use Case Map notation

I have proposed a number of extensions to the Use Case Map notation which allow or assist performance information to be annotated, including

- Timestamp points[1] (in Section 3.2.4). In order to specify performance assertions for a Use-Case Map, we need some way to identify points along paths at which and between which the assertions are made. Required throughputs and response-times are obvious assertions. Timestamp points fulfill this need.

- Responses and response-time requirements (also in Section 3.2.4) . It is useful to be able to explicitly and graphically identify response-time requirements on a Use Case Map. In this way an architect or designer can easily identify performance critical path segments, and can spot errors in response-requirement specification. The response-requirement extension to UCMs (a labelled dotted line with an arrowhead) provides such an identification. Performance predictions or measurements can be attached to the response requirement annotation as has been done in this thesis.

- The ability to show at the level of a stub that its plugin will assign at least one responsibility to an anchored data object (see Section 3.2.2). If multiple tokens travelling on a UCM can touch a data object simultaneously, the data object should be protected

[1]The actual symbol shape on a path was suggested by R. Buhr during a private conversation

in some way to ensure it is seen with a consistent state. To easily identify such data objects if they are accessed in a plugin, that access is indicated using a dotted line from the stub. A number of types of access to the data object, such as read and write, can be indicated using arrowheads pointing either to the data object or to the stub.

- Display of the criteria by which a choice is made at an OR fork (see Section 3.2.3). Although a way to determine which branch of an OR fork is followed by a token is not necessary to define the possible scenarios that can occur in a system, such determination (or at least relative frequencies) is necessary when we want to evaluate the performance of a UCM. To make this determination we can either give a relative frequency for each branch, or associate each branch with a certain value held by an element of data belonging to the token.

7.2.2 Virtual implementation

A concept of virtual implementation has been described, which transforms a set of scenarios and concurrency architecture decisions into an operational form which can be evaluated (see Chapter 4).

The goal of the virtual implementation is not primarily an automated design to be followed by an implementer, but an ideal execution of the specification. The performance achieved by the virtual implementation becomes a target for the implementer who has many additional constraints (such as language, operating system, etc.) to contend with. For example, although recognizing experimental systems such as Real-Time Mach and Chorus [46], the author knows of no commercial operating-system kernel supporting EDF thread scheduling, so the actual scheduling may have to be tuned to approach the evaluation.

The virtual implementation is also an exploration of ideal design, for example resolving the implementation of potentially parallel paths in a sequential process.

The virtual implementation was constructed using :

- A set of rules which control the creation, movement, cloning and destruction of tokens.
- A set of rules which ensures that at any given moment each thread is scheduled with an intelligent deadline.

The virtual implementation is instrumented so that it can report

- the mean response time and fraction of “successful” responses for each type of response.
- the amount of concurrency-related overhead observed: both the total concurrency-related overhead, and the overhead broken down by type (context switching, messaging, deadline adjustment, and semaphore operations used for protecting access to shared data objects).
- The utilization of each device instance.

7.2.3 Metrics to detect concurrency problems

Three measures have been invented and tested, with the goal of identifying defects in concurrency:

- An inversion metric which reports, at each significant point where a path crosses into a process, the time-averaged number of waiting tokens and the percentage of that which is deadline-inverted (see Section 5.4.1).
- An external device metric which reports at each point where a path crosses into a process the time-averaged need for devices by tokens waiting at that point, and the fraction of that need which is potentially poorly scheduled (see Section 5.4.2).
- An internal device metric, similar to the external device metric, but which measures device need for tokens waiting inside the threads of a process (see Section 5.4.3). The fraction of that internal need which is potentially poorly scheduled is also measured.

These metrics complement the utilization and concurrency-related overhead measurements reported above.

Tutorial examples were constructed, evaluated, and diagnosed to highlight certain concurrency problems:

- Deadline inversion (see Section 5.1).
- Excessive serialization due to only one token being able to travel along a given path segment at a given time (see Section 5.2).
- Excessive serialization after an AND fork (also see Section 5.2).
- Excessive concurrency-related overhead (see Section 5.3).

Changes to UCMs of the examples were proposed, evaluated and diagnosed to show ways in which the concurrency problems can be resolved (see Section 5.5).

