Efficient Analysis of Temporal Properties for Real-Time Systems

—

A Formal Framework, Supporting Protocols, and an Implementation

Habilitationsschrift
zur Erlangung der Lehrbefähigung
für das Fach Informatik

vorgelegt dem Rat der Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von

Frank Müller, Ph.D.

geb. am 24.3.1966 in Berlin

Präsident
der Humboldt-Universität zu Berlin
Prof. Dr. Jürgen Mlynek

Dekan
Prof. Dr. Bodo Krause

Berlin, den 15. September 2000

Gutachter/Gutachterinnen:

1.

2.

3.

To Lisa

Acknowledgements

I would like to to express my gratitude to Prof. Dr. Klaus Bothe for his advice over the past years. I would also like to thank the students that I supervised for their assistance on implementations, namely Thomas Lehmann for enhancing the cache simulator, Jens Rumstich for improving the timing analyzer, Claus Wagner for his work on synchronization protocols, and Niklas Hahn and Thomas Röblitz (as well as students from various project classes) for their help on DSM-Threads. Chris Healy provided continued support for the timing analyzer. Joachim Wegener proved to be an inspiring colleague during our comparative study between static analysis and evolutionary testing.

Contents

List of Tables

List of Figures

Abstract

Schedulability theory for real-time and embedded systems has matured during the past decades to provide a sound base for the verification of the temporal properties of a real-time system, *i.e.*, the verification that each task can complete execution within its deadline. However, this theory is based on the assumption that the worst-case execution time (WCET) of a task be known a priori. This work focuses on analytical methods to determine the WCET statically.

First, a formal framework for safe timing analysis is derived, an implementation is provided and results on timing analysis for uniprocessors are presented. It is shown that a component-based approach separating the process into several steps of the analysis based on the modeled entities results in an efficient and practical solution of the problem. Then, an introduction to the overall toolset for static timing analysis is given followed by a more detailed derivation of methods for performing the static cache analysis. For this purpose, a data-flow framework is developed that provides classifications of cache references suitable for the static prediction of the WCET. A brief overview of the integration of these results into pipeline simulation and path analysis concludes the description of the framework for uniprocessor timing analysis.

Second, the problem of transferring the results of timing analysis for uniprocessors to distributed systems is discussed. Key services of distributed systems are identified that require additional protocols to provide guarantees for real-time systems. In particular, a novel distributed protocol to arbitrate resource requests is introduced that provides a decentralized solution to the mutual exclusion problem while supporting system-wide priorities. Finally, open problems in timing analysis are identified and a road map for future research on timing analysis is presented.

Prologue

In the following, a number of our contributions in the area of real-time systems are summarized and related to each other. While some aspects are discussed in detail hereafter, others are described in more detail in previous publications. The following publications constitute an integral part of this work:

1. the work on static instruction cache analysis [52] for Chapter 4;
2. the work on pipeline and path analysis, in particular with regard to cache issues, within the timing analyzer [19] for Chapter 5;
3. the comparison of static and dynamic timing analyses [68] for Chapter 6;
4. the work regarding the DSM-Threads system [45, 46] for Chapter 11;
5. and the work on distributed synchronization [48, 50] for Chapter 12.

These publications should be consulted in conjunctions with the corresponding chapters. These chapters provide an overview of the work and its relation to other chapters while the publications present the formal and technical details.

In addition, further detail about DSM-Threads is provided in [49, 51] and [58]. These publications do not constitute an integral part of this work but may be consulted for additional reading.

The following statements are made in accordance with the bylaws for "habilitations". The publications in reference to items 2 and 3 in the above list include collaborative work with co-authors. The original contributions of the author Frank Müller are distinguished from the contributions of the other co-authors follows:

- The issues in reference to cache analysis and the integration of cache analysis results into the timing analyzer within publication [19] constitute a contribution by Frank Müller. This includes contributions on handling cache behavior during pipeline simulation, path analysis, loop analysis and program analyses.

- The issues in reference to static timing analysis (SA) within publication [68] constitute a contribution by Frank Müller. This includes the description of SA, experiments with SA and the aspects regarding SA within the comparison between SA and evolutionary testing (ET).

Introduction

Real-time systems can be characterized as special-purpose application programs that have to meet certain temporal criteria. Many real-time applications are deployed in customized environments where their objective is to control peripheral devices either autonomously or with the aid of human interaction. The controlled devices range from aircraft components over machines on the factory floor to games and toys.

A real-time application is characterized by temporal constraints inherent to the design of the application. These constraints may be imposed by designers and engineers or they are derived from specifications of sensory devices and actuators. A real-time application typically consists of a number of system tasks, *e.g.*, distinguished by their functionality in obtaining sensor data, performing calculations or controlling actuators. Tasks may be invoked periodically or aperiodically. The former refers to repeated invocation of the task according to the period, *e.g.*, every 2 ms, while the latter may be triggered by events. Once invoked, a real-time task has a deadline, which specifies the point in time where its execution must be completed.

Soft and hard real-time systems are being distinguished depending on the criticality of the application. Hard real-time systems may be subject to environmental threat or loss of life if temporal specifications cannot be met, *i.e.*, if a deadline is missed. In contrary, soft real-time systems do not have any catastrophic impact on the environment, even if deadlines are occasionally missed. In fact, soft real-time systems are often designed to tolerate sporadically missed deadlines.

The task set of a real-time system may be specified by certain parameters, such as periods, deadlines and worst-case execution times (WCET). The WCET specifies the longest possible execution time of a task. Schedulability theory then provides a sound base to determine if the temporal specifications of such a system can be met. Schedulability analysis determines if the deadline of each task can be met under any possible system load, given the specifications.

The verification of the temporal aspects of a real-time system depends on the assumption that the WCET of a task be known *a priori*. Determining the WCET of a task is non-trivial, especially in the presence of modern computer architectures. Pipelined execution may result in temporal overlap of instruction execution and latencies of memory references, in particular due to caches, which could stall execution. In addition to the WCET, the best-case execution time (BCET) may also be utilized, *e.g.*, to ensure that sensory data is sampled only within certain intervals if sensors may supply undefined (and potentially harmful) readings when logic values have to stabilize first. The task of determining the extreme execution times (WCET and BCET) is called timing analysis. This document focuses on the progress of timing analysis of real-time systems, which is a prerequisite for temporal verification through schedulability analysis.

0.1 Motivation

Systems with sensory inputs controlling actuators through their output for a specific application are termed embedded systems. Embedded systems are currently one of the strongest areas of growth. The number of deployed embedded processors has outnumbered the number of new general-purpose processors for personal computers and workstations since 1995 ?? [16]. In 1999, even the sales value of the embedded systems exceeded that of the PC market, and the trend is predicted to continue in the future. Embedded systems will enter many areas of our daily lives, ranging from appliances to the automotive sector. In these areas embedded processors are announced to increase by multiple orders of a magnitude.

Real-time systems are a subset of embedded systems. In both systems, sensors provide input, which is utilized to perform calculations in order to determine and control the actions of actuators. Real-time systems differ from non time-critical embedded systems in that their timing constraints are critical to ensure system functionality and, in particular, system safety. For example, temporal constraints may not be an issue for toys, certain games and even selected industrial or aircraft functions. Jittery images of games and delays when switching on non-critical appliances in planes may be annoying but do not affect the functionality or safety. But aircraft control instruments for fly-by-wire designs or automotive appliances such as anti-blocking systems (ABS) for breaks have to function within certain temporal constraints in order to ensure safety.

These examples illustrate the significance of the verification of temporal aspects for hard real-time systems. In addition, schedulability analysis may also become more and more pertinent for soft real-time system. The motivation for applying temporal verification may, however, differ. Cellular appliances, such as mobile phones, are an exploding market. Here, consumer acceptance may be the driving force behind applying schedulability analysis. A cellular phone that loses connections frequently may cause severe economical losses for a company. Switches on network connections that sporadically malfunction or result in reduced bandwidth may cause customers to change providers (or providers to change backbone suppliers). A game device with jittery images may adversely affect consumer acceptance. These examples illustrate that economical reasons may cause methods originally derived for verification within hard real-time systems to be applied in soft real-time environments.

Timing analysis is the underlying method required before schedulability analysis can be performed. Methods for timing analysis may, therefore, remain a focus in research on hard real-time systems as well as receive increased attention within soft real-time systems.

0.2 Problem Statement

The area of timing analysis investigates methods to determine the extreme execution times of a task or some code portion. The execution time of a piece of code may either be determined experimentally through dynamic testing or analytically through static methods. The former approach, dynamic timing analysis, has been used in the past. Direct measurements of the execution time through on-chip timers may provide concrete results for actual executions. Indirect timing methods, *e.g.*, through logic analyzers, may reduce interference with the results of the measurements. Program instrumentation to record instruction execution frequencies and combine the results with cycle-level information about the overhead of executing specific instructions provides another means of dynamic timing analysis.

These methods mainly suffer from two problems. First, computer architectures have become increasingly complex. A single instruction may have different execution characteristics depending on its interaction with other instructions. The main factors of unpredictability within architectures stem from pipelining and caching. Structural and data hazards create dependencies between instructions causing stall cycles. Memory references result in significant variations in latency depending on whether or not an item is cached. Hence, instruction frequencies are no longer adequate to determine the execution time of programs.

Second, it is difficult to determine the circumstances that cause a program to exhibit its worst-case execution time behavior. For programs with trivial inputs, a worst case input may be known for a given algorithm, *e.g.*, for simple sorting algorithms. But even if the worst-case input in terms of algorithmic complexity is known, there is no guarantee that this case actually corresponds to the worst-case temporal behavior due to its dependency on architectural aspects, as discussed before. Furthermore, programs with only moderately complex inputs may not have a known worst-case input or the task of determining this input may not be easy. As a result, dynamic timing analysis often utilizes sampling methods where only a subset of the input space can be probed to approximate the BCET and WCET. In general, these results will only be approximations since the actual BCET may be overestimated while the WCET may be underestimated. In particular, there has been no metric to determine how severe these differences are from the actual values.

Static timing analysis relies on analyzing a program without actually executing it. It does not require knowledge about the input specification. Instead, it analyzes paths through the control flow to determine the longest possible execution path. Hardware components are generally simulated to obtain cycle-level predictions of both BCET and WCET. These simulations take cache and pipelining effects into account but they may make sacrifices in terms of their precision in order to keep the simulation models efficient. In other words, simulation models result in safe predictions that do not overestimate the BCET and, conversely, do not underestimate the WCET. On one hand, they may not result in precise predictions due to data dependencies and other sources of lack of knowledge at the level of static program analysis. On the other hand, static approaches do provide a safe and efficient method to determine the extreme execution times. This results in a realistic framework for timing predictions whose results are directly applicable to schedulability analysis.

0.3 Goals and Results

This document describes research on timing analysis for uniprocessors and multiprocessors. Within uniprocessors, a modular framework for static timing analysis is developed that allows the separation of simulation aspects. In particular, a formal framework for cache analysis is proposed that can be formulated as a data-flow problem, which has been implemented. This method, called static cache simulation, allows memory references to be categorized according to their cache behavior. This cache information is then utilized by a separate component, which also models the pipeline execution and performs a path analysis. The resulting static timing predictions are shown to provide tight and safe estimates of the actual extreme execution times. In addition, static timing analysis and dynamic timing analysis are shown to complement each other since they bound the actual extreme execution times from opposite ends, thereby providing a degree of accuracy for both methods. For distributed systems, novel synchronization protocols are developed, proved correct and evaluation to allow the results of timing analysis of uniprocessors to be transfered to distributed real-time systems.

Part I discusses issues of timing analysis for uniprocessors, for which a set of tools has been developed. Chapter 1 introduces the results of schedulability analysis for uniprocessors and describes our tool for schedulability analysis of programming languages such as C and Ada. Chapter 2 gives an overview of methods for static timing analysis. In Chapter 3, modifications to a compiler for its integration into the tool set are discussed using the Gnu Ada Translator as an example. In Chapter 4, a framework for cache analysis is formally derived and implemented as another tool. Chapter 5 follows with the description of a tool for pipeline simulation and path analysis. Chapter 6 compares the approaches of static cache analysis and dynamic cache analysis. Chapter 7 outlines the design of a task analysis tool. Chapter 8 discusses future work and Chapter 9 summarizes the results of timing analysis for uniprocessors.

Part II focuses on the applicability of the results of timing analysis for uniprocessors in a distributed environment. Chapter 10 introduces several methods of timing analysis for distributed systems. Chapter 11 gives an overview of a distributed runtime system, called DSM-Threads, which is utilized for experiments in the next chapter. In Chapter 12, a novel protocol for distributed synchronization is derived, proved correct and evaluated. This protocol provides support for priorities in a distributed real-time system, which allows timing analysis to be applied in such an environment. Chapter 13 discusses future work while Chapter 14 summarizes the work on timing analysis for distributed systems. Finally, conclusions of both parts I and II are given.

Part I

Uniprocessor Systems

Chapter 1

Schedulability Analysis

This chapter is structured as follows: First, an overview of methods for schedulability analysis is given. Then, the fundamental design issues of our tool for schedulability analysis are described and the overall methodology and motivation for this work is given. Next, aspects of the implementation are explained followed by a discussion on the applicability of this work to Ada tasks. Finally, the work is contrasted with previous work and conclusions for this chapter are given.

1.1 Foundations

The purpose of schedulability analysis for real-time system is to determine statically if a given task set meets its temporal requirements. Many of today's results of schedulability analysis are based on a study by Liu and Layland for uniprocessors [39]. Liu and Layland based their analysis on a number of restrictions:

1. Tasks are *periodic*, *i.e.*, the task becomes ready (its task body is reactivated) after a relative *period* of time has elapsed. A task may be invoked (*i.e.*, it may start running) at or after its activation according to the scheduling policy (see below).

2. Each task has a *relative deadline* and an *absolute deadline*. An absolute deadline is the point in time where a certain activation of a task must have completed its execution. The relative deadline is the duration between the activation point and the absolute deadline. The relative deadline (simply referred to as *deadline* in the following) is assumed to be equal to the period in the original work by Liu and Layland [39].

3. All tasks initially start at time 0.

4. Each task has a known *worst-case execution time (WCET)* (a.k.a. maximum computation time).

5. Tasks are *independent*, *i.e.*, they do not share any resources or communicate.

Notice that restriction 2 constrains the analysis to *hard* real-time systems, *i.e.*, systems that do not tolerate any missed deadlines. In a hard real-time system, deadline misses may have catastrophic impacts on the environment or result in loss of life. In contrast, *soft* real-time systems may tolerate missed deadlines occasionally without causing such catastrophic impacts.

Rate-Monotone Analysis and Earliest-Deadline-First Analysis

With the above restrictions, utilization-based tests for schedulability analysis were developed, namely *rate-monotone analysis (RMA)* and *earliest deadline first (EDF)*. A task set is called *schedulable* if its overall utilization does not exceed a certain threshold. RMA uses a static priority assignment of tasks while EDF assigns priorities dynamically during scheduling. Systems have to support preemption and strict priority scheduling, *i.e.*, the scheduling policy requires that when a higher priority task becomes ready it preempts a lower priority task and starts running such that the running task is always a task with the highest priority.

Relaxing Restrictions

In later studies, several restrictions were relaxed to allow for

- deadlines to be smaller than periods,
- arbitrary start times and
- dependencies between tasks due to resource allocation.

Figure 1.1 depicts the parameters of a task.

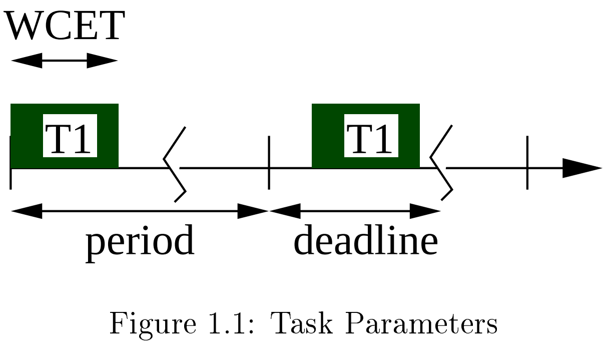


Figure 1.1: Task Parameters

Response Time Analysis

A second analysis approach was developed based on response-time analysis (RTA). Instead of checking system utilization, the (worst-case) response time is determined for each task under RTA. A task set is called *schedulable* if the response time does not exceed the deadline for each task. The scheme was originally developed for deadline-monotonic static priority assignments but has been extended to work with arbitrary static priority assignments. It is also subject to preemptive and strict priority scheduling. RTA has the advantage that it may determine if a system is schedulable where RMA supplies inconclusive results for the same task set. Furthermore, RTA allows inferences about the cause of failure on a per-task level where the cause of failure may not be determined as easily with RMA and EDF.

These sound theoretical foundations for schedulability theory are all based on the assumption that the WCET of a task be known *a priori* (statically). Over the last decade, considerable effort has been made to approach the problem of *timing analysis*, *i.e.*, to determine the WCET of a task. Part I of this document contributes one approach to timing analysis.

The remaining sections of this chapter outline the applicability of schedulability theory to Ada tasks and provide a description of the design and implementation of a schedulability analysis tool for Ada tasks. Schedulability analysis is the last step in the framework for timing analysis. Nonetheless, the schedulability analysis tool will be discussed in this chapter since it motivates the work presented in the consecutive chapters and gives more details about the theoretical results on schedulability analysis.