Another tutorial example was presented for which small changes to the model would produce at a certain point along a path either no concurrency problems, a potentially poorly scheduled device need problem, or an inversion problem (see Section 5.5.4).

7.2.4 The PERFECT tool

A tool named PERFECT (PERFormance Evaluation by Construction Tool) was developed, to allow concurrency architectures for substantial systems to be evaluated and diagnosed by anyone (see Section 4.6).

PERFECT reads in XML files containing UCMs which have been entered using the UCM Navigator graphical editing tool [47]. Documentation and additional information about UCMs and the UCM Navigator is available on the web site www.usecasemaps.org.

7.2.5 Case study

A case study was constructed to show how the methodology and the PERFECT tool can detect multiple concurrency problems in a substantial application (see Chapter 6). The application constructed was a group communication server. A series of concurrency architectures were proposed to show how the architecture can be tuned to yield good performance. In the case study, increasing concurrency in the architecture at first improved performance but eventually caused performance to diminish. All the metrics were demonstrated in a variety of situations.

7.3 Straightforward generalizations

Below are listed some ways in which the work presented in this thesis can be readily generalized.

- Including software services assigned to some process in addition to the service by devices currently included in the external device metrics. The metrics could then be renamed external service metrics. The expanded metrics would identify potentially poorly scheduled service need. The server processes could either be local or remote.

- Covering distributed systems evaluation. The generalization would use (for evaluation purposes) global time for EDF scheduling, and the external service metrics would include remote services as described above. Operations on shared data objects might be remote.

- Being able to specify different mixes of loads under which the response-time requirements should hold. For example, consider a system with the following time-dependent loads:

  - During the day, the following peak workload: 2 type A requests/second and 27 type B requests/second.
  - At night, a different peak workload: 5 type A requests/second and 9 type B requests/second.

- Using a notation other than UCMs to specify scenarios and concurrency architectures. One possibility is a custom variant of Petri Nets.

7.4 Future Research

The following sections present some possibilities for future research arising from this thesis.

7.4.1 Validation on a system supporting EDF scheduling

In order to validate the evaluations, we should run some test applications using an operating system kernel which supports EDF scheduling. The case study is one such test application, but would require that the ACE framework run on the EDF kernel. Some experimental versions of Real-Time Mach support EDF scheduling, so this would provide one option. Real-time Mach also supports the style of messaging used in the virtual implementations.

7.4.2 Displaying results in the UCM Navigator

It would be useful to be able to send evaluation and diagnostic results back to the UCM Navigator. Ideally, one could invoke PERFECT from UCMNav, and the results could be fed back into the XML file and displayed in UCMNav. Currently, results have to be entered by hand onto a UCM diagram in order to see them in context of the diagram.

7.4.3 Automatic identification of efficient concurrency architectures

It would be useful to develop an algorithm which proposes a concurrency architecture and an initial set of hardware, and automatically analyzes evaluation and diagnostic results and searches for efficient concurrency architectures. Such an algorithm would first choose hardware components of sufficient speed to handle the offered load and also allow all responses to finish within their requirements assuming no contention for devices. The algorithm would then choose an initial concurrency architecture, possibly one single-threaded process as was done in the case study. The algorithm would then iterate through evaluating and diagnosing an architecture, would choose the process which had the greatest metric problems, and would either split the process or multi-thread it.

When architectural alterations no longer yielded improvements, if all of the responses were not meeting their requirements the algorithm would analyze which responses were failing and would select one or more devices to make faster.

Bibliography

[1] 4th workshop on process algebras and performance modelling, 1996.

[2] Z.100 (11/99). *Specification and Description Language (SDL).* ITU-T, Geneva, November 1999.

[3] Z.120 (1999). *Message Sequence Chart (MSC).* ITU-T, Geneva, November 1999.

[4] Robert K. Abbott and Hector Garcia-Molina. Scheduling i/o requests with deadlines: A performance evaluation. In *IEEE Real-Time Systems Symposium*, pages 113–124, December 1990.

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

[6] V.S. Adve and J. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of SIGPLAN98*, Montreal, June 1998.