1.2 Overview of Our Tool for Schedulability Analysis

This work describes the design and implementation of a tool for schedulability analysis of Ada tasks. Based on the results of schedulability theory, a component-based design through the object-oriented features of Ada 95 provides portability, composability and extensibility together with simplicity for this tool. It is part of a larger timing analysis framework that works in conjunction with the Gnat compiler. This framework provides the means to make schedulability analysis an integral part of the software development cycle of real-time applications.

The language Ada was designed as a general-purpose language but it distinguishes itself from other general-purpose languages by its facilities for real-time and embedded systems. Ada provides support for real-time systems on the language level through tasks, synchronization, time facilities and the real-time annex [25]. Ada implementations often cater to real-time and embedded systems by providing cross compilation for embedded architectures and tools for program analysis of real-time applications. In the following, a system adhering to the real-time annex in terms of scheduling and locking policies as well as the Ravenscar Ada 95 subset [6] are assumed.

This work introduces our tool for real-time schedulability analysis. It calculates if a task set will meet its deadlines for a given scheduling paradigm. The tool is designed for extensibility in terms of new protocols as well as additions to existing protocols. Features of protocols can be combined to model a compound scheduling model by using object-like components as building blocks for such models. This is demonstrated by the addition of priority ceiling and inheritance protocols to arbitrary base protocols. The tool is implemented in Ada and supports rate-monotone analysis (RMA), response-time analysis (RTA) and earliest deadline first (EDF) analysis [39, 2]. It covers analysis with and without resource allocations, the latter optionally including protocols to avoid priority inversion. The tool is applicable to schedulability analysis of Ada tasks but is not necessarily limited to Ada. It is designed for integral compilation and schedulability analysis in the software development cycle of real-time systems.

The work is aimed at real-time scheduling in a uniprocessor environment where schedulability theory has supplied firm results that can be applied during the software development cycle of real-time applications. The target systems are hard real-time systems with firm deadlines or soft real-time systems with subsets of hard tasks, the latter possibly due to economic reasons rather than the risks of a system on its environment, as given in traditional hard real-time systems. The application area are systems with preemptive and strict priority scheduling and given task parameters, such as periods, worst-case execution times and resource requirements, just to name a few.

1.3 Design Goals and Methodology

The design of the tool for schedulability analysis was motivated by the following aims: simplicity, detailed analysis, portability, composability, extensibility and its ability to be integrated into a larger analysis environment. These issues were addressed as follows:

Simplicity: The result of the analysis either confirms the schedulability of a task set or rejects it, *i.e.*, the user receives a yes/no answer. In some cases, schedulability analysis may remain inconclusive but in this case other schedulability methods are suggested to yield conclusive results.

Detailed analysis: Optionally, the internal results of analyzing the task set are reported. This allows a user to asses how close to exceeding its schedulability bounds a task set is. It provides the means to report which task caused a task set not to be schedulable. The results may also be helpful for tuning purposes.

Portability: The analysis is performed on a set of tasks with given parameters that abstract from the implementation language, operating system and architecture.

Composability: Analysis methods are encapsulated into different components that can be combined to form a model for a scheduling framework with certain protocols, *e.g.*, for response-time analysis with resources and the priority inheritance protocol.

Extensibility: The interfaces of the analysis components serve as a contract for the addition of new protocols or methods for schedulability analysis. They can then be combined with existing methods and protocols.

Integration: The tool accepts the specification of the task set as a human readable input and also produces simple output, which is suitable for an integration into a larger analysis framework for automatically generated input and output potentially forwarded to a graphical user interface.

Schedulability analysis is based on the specification of a task set and a scheduling method. This specification contains two parts. The first part describes general settings while the second part specifies the parameters of each task.

General settings: This information applies to the entire task specification.

- The method for the schedulability analysis can be varied. Current methods include RMA, EDF and RTA.
- The time unit can be specified. This unit applies for all further numerical specifications of times.
- The resource protocol can be varied among priority ceiling and priority inheritance. Alternatively, no resource protocol can be selected, which implies that

priority inversion may occur and will be accounted for during schedulability analysis.

Task Parameters: For each task, a number of parameters are expected.

- The *task number* simply serves as a naming reference to relate the output of the schedulability analysis to a specific task out of a *set of tasks* with a cardinalidty of n.
- The *period* describes the rate of reactivation of the task.
- The *deadline* specifies the latest point in time that a task's execution must have completed. The deadline is always interpreted relative to the previous activation time.
- The *worst-case execution time (WCET)* represents the maximum duration that the task may need to execute. It must account for the duration from an invocation to the completion of a task along the longest possible execution path. The computation time may be specified as a singleton or as a list of m elements, where each element specifies the worst-case duration for a specific phase of the task's execution. In each phase, a task may hold different resources.
- The resource list is a list of m elements. Each element corresponds to a distinct phase and specifies a set of resources that the task holds during this phase, *i.e.*, it has the same length as the WCET list. The empty set indicates a phase without resources.

Based on this description of a task set and the analysis method, the tool performs the requested schedulability analysis, reports the results and, in case of a negative outcome, may suggest alternative analysis methods for the same task set.

The following symbols are used for schedulability analysis, divided into user-specified parameters and results calculated by the tool:

- user-specified parameters:

D_i - Deadline of task i;

T_i - Period of task i;

C_i - Worst-case execution (or computation) time of task i;

- calculated results:

  P_i - Priority of task i; (Higher values imply that a task is more important.)

  R_i - Response time of task i;

  B_i - Blocking time (or factor) of task i.

The priority P_i is listed under calculated results here. Notice that the priority may either be a user-specified parameter or a calculated results depending the the method for schedulability analysis, as seen in the following. Depending on the chosen method, schedulability analysis is based on the following theoretical results:

RMA: Rate-monotone analysis is based on a rate-monotone priority assignment, *i.e.*, shorter periods imply higher priority (higher urgency) thereby enforcing priority ordering of tasks. Schedulability of the task set with cardinality n is determined by checking on the system utilization calculated as the sum of each task's contribution to the utilization [39]:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1).$$

In addition, the critical zone is constructed, *i.e.*, a scheduling plan is given for the instance where all tasks are released at time 0. The critical zone reaches from time 0 to either the point in time where all tasks have completed execution at least once or where a task has missed its deadline. In the presence of resources, the blocking time is included in the schedulability test and the test is applied for each task separately [10]:

$$\forall i : 1 \leq i \leq n \rightarrow \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i(2^{\frac{1}{i}} - 1).$$

If the main inequality holds, the task set is schedulable. When system utilization exceeds 100%, the task set is not schedulable. However, if utilization exceeds the right-hand term of the inequation but is less than or equal to 100%, the test remains inconclusive and other analysis methods, such as RTA, need to be applied.

EDF: For earliest-deadline-first scheduling, the following schedulability test is performed [39]:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$

This test allows a maximum utilization of 100%. In the presence of resources, the blocking time is included in the schedulability test and the test is applied for each task separately [61]: For priority inheritance

$$\forall i : 1 \leq i \leq n \rightarrow \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq 1$$

and for priority ceiling

$$\sum_{i=1}^{n} \frac{C_i + B_i}{T_i} \leq 1.$$

RTA: Priorities are assigned in a deadline-monotone fashion. The schedulability test uses a fixed-point algorithm based on the following formulas [2]:

$$R_i^0 = C_i$$

$$R_i^{k+1} = C_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^k}{T_j} \rceil C_j.$$

If R_i^k equals R_i^{k+1}, then the fixed point is reached and the response time of task i has been determined. If the response time does not exceed the task's deadline, then the task is schedulable. In the presence of resource constraints the formula is augmented by a blocking factor [2, 10]:

$$R_i^{k+1} = C_i + B_i + \sum_{j=1}^{i-1} \lceil \frac{R_i^k}{T_j} \rceil C_j.$$

RTA provides conclusive results to determine if a task set if schedulable where RMA may be inconclusive or if priorities are not assigned in a rate-monotone fashion.

Blocking factors are optional and will be omitted in the absence of resources. Priority assignment occurs automatically for the above methods unless specified otherwise. In other words, a user does not have to be concerned about priorities as long as the traditional methods are adequate to describe a real-time application. Interrupts may be modeled by specifically assigning high priorities to the task that represents an interrupt handler. This priority assignment may violate the rate-monotone or deadline-monotone assignment. This does not pose any problems for RTA. For RMA, on the other hand, it is treated separately by an extension of the above formulas along the lines of the rate-monotone handbook [31].

Resources are taken into account by determining a blocking time for tasks covering delays incurred by lower priority tasks engaged in critical sections that cause a higher priority task to suspend until resources become available. Blocking times are based on the duration $D_{j,k}$ of "the longest critical section of task j among those guarded by semaphore S_k" [10]. These durations can be derived from the worst-case execution time of the critical sections protected by the same semaphore.