[7] D. Amyot, F. Bordeleau, R.J.A. Buhr, and L. Logrippo. Formal support for design techniques: a timethreads-lotos approach. In *FORTE VIII, 8th International Conference on Formal Description Techniques*, pages 57–72. Chapman & Hall, 1995.

[8] Daniel Amyot and Andrew Miga. Use case maps linear form in xml. http://www.UseCaseMaps.org/UseCaseMaps/xml, 1999.

[9] Marco Baldassari and Giorgio Bruno. Protob: An object oriented methodology for developing discrete event dynamic systems. *Computer Languages*, 16(1):39–63, 1991.

[10] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison Wesley, 1998.

[11] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. *Queueing Networks and Markov Chains, Modeling and Performance Evaluation with Computer Science Applications.* John Wiley & Sons, 1998.

[12] G. Booch. *Object-Oriented Design.* Benjamin/Cummings, second edition edition, 1994.

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

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

[15] R.J.A. Buhr and R.S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice Hall, `http://www.usecasemaps.org/usecasemaps/pub/ucm_book95.pdf` edition, 1996. 302 pages.

[16] C.M. Woodside and R. Romaniuk. An activity sequence network specification tool for software engineering. In *Proc. Can. Conf. on Electrical and Computer Engineering*, September 1990.

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

[18] J.N. Daigle, S.S. Liu, and K.W. Wong. Design alternatives for communication processing systems. In *GLOBECOM'83 Conf. Rec.*, pages 351–359, San Diego, CA, Nov.–Dec. 1983.

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

[20] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.G. Sevcik. *Quantitative System Performance*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, 1984.

[21] Hesham M. El-Sayed. *A Framework For Automated Performance Engineering of Distributed Real-Time Systems*. PhD thesis, Carleton University, Ottawa, Ontario, CANADA, October 1999.

[22] G. Estrin. A methodology for the design of digital systems, supported by SARA at the age of one. In *AFIPS Conference Proceedings, NCC*, 1978.

[23] G. Estrin, R.S. Fenchel, R.R. Razouk, and M.K. Vernon. Sara (system architects apprentice): Modeling, analysis, and simulation support for design of concurrent systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, February 1986.

[24] D. Coleman et al. *Object-Oriented Development: The Fusion Method*. Object-Oriented Series. Prentice Hall, 1993.

[25] E.Deelman et al. POEMS: end-to-end performance design of large parallel adaptive computational systems. In *Proc. of First International Workshop on Software and Performance (WOSP98)*, pages 18–30, October 1998.

[26] Haritsa et al. Mandate: Managing networks using database technology. *IEEE Journal on Selected Areas in Communications*, 11(9):1360–1372, Dec. 1993.

[27] B. Ghribi. A model checker for lotos. M.sc. thesis, Dept. of Computer Science, University of Ottawa, Ottawa, Canada, 1992.

[28] G.M. Yee and C.M. Woodside. A transformational approach to process partitioning using Petri nets. In *Proc. Int. Comp. Symposium 90 (ICS90)*, pages 395–401, December 1990.

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

[30] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, July 1987.

[31] David Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Second IEEE Symposium on Logic in Computer Science*, volume 8, pages 54–64, New York, July 1987. IEEE Press.

[32] Jayant R. Haritsa, Miron Livny, and Michael 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.

[33] C.A.R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.

[34] Jiandong Huang, John A. Stankovic, Don Towsley, and Krithi Ramamritham. 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.

[35] Neil Hunt. Performance testing c++ code.
http://www.rational.com/products/quantify/prodinfo/whitepapers

[36] I. Jacobson, M. Christeron, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[37] J.E. Neilson, C.M. Woodside, D.C. Petriu, and S. Majumdar. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Transactions on Software Engineering*, 21(9):776–782, September 1995.

[38] Ben Kao and Hector Garcia-Molina. Subtask deadline assignment for complex distributed soft real-time tasks. In *14th International Conference on Distributed Computing Systems*, pages 172–181, June 1994.

[39] Ben Kao and Hector Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1268–1997, December 1997.

[40] R. Kazman, M. Klein, and P. Clements. Evaluating software architecture for real-time systems. *Annals of Software Engineering*, 7, 1999.