Requests for a common resource by different tasks may result in a situation termed *priority inversion*. Priority inversion describes the situation where a low priority task L holds a resource protected by semaphore $S1$ when a medium and a high priority task (M and H, respectively) become ready at $t1$ as depicted in Figure 1.2(a). The high priority

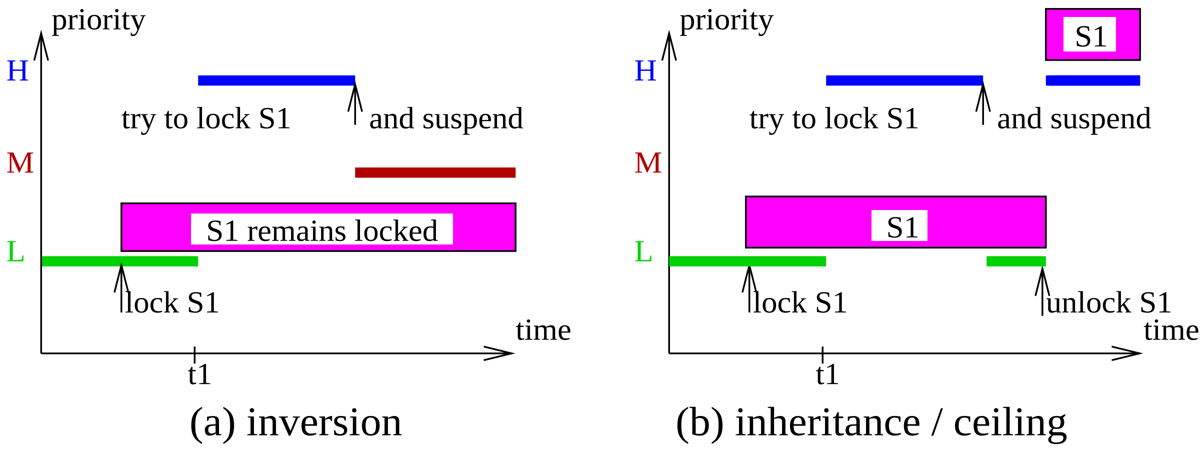


Figure 1.2: Resource Allocation under Priority Scheduling

task H then starts executing and tries to acquire the same resource $S1$, which blocks H. Then, M executes, thereby blocking H during M's entire execution. H has to wait for L to proceed and release $S1$ but L has to wait for M to finish. This situation of a lower priority task M interfering with a higher priority task H such that the execution of M does not contribute to H's progress is termed priority inversion. Notice that any number of medium priority tasks without resource requirements might become ready during the inversion period and would execute until finished while blocking L and thereby H. Hence, priority inversion may result in long blocking times.

A number of protocols have been proposed to counter this problem. The most common approaches are the priority-ceiling and priority-inheritance protocols. Both protocols distinguish a *base priority* statically assigned to a task and an *effective priority* that is dynamically changed according to protocol rules. Strict priority scheduling adheres to the effective priority, which is transparent to the application.

The priority inheritance protocol boosts the effective priority of a lower priority task holding a resource upon contention to the level of the highest priority requester. Upon resource release, the priority is lowered to its previous level. In Figure 1.2(b), H blocks when requesting the semaphore and inherits his priority to L. Thus, L continues execution up to the point where it leaves its critical section, releases the semaphore and lowers its effective priority to the base level. Then, H proceeds execution by obtaining the semaphore. Notice that M blocked neither H nor L during the contention period.

Another protocol, the priority ceiling protocol, also boosts the effective priority of a lower priority task holding a resource upon contention to the level of the highest priority requester. However, a resource request without contention is only granted under certain conditions. These conditions use the notion of a priority ceiling for each resource whose value equals to the highest base priority of any task accessing the resource. A resource request is granted only if no other task holds a resource or if the requester has higher effective priority than any other task holding resources or if he has the highest effective priority and holds the resources with the highest priority ceiling. The difference between inheritance and ceiling becomes apparent when multiple semaphores $S1$ and $S2$ are used as depicted in Figure 1.3. Inheritance causes priority boosting upon contention for the

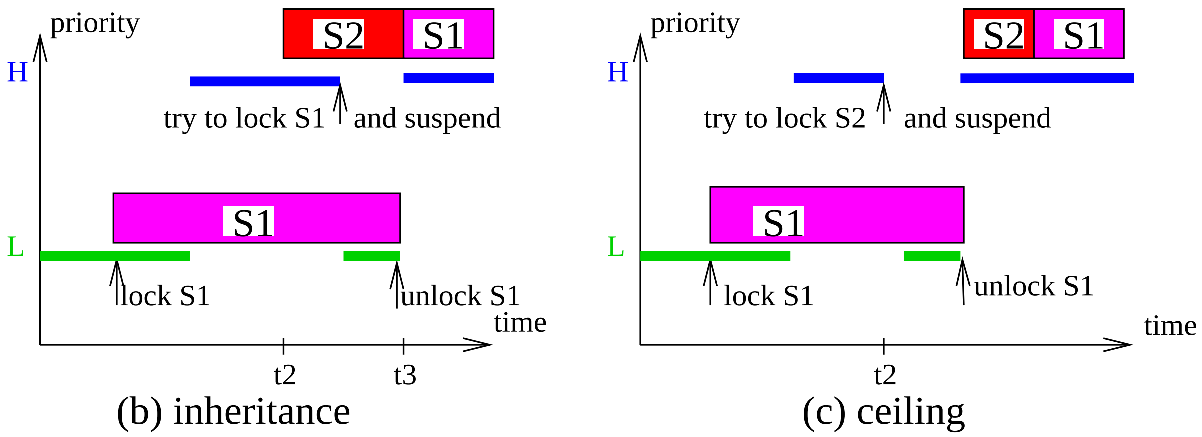


Figure 1.3: Priority Protocols for Resource Allocation

same resource protected by S1 at time $t3$. The ceiling protocol forces early boosting of L's priority at $t2$. The ceiling value of $S1$ and $S2$ is the same, determined as the priority level of H who uses both semaphores. Since L uses $S1$ at $t2$, the request for $S2$ by H cannot be granted and L's effective priority is boosted to H's level.

Let us assume there are m semaphores to protect critical sections that access shared resources. Each semaphore has a ceiling priority $C(S_k)$. The blocking times are calculated as follows (depending on the priority protocol) [10]:

Priority Ceiling Protocol: The ceiling $C(S_k)$ of a semaphore is the maximum of the priorities of tasks using this semaphore. Then, the blocking time of task i is given by:

$$B_i = \max_{j,k}[D_{j,k} : T_j < T_i \wedge C(S_k) \geq P_i].$$

Priority Inheritance Protocol: The ceiling $C(S_k)$ of a semaphore is the maximum of the priorities of tasks using this semaphore or of any outer ceiling $C(S_l)$ for a semaphore S_l that is requested before S_k is requested (S_k nested within S_l), which ever is larger. Then, the blocking time of task i is given by:

$$B_i = min(B_i^l, B_i^s)$$

where

$$B_i^l = \sum_{j=i+1}^{n} \max_{k}[D_{j,k} : C(S_k) \geq P_i]$$

and

$$B_i^s = \sum_{k=1}^{m} \max_{j>1}[D_{j,k} : C(S_k) \geq P_i].$$

The term B_i^l determines the blocking for *all tasks* with lower priority using common semaphores. The term B_i^s represents the blocking for *all semaphores* that are being used by lower priority tasks. Hence, the former may overestimate the actual blocking by taking blocking on the same semaphore by multiple tasks into account while the latter may result in overestimation due to blocking by the same tasks on distinct and non-nested semaphores. The minimum of these two values, therefore, represents a safe estimate for the blocking factor of a task i. For nested critical sections, we require that the durations $D_{j,k}$ of outer semaphore include the durations due to inner semaphores in a transitive fashion. An example will be given later on.

No Protocol: In the absence of such a protocol, priority inversion may cause a lower priority task to block a higher one for the duration of its entire execution. These long delays are prone to cause schedulability problems where one of the above protocols still allows a task set to be schedulable. Nonetheless, in the absence of the above protocols within a scheduling framework, priority inversion can be included in the calculation of the blocking times. This allows schedulability analysis to be performed as usual [31].

1.4 Implementation

The methods for schedulability analysis are applied to a task set by determining the priority of a task and calculating the utilization or response time, depending on the analysis method. The priorities are based on rate-monotone assignments for RMA and arbitrary priority assignments for RTA, *e.g.*, deadline-monotone assignments, *i.e.* tasks with smaller values statically receive higher urgency (priority). For EDF, the task closest to its deadline during runtime (dynamically) receives the highest priority. The calculations for schedulability may involve fixed-point algorithms that either iterate until they reach a stable state or they determine that a deadline is violated and terminate prematurely. The blocking times are also calculated by a fixed-point algorithm and are subject to the same constraints. In addition, an event-based simulation of a scheduling scenario can be constructed for a task set and a set of release times. This discrete event modeling provides an efficient way to simulate scheduling actions, which are then recorded for later replay by the user. Figure 1.4 depicts the Ada code for the calculation of critical zones. The critical zone is constructed in a loop as follows. At the beginning of each iteration, the current task is either invoked or continued. Then, the highest priority task task at the next scheduling point is identified through `Get_Next`. If this next task differs from the current task, then a preemption point is recorded, *i.e.*, the current task is suspended and the next task is invoked. Newly activated tasks for the next scheduling point are then added to the list of ready tasks. Also, all tasks are checked to see if any of them missed their deadline. If either a deadline has been missed or the critical zone is complete, the loop terminates. Otherwise, it reiterates with the next task as its current task. For a rate-monotone priority assignment, the critical zone is complete when the task with the longest period (implying the lowest priority) has completed execution. If a deadline is missed before this point, then the task set is said to not be schedulable for this given scheduler.

```
begin - -> procedure Compute_Crit_Time_Zone.
   loop
      Switch(Current_Pos).Whenn := Time_Ptr;
      Switch(Current_Pos).Which := Current_Task;
      if Task_Data(Current).Suspended and
            Task_Data(Current).To_Run /=
            Task_Data(Current).Wcet_Sum then
         Switch(Current_Pos).What := Continued; - -> task not completed
      else
         Switch(Current_Pos).What := Started;
      end if;
      Current_Pos := Current_Pos + 1;
      Next := Get_Next; - -> get next task to be running.
      if Exists_Suspending_Task then
         - -> other task which preempts CURRENT?
         Passed_Time := Task_Data(Next).Start - Time_Ptr;
         Task_Data(Current).To_Run := Task_Data(Current).To_Run -
                                      Passed_Time;
         Task_Data(Current).Suspended := True;
         Time_Ptr := Time_Ptr + Passed_Time;
         Switch(Current_Pos).What := Suspended;
         Exists_Suspending_Task := False;
      else
         Time_Ptr := Time_Ptr + Task_Data(Current).To_Run;
         Task_Data(Current).To_Run := Task_Data(Current).Wcet_Sum;
         Task_Data(Current).Suspended := False;
         Task_Data(Current).Times := Task_Data(Current).Times + 1.0;
         Task_Data(Current).Start := Task_Data(Current).Times *
                                     Task_Data(Current).Period;
         Switch(Current_Pos).What := Finished;
      end if;
      Switch(Current_Pos).Whenn := Time_Ptr;
      Switch(Current_Pos).Which := Current;
      Current_Pos := Current_Pos + 1;

      Correct_Starts;
      Check_Deadline_Miss(Switch, Current_Pos);

      - -> exit when critical zone completed:
      exit when Current = N and not Task_Data(N).Suspended;
      Current := Next;
   end loop;
end Compute_Crit_Time_Zone;
```

Figure 1.4: Calculating the Critical Zone

The task set is supplied as the standard input for the analysis tool while results are reported as standard output. Figure 1.5 depicts an example for an RMA task set without resource as an input, where task 1 has a period of 3 and WCET of 1 and task 2 has a period of 4 and WCET of 2. RMA remains inconclusive for this task set. The critical zone consists of a single execution of each task in this case without a missed deadline. A sequence of executions may be generated from this output by a graphical user interface, such as depicted in Figure 1.6. It turns out that this task set is schedulable under RTA.

Input	Output
# data file for the anatool.	Analysis method: RMA
info output mode: full	Unit: ms
method: RMA	task number, now: 1; in input: 1
unit: ms	period: 3.000
	computation times: 1.000
task: 1	sum of computation times: 1.000
period: 3	priority: 2
computation time: 1	
	task number, now: 2; in input: 2
task: 2	period: 4.000
period: 4	computation times: 2.000
computation time: 2	sum of computation times: 2.000
	priority: 1
	at 0.000 task 1 STARTED
	at 1.000 task 1 FINISHED
	at 1.000 task 2 STARTED
	at 3.000 task 2 FINISHED
	Critical time zone ranges from 0.000 to 3.000.
	Utilization of 0.833 exceeds the bound of 0.828.
	All time data in ms.
	This set of tasks MAY NOT BE schedulable.
	RMA remains inconclusive, try RTA.

Figure 1.5: Sample Execution for RMA

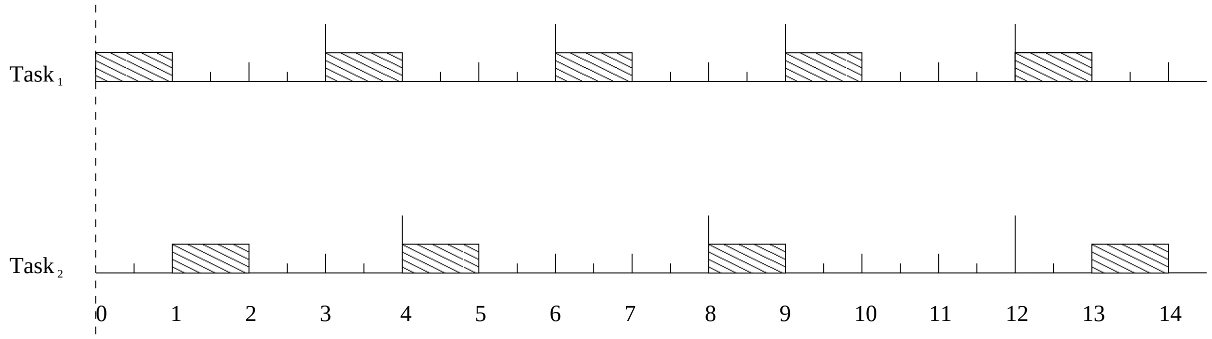


Figure 1.6: Sample Schedule for RMA with Critical Time Zone

Figure 1.7 depicts the schedulability information for a task set with resources under RTA, where priorities were assigned in a deadline-monotone fashion. The table represents the longest critical sections for pairs of tasks and resources, *i.e.*, the components $D_{j,k}$ introduced in the context of the priority ceiling and inheritance protocols, and is used to determine the blocking times (called blocking factors in the Figure). For example, the blocking time B_i^l is determined from the maximum value of each row for lower priority tasks. B_i^s is the maximum from each column of a semaphore that task i uses. The blocking times (factors) and task parameters are then reported for each task (only depicted in part in the Figure). This task set is schedulable since the response time of a task does not exceed its deadline. The verbose information is optional and can be suppressed by omitting the full mode as an input/output mode.

1.5 Schedulability Analysis for Ada

The schedulability analysis is based on the specification of parameters for tasks. These tasks have to represent sequences of code without task suspension other than suspensions caused by preemption or due to resource requirements. Hence, Ada tasks may not always correspond to scheduling tasks (simply called tasks in the following) but they may often consist of a sequence of phases. Ada tasks may be subject to suspension due to synchronization, delays or due to I/O requirements. Thus, the code of an Ada task has to be partitioned into a sequence of phases with consecutive start times and deadlines. Voluntary delays and I/O delays may be accounted for by offsets of the start time relative to the previous deadline in a sequence of phases. Ada tasks must be statically created rather than dynamically and execute an infinite loop whose body represents the sequence

Input	Output
# data file for the anatool. method: RTA	Analysis method: RTA
unit: ms	Unit: ms
resource access protocol: inheritance	Resource access protocol: INHERITANCE
info output mode: full	
	R1[4] — R2[4] — R3[3] —
task: 1	————————————————-
period: 40	1— 1.000 2.000 0.000
deadline: 40	2— 0.000 9.000 3.000
computation time: 1.0, 1.0, 2.0, 2.0	3— 8.000 7.000 0.000
resource: {0}, {1}, {2}, {0}	4— 6.000 5.000 4.000
task: 2	blocking factor task 1: 17.000
period: 80	blocking factor task 2: 14.000
deadline: 80	blocking factor task 3: 6.000
computation time: 1.0, 9.0, 3.0, 3.0	blocking factor task 4: 0.000
resource: {0}, {2}, {3}, {0}	
	task number: 1
task: 3	period: 40.000
period: 100	deadline: 40.000
deadline: 100	computation times: 1.000, 1.000, 2.000, 2.000
computation time: 1, 8, 7, 3	sum of computation times: 6.000 to 3.000.
resource: {0}, {1}, {2}, {0}	resources: {0}, {1}, {2}, {0}
	priority: 4
task: 4	[...]
period: 120	response time task 1: 23.000
deadline: 120	response time task 2: 36.000
computation time: 4, 6, 5, 4, 2	response time task 3: 53.000
resource: {0}, {1}, {2}, {3}, {0}	response time task 4: 68.000
	All time data in ms.
	This set of tasks IS schedulable with RTA.