[41] Takuro Kitayama, Tatsuo Nakajima, Hiroshi Arakawa, and Hideyuki Tokuda. Integrated management of priority inversion in real-time mach. In *Proc. IEEE Real-Time Systems Symposium*, December 1993.

[42] J. Leung and J. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1992.

[43] C.L. Liu and J.M. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Jounal of the A.C.M.*, 20(1), January 1973.

[44] Clifford W. Mercer. An introduction to real-time operating systems: Scheduling theory. http://www.cs.cmu.edu/afs/cs/project/art-6/www/publications.html, November 1992.

[45] C.W. Mercer and H. Tokuda. The arts real-time object model. In *11th IEEE Real-Time Systems Symposium*, pages 2–10, December 1990.

[46] Yves-Olivier Metais. Conception et implantation d'un ordonnanceur a echeance au sein du noyau chorus. Memoire, Conservatoire National des Arts et Metiers Paris, March 1994.

[47] 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.

[48] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[49] A.K. Mok. The decomposition of real-time system requirements into process models. In *CH2105-5/84*, pages 125–134, 1984.

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

[51] HweeHwa Pang, Miron Livny, and Michael J. Carey. Transaction scheduling in multiclass real-time database systems. In *IEEE Real-Time Systems Symposium*, pages 23–34, December 1992.

[52] D. Peng and K.G. Shin. Modeling of concurrent task execution in a distributed system for real-time control. *IEEE Trans. on Computers*, C-36(4):500–516, April 1987.

[53] J. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, Reading, Mass., 1986.

[54] J. Quemada, S. Pavon, and A. Fernandez. Transforming lotos specifications with lola: The parametrized expansion. In K. J. Turner, editor, *Formal Description Techniques, I*, pages 45–54. IFIP/North-Holland, 1988.

[55] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[56] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[57] Douglas C. Schmidt. The adaptive communication environment: an object-oriented network programming toolkit for developing communication software. In *Proceedings of the 12th Sun User Group Conference*, San Jose, California, June 1993.

[58] W. Craig Scratchley and Huijun Yang. A group communication server implemented with ace. A project report for a course taught by Prof. Dorina Petriu at Carleton University, Ottawa, Canada, December 1998.

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

[60] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), 1990.

[61] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[62] S.S. Yau and M.U. Caglayan. Distributed software system design representation using modified Petri Nets. *IEEE Trans. on Software Engineering*, SE-9(6), November 1983.

[63] Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *Proceedings of USENIX technical conference*, June 1998.

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

[65] H. Tokuda, C. Mercer, Y. Ishikawa, and T. Marchok. Priority inversions in real-time communication. In *10th IEEE Real-Time Systems Symposium*, pages 348–359, December 1989.

[66] Kenneth J. Turner. *Using Formal Description Techniques, an Introduction to Estelle, Lotos, and SDL*. Wiley, Chichester, England, 1993.

[67] M. Vernon, E. de Souza e Silva, and G. Estrin. Performance of asynchronous concurrent systems: The UCLA graph model of behavior. *Proc. 9th Int. Symp. on Comp. Perf. Modelling, Measurement, and Evaluation*, pages 153–172, May 1983.

[68] Philip S. Yu, Kun-Lung Wu, Kwei-Jay Lin, and Sang H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

Colophon

This thesis was prepared by the author on an Intel-based computer running the OPENSTEP operating system. LaTeX2e files were edited using `TeXEdit`, processed with the help of `TeXWorks`, and proofed using `HyperTeXview`. The bibliography was prepared with `bibtex` and the OPENSTEP program `Bibliography`. Figures were drawn using `Diagram!2`, which was created by Lighthouse Design Ltd. before it was acquired by Sun Microsystems, Inc. Tables were created using the `Tables` program which Lighthouse Design bought from a company named Xanthus. The logarithmic plot in Chapter 6 was prepared using `Mathematica` on a Sun workstation.

The final typeset thesis was translated into PostScript using the teTeX distribution of `dvips`, and printed on Eaton 25% Cotton Fibre Laser Paper using a HP Color LaserJet Printer and a Lexmark OptraN Printer.

The `.pdf` version of this thesis was created by first generating a postscript file containing postscript type1 fonts instead of metafont fonts, and then converting from PostScript using Adobe `distill`.