Figure 1.7: Sample Execution for RTA

of phases. The worst-case execution time for each phase has to be specified as part of the input for the schedulability analyzer.

Synchronization requires further work. Ada 95 protected objects (excluding protected entries) can simply be treated as resources within the existing scheduling frameworks. They represent the common case of asynchronous communication already accounted for in the models. Such restrictions on tasking and synchronization are consistent with the requirements imposed by the Ravenscar subset of Ada 95 [6]. The Ravenscar profile also allows restricted use of protected entries for simple task synchronization, which may be represented by partial ordering of task phases as discussed in the next section.

Rendezvous, on the other hand, represent synchronous communication. Either the caller or the receiver (acceptor) may be stalled on a rendezvous, depending on the dynamic schedule. This scenario cannot simply be modeled as resources. A restricted solution was suggested by handling rendezvous as synchronous sessions [7]. This model requires that different rendezvous may only occur in a certain order that allows each rendezvous in sequence to be assigned priorities that are strictly monotonically decreasing.

We are currently working on integrating the schedulability analysis into our existing timing analysis framework. We implemented a set of tools to derive the WCET of arbitrary code sections from the source program, as depicted in Figure 1.8.

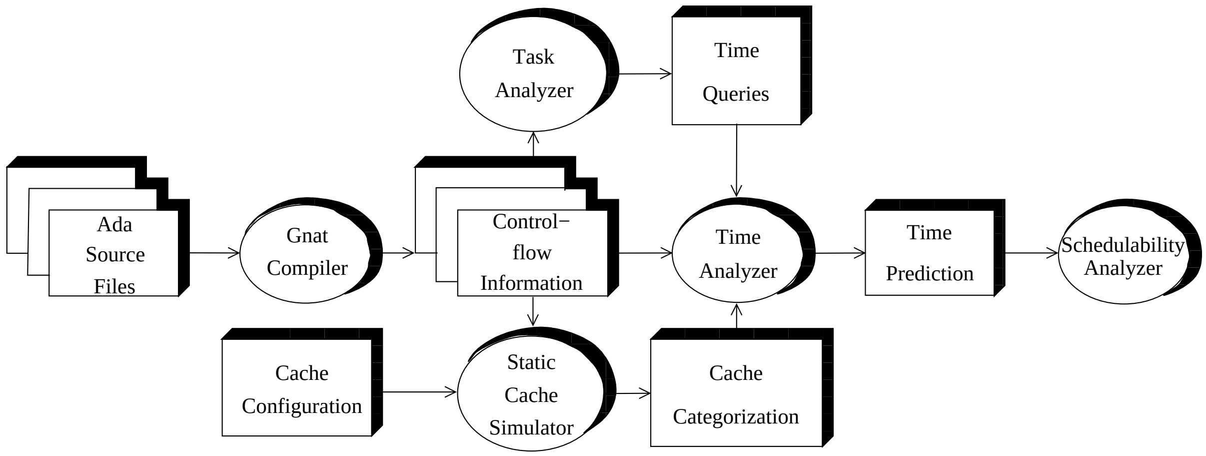


Figure 1.8: Toolset for Timing Analysis

The information required for WCET prediction is extracted by a modified version of the Gnat compiler [67] and then fed to a cache simulator [52, 69] and a timing analyzer performing pipeline simulation and path analysis [19]. The missing link is a task analyzer whose objective is to determine the phases for each Ada task. Phases are separated by synchronization points, which are well-defined in the Ada 95 language [25]. The timing analysis tool is then requested to report the WCET of each phase. This information, together with the specification of the task set, provides the input to the schedulability analysis. Overall, the framework allows software development of real-time systems with an integrated timing analysis. Compilation of software components may be accompanied by an automatic check to determine if timing requirements can still be met. If timing violations are detected, proper steps can be taken during software development, ranging from reverting changes that caused deadline overruns over activation of different scheduling methods to a partial redesign of the culprit task or, in the worst case, even a complete redesign based on insight knowledge about the reasons that caused problems, to resolve them.

In summary, current work includes the task analyzer and extensions to the schedulability tool to handle resources for EDF and the validation of methods for non-monotone priority assignments (*e.g*, due to interrupts) for the analysis methods.

1.6 Related Work

Stoyenko describes a schedulability analyzer [62] specifically targeted for the Real-Time Euclid environment but not applicable to more general schedulability problems. Caisarts [22] is an extensible, rule-based tool for schedulability analysis. It supports a window-based user interface and a large number of task parameters. Its strength is to give advice in situations where schedulability problems arise. Scheduler 1-2-3 [66] allows schedulability analysis for RMA. Perts [41] includes schedulability analysis for RMA and RTA with resources and is part of a larger framework for software development with a set of scheduling algorithms whose code can be selected from a library. Stress [1] provides schedulability analysis for a number of predefined frameworks including RMA and RTA but also allows user-defined scheduler support through a programming-language based specification, which is also used for simulations of new protocols.

Our approach differs from the previous work by its component-based approach that uses the object-oriented programming paradigm. In this sense, it is probably closest to the goals of Perts and Stress. The main difference from prior work is that our work aims at an integration into a larger toolset for software development of real-time systems in the context of the Gnat compiler, which may be unprecedented. Furthermore, our work builds on the results of the rate-monotone handbook [31], Buttazzo's compilation of schedulability results [10] and the approach of deadline scheduling [61]. It goes beyond the previous work in several ways. We account for priority inversion in the absence of a priority protocol, we support nested resources and we allow ad-hoc priority assignments to tasks to model interrupts.

1.7 Conclusion

The chapter described the design and implementation of a tool for schedulability analysis of Ada tasks. The component-based design through the object-oriented features of Ada 95 provides portability, composability and extensibility together with simplicity for this tool. It is based on the results of schedulability theory, including RMA, EDF and RTA with and without resource constraints, optionally with priority ceiling and inheritance protocols. Priorities are automatically assigned to tasks according to the analysis method. In addition, the framework supports user-assigned priorities during the analysis that go beyond the automatic assignments. The schedulability tool is part of a larger timing analysis framework that works in conjunction with the Gnat compiler. This framework provides the means to make schedulability analysis an integral part of the software development cycle of real-time applications.

Chapter 2

Static Timing Analysis

The objective of timing analysis in general is to determine the extreme execution times of code portions, *e.g.*, for the code of a task. This comprises both the best-case execution time (BCET) and the worst-case execution time (WCET). Chapter 1 motivated the need to determine WCETs to perform schedulability analysis. The BCET may be needed to ensure minimal temporal requirements. For example, sensory devices may enforce certain minimal sampling rates, *i.e.*, due to hardware requirements a certain minimum duration may have to elapse between two consecutive sensor readings. – Otherwise sensory readings may be invalid, which may cause control actions on actuators that were not intended for false sensor values but rather for reacting to exceptional and potentially dangerous situations.

Timing analysis for uniprocessor systems has been subject to a number of assumptions to make the problem tractable. First, tasks have to adhere to the restrictions imposed by schedulability analysis as explained in Chapter 1. This means that tasks should be periodic with deadlines less than their periods, they should have a proper priority assignment and use resources under priority ceiling or inheritance.

Second, static timing analysis requires that the timing characteristics of a task be analyzable but not necessarily executable. The code of a task has to be available either as source code, as an intermediate representation or as object code, depending on the level of analysis. Since the halting problem is known to be NP complete [15], loops within a task should have an upper and a lower bound on the number of iterations. Heap allocation, recursive calls and direct calls through pointers are not allowed. Heap allocation may cause unpredictable delays due to its interaction with the virtual memory layer. Recursion may cause a number of invocations with an unknown static bound. Indirect calls prevent interprocedural analysis since the callee is unknown at static time. Some of these restrictions are imposed to simplify the model for timing analysis. Certain restrictions may be lifted to some extend at a later point. For example, recursion is permissible if the depth of recursive calls is bounded, optionally specified by a pragma. Furthermore, heap allocation from a fixed size pre-allocated memory area may be predictable.

Overall, these assumptions serve the purpose of ensuring the predictability of a task's execution in terms of its temporal behavior. The assumptions have been reflected in the definition of language subsets suitable for real-time systems, such as the Ravenscar profile for the Ada language [6].

2.1 Static Analysis Methods

Static timing analysis is a method to bound extreme execution times (BCET and WCET) through program analysis, *i.e.*, without actually executing any application code.

Several approaches exist for static timing analysis that can be distinguished by their analysis technique and the level of analysis. Analysis techniques address *how* the objectives of timing analysis can be met. Static timing analysis comprises a number of tasks, ranging from cache simulation over pipeline simulation to path analysis, just to name the most significant issues involved. The two distinct analysis techniques used today are *integrated analysis* and *separate analysis*. The former uses one framework to collectively simulate the above aspects while the latter approach separates the simulation of the aspects in distinct components. Integrated simulation has the advantage that correlations between simulated aspects can be incorporated more easily but often result in higher complexity and lower performance. Separating simulation components simplifies the task of simulation but has to address the problem of a potential loss of accuracy of the overall simulation since interactions between components may not always be possible.

The level of analysis determines which kind of representation is utilized for simulation purposes. At the level of source code, timings may be related to programming constructs, which may be helpful during program development but it becomes difficult to include aspects of code optimization and the architecture. Intermediate representations provide more details for timing analysis at a cycle level while still abstracting from a specific architectural representation. Analyzing object code has the advantage that library code can be included in the absence of the corresponding sources but may become more dependent on a specific architectural representation.

Today, mainly five fundamental models for static timing analysis exist (see Table 2.1). First, a source-level oriented timing schema propagates the WCET/BCET through a tree of timing nodes and handles pipelined RISC processors with first-level split caches [55, 38, 29]. Second, a hardware simulator can be extended to handle known and unknown values in order to simulate an execution with unknown inputs on the cycle level for a given executable [42]. Third, a constraint-based method models architectural aspects,

analysis technique	integrated			separate	
level of analysis	timing schema	hardware simulator	constraint based		data-flow based
source code oriented	[55, 38, 29]		[35, 36]		
intermediate code					[19, 21, 69, 52]
object code		[42]		[14, 65]	

Table 2.1: Classification of Static Timing Analysis Work

including caches, *via* integer linear programming [35, 36]. Forth, the approach described in this document uses data-flow analysis to model the cache behavior. Pipeline simulation is handled later within a timing analyzer that also performs path analysis [19, 21, 69, 52]. The first, second and third approaches use integrated analysis of caches while our approach and another approach discussed below use separate analysis. Separate analysis allows us to deal with multi-level memory hierarchies or unified caches [47]. Another approach to separate analysis that also uses data-flow analysis was inspired by our work but diverged over time and now employs a different data-flow model [14]. This approach has been combined with integer linear programming to model architectural characteristics [65].

Overall, separate analysis has proved to be much more efficient. Separate analysis can be orders of a magnitude faster in terms of simulation overhead when compared to the integrated approach. The results also indicate that the level of accuracy is similar for both separate and integrated analysis methods.

In the following, an overview of the components for our approach is given. The next chapters continue by describing these components.

2.2 Analysis Components

Our approach to static timing analysis can be characterized as an intermediate code-based analysis that corresponds to the actual instructions of the architecture but abstracts from the instruction representation for the sake of portability. Thus, code portions of a real-time task (or of a task's subset of code) are analyzed for a given architecture (currently the SPARC), optionally with instruction and data caches with parameters such as cache size, line size and associativity. The analysis uses closed formulae to derive cycle-level estimates of the WCET and BCET.

Our framework of timing analysis uses a set of tools as discussed in Chapter 1 and depicted in Figure 1.8. An optimizing compiler has been modified to emit control-flow information and information about data references in addition to regular target code generation. From the control-flow information, the calling structure of functions can be inferred. Until recently, the research compiler VPCC/VPO [3] performed this task. We have integrated Gnat/Gcc [60, 67] into this environment, as described in the following chapters.

A static cache simulator uses the control-flow information and calling structure in conjunction with the cache configuration to produce instruction and data categorizations, which describe the caching behavior of each instruction and data reference. In the past, separate analyzers for instruction and data caches were used since data references require separate preprocessing *via* an address calculator. The next chapter also describes the efforts of building a single analyzer for unified caches and the handling of secondary caches. The timing analyzer includes a simulator for pipeline execution. It overlays the pipeline behavior and the effect of cache categorizations while performing a path analysis over the control-flow information of the program. The timing analyzer yields predictions of the BCET and WCET for portions of the program or the entire program, depending on the request. Requests can be issued by a user, or, as depicted in the figure, by a task analysis tool that determines synchronization points within the application code. These synchronization points allow the specification of task phases according to the requirements for schedulability analysis. Hence, BCET and WCET predictions of task phases can be forwarded to a schedulability analysis tool, which determines if the overall schedule is feasible (see Chapter 1).

Chapter 3

Compiler Integration

This chapter details the efforts to integrate a commonly available and widely accepted compiler, the Gnat/Gcc family, into our tool set for timing analysis. We discuss the design and implementation of our modifications to Gnat/Gcc, where we accommodated our enhancements in a widely portable way within the common back-end of Gnat and Gcc. We compare our experience with prior modifications of a research compiler and we generalize our findings by stating common requirements for extensions of other compilers.

Notice that Gnat and Gcc share a common back-end that is suitable to accommodate the required modification, enabling its use for Ada real-time applications and embedded C programs at the same time. It is our intend to demonstrate with such a prototype environment that static timing analysis should become an integral part of the software development cycle for hard real-time applications and portions of soft real-time applications requiring high reliability.

3.1 Prerequisites for a Compiler

Modifications to a compiler can be performed at various levels. Early work on static timing analysis defined a timing schema based on source-level language constructs [55, 38]. Such a modification to the front-end of a compiler may seem more portable but the effects of optimizations and code generation cannot be taken into account. This approach generally assumed the absence of global optimizations (across basic blocks) and loop optimizations. Thus, its applicability for realistic code is severely limited.

Our approach has always been to modify the back-end of an optimizing compiler after all optimizations have been performed. This approach provides the means to analyze optimized code. At a first glance, it may seem to limit portability. However, when the intermediate representation within the back-end is general enough to represent arbitrary architectures, modifications to the back-end become portable. Thus, a *first requirement* for modifying a compiler is the accessibility of the intermediate representation after optimizations have been performed. The verification of timing constraints of actual applications are also facilitated when the analysis is performed on the intermediate representation before machine-dependent assembler code is emitted. For the VPCC/VPO compiler as well as the Gnat/Gcc compiler, we performed most of the analysis on RTLs (referred to as register transfer lists or register transfer language).

For creating input for our simulation tools, information about the control flow, instruction types, operands, the number of instructions, loops, loop frequencies and function calls are required. Assumptions are made about the source programs that are common for timing analysis. In particular, it is assumed that a deterministic call graph (absence of indirect calls) and the absolute upper and lower loop bounds are given (to ensure termination). If the bounds of loops cannot be derived analytically, the user will be prompted during timing analysis to supply this information. The control-flow structure, loop nesting, etc. have to be valid at the time that the information is extracted, *i.e.,* during or before emitting assembler code. The code optimizations either have to be performed before our flow analysis or the compiler has to keep the flow information up to date. Thus, a *second requirement* is the ability to generate flow information (or access such information if already calculated) from the intermediate representation, thereby enhancing portability. In fact, in many cases the compiler will already provide abstract data structures and algorithms to calculate this information since it is generally needed to perform optimizations. This insight about calculated information also illustrates the benefits of generating the flow information in a compiler back-end *vs.* analyzing executable code and having to re-generate the flow information that used to be available when the executable was generated.

Our modifications of the Gnat/Gcc compiler add a new program module and extend the back-end of the compiler with calls to functions that are defined in this new program module. The functions operate with global variables defined in this module, which constitute logical channels between the compiler back-end and the new module. Every function has a well-defined call site within the compiler back-end. The variables are split into two sets, registers and switches. The design will be detailed in the following section.

3.2 Modifications of Gnat/Gcc

We modified the back-end of the Gnat/Gcc compiler to emit instruction information for Ada and C programs. The compiler provides the input for the static cache simulator, which returns results of the cache simulation, and the pipeline simulator, which uses the results of the cache simulation and returns timing predictions. In our approach, most modifications are performed on the RTL-level in order to provide high portability. The overall approach can be described as a register-switch system where switches provide information about analyzed program constructs and registers supply concrete information about relevant program entities. Switches are booleans and registers are integers and stacks. For example, if the switch *it_was_conditional_jump* evaluates to true, then we know that the analysis has just passed a conditional jump encoded as an RTL. If the top of the stack *count_loop_stack* contains the value 3, then this indicates that the current program constructs are on the loop nesting level of depth 3. The register *count_instructions* counts the machine instructions. It would be possible to define the operational semantics of the register-switch system through transitions on the states whose components are sets of switches, sets of registers and the program represented through RTLs.

The entire program analysis is performed before assembler code emission. All operations are defined to cooperate with peephole optimization so that the correctness of our approach is guaranteed for all optimizations, even for machine-dependent ones.

The modifications of the compiler could be split into two sets:

1. collecting data-flow information and

2. counting machine instructions.

The entire data-flow analysis is performed on the RTL-level. It provides information about jumps, loops and function calls and creates basic block nodes that are relevant for our cache and pipeline simulation models. Every basic block contains information about its number of instructions, first and last RTLs, etc. Figure 3.1 shows the structure of basic blocks generated by our data-flow analysis as a sequence of references to RTLs within the global stream of RTLs. Unfortunately, Gnat/Gcc does not incrementally update such information but rather calculates it on demand from the RTLs when needed.

The counting of machine instructions is performed by inspecting instruction templates during assembler code emission. For the Gnat/Gcc compiler, it is impossible to perform this operation on the RTL-level because machine-dependent components of the compiler contain templates that transform one RTL into multiple machine instructions.

Functions for operations on switches, registers and functions for computing information that we need for the data-flow analysis are implemented in an additional program module, namely the timing module. Figure 3.2 depicts the infrastructure of the modified compiler back-end where only switches and registers connect the new timing module to the original compiler back-end.

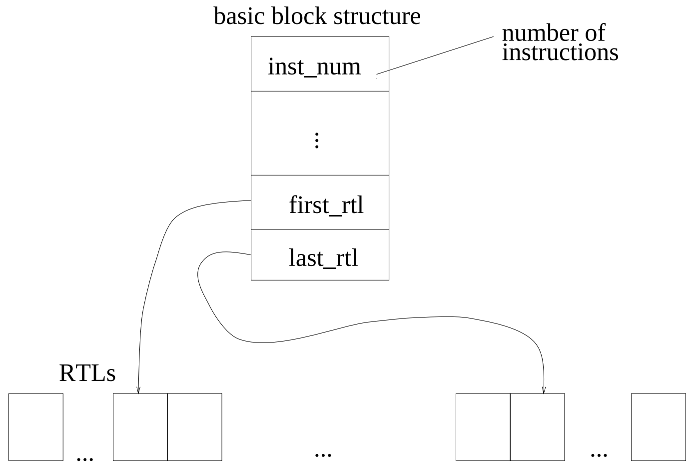


Figure 3.1: Structure of basic blocks

This modular approach simplifies the development and, at the same time, facilitates the integration of our modifications into new versions of Gnat/Gcc compiler releases. Besides the additional timing modules, we added 250 lines to the existing 300,000 lines in three out of 120 back-end modules, *i.e.*, our changes are minimal (less than 0.1% of the code) and localized in terms of the module structure. We intend to support future compiler upgrades by simply merging calls to our new module into the components of the compiler back-end. Even when the Gnat/Gcc back-end is modified, we can still insert the calls at their proper places unless major structural changes of the back-end occur, *i.e.*, by using context-sensitive merging through the tool `patch`. This allows us to support future Gnat/Gcc releases, even though our enhancements for timing analysis are not an integral part of Gnat/Gcc.

3.3 Comparison with Another Compiler

We integrated the VPCC/VPO research compiler into our previous tool set. The compiler back-end VPO was modified to emit instruction information for the simulation process. Like Gnat/Gcc, VPO also uses the concept of RTLs but utilizes a different internal representation. Furthermore, the control-flow information is incrementally updated during optimizations. For the VPO compiler, the step of translating RTLs into concrete assembler code is *pure* in the sense that there exists a one-to-one correspondence between RTLs and machine instructions. However, the abstract representation of RTLs is machine-independent, which keeps the analysis portable. In other words, there is no code that generates assembler code templates with more than one machine instruction.

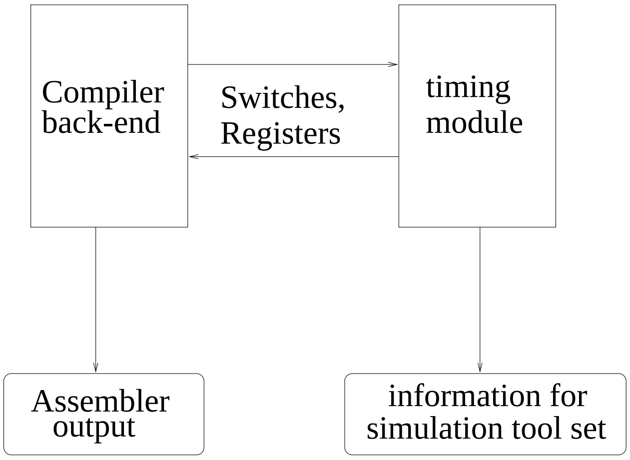


Figure 3.2: Infrastructure of the extended compiler back-end

For VPO, the entire analysis was realized on the RTL-level, *i.e.*, the modifications for emitting basic block information for the static cache simulator were fully portable without problems. The loop structure and data-flow information was readily available, which greatly facilitated the generation of the required input for our tool set. For Gnat/Gcc, we even found instances of assembler instructions that were specified within the machine description and emitted assembly straight into the output stream without using the proper output interfaces. For such instances, we needed to insert calls to our module within the machine description. Only the inconsistent use of standard interfaces within the Gnat/Gcc compiler prevented us from using portable methods within all places.

3.4 Implications for Other Compilers

We encountered a number of previously stated problems when we integrated Gnat/Gcc into our timing analysis environment. Our experience shows that other compilers can be integrated more easily with our timing analysis environment when there exists a one-to-one correspondence between the intermediate representation and machine instructions, *i.e.,* the translation of the intermediate representation is pure in the previously stated sense. Many compilers use different intermediate representations depending on the kind of optimizations that are currently being performed. For such cases, only some information will be incrementally updated throughout the entire optimization process. If the information is insufficient to drive the input for timing analysis, additional analysis has to be performed within the back-end. We encountered this problem with Gnat/Gcc for the control flow, the loop structure and data-flow information while VPO already supplied the proper information in its data structures. Our infrastructure of inserting calls into the existing back-end facilitates such post-optimization analysis. The analysis is thereby isolated within an additional module and driven by the register-switch model.

Our efforts concentrated on RISC machines as target architectures, in specific the SPARC architecture. Handling CISC machines requires minor extensions of the interfaces for the simulation tools to represent a variable instruction format that can be accommodated within our current design. Likewise, current work includes extensions for the Intel Pentium architecture. Our findings so far indicate that most of our work will be portable across architectures within Gnat/Gcc.

3.5 Conclusion

We described our work of integrating a commonly available and widely accepted compiler, the Gnat/Gcc family, into our tool set for timing analysis. We discussed the design and implementation of our modifications to Gnat/Gcc. Our enhancements were integrated by inserting calls to our analysis module from machine-independent modules within the common back-end of Gnat and Gcc. We performed the analysis immediately before emitting assembler code to cover the effect of peephole optimization, which is the last optimization within Gnat/Gcc. Due to the inconsistent handling of machine dependencies within Gnat/Gcc, a small number of modifications had to be made within machine-dependent code. Furthermore, we had to recalculate control-flow and data-flow information since they are calculated on demand and only within certain optimizations. Our experience with other compilers shows that this integration work is facilitated when flow information is incrementally updated during optimizations and if there is a one-to-one correspondence between the intermediate representation and machine code. This work is one step on the way to make the analytical approach of static timing analysis available to the real-time community. Using these tools, the resulting timing predictions can be used in schedulability analysis to verify the overall temporal behavior of tasks.

Chapter 4

Static Cache Simulation

Caches were initially perceived to introduce non-predictive temporal behavior to real-time systems [59, 18]. First, preemptive scheduling may impose context switches at arbitrary points during program execution. When a context switch occurs, the cache content of one task may be invalidated by the execution of another task. Second, even in non-preemptive scheduling, caches impose non-trivial problems due to varying latencies for references within the memory hierarchy. An instruction's execution time can vary greatly depending on whether the instruction causes a cache hit or miss. Whether or not a particular reference is in cache depends on the program's previous dynamic behavior.

The first problem, *i.e.*, unpredictable context switching for preemptive scheduling, can be countered by hardware and software solution for cache partitioning [30, 70, 44, 37]. However, these approaches find little use in practice due to a loss of cache performance and the need for special-purpose hardware when caches are partitioned. Recently, attempts have been made to incorporate cache invalidation on preemption into rate-monotone analysis and response-time analysis, which allows WCET predictions for non-preemptive systems to be used in the analysis of preemptively scheduled systems [8, 9, 33]. These efforts then reduce cache prediction to the second problem, *i.e.*, predictability of cache behavior without taking preemption into account. It is the second problem that this work focuses on.

The prediction of the temporal behavior of memory references in the presence of caches is accomplished through a static analysis framework that bounds the cache performance for real-time applications. The information resulting from static timing analysis can be utilized within the extended framework for schedulability analysis described in Figure 1.8. Thus, the results of this work for static cache analysis are applicable to both preemptive and non-preemptive systems. However, for preemptive systems the overhead of operating system calls would need to be reflected in the model to account for delays due to resource allocation and synchronization between tasks.

The following section summaries the approach and the results of static instruction cache simulation as described in [52], which constitutes an integral part of this work.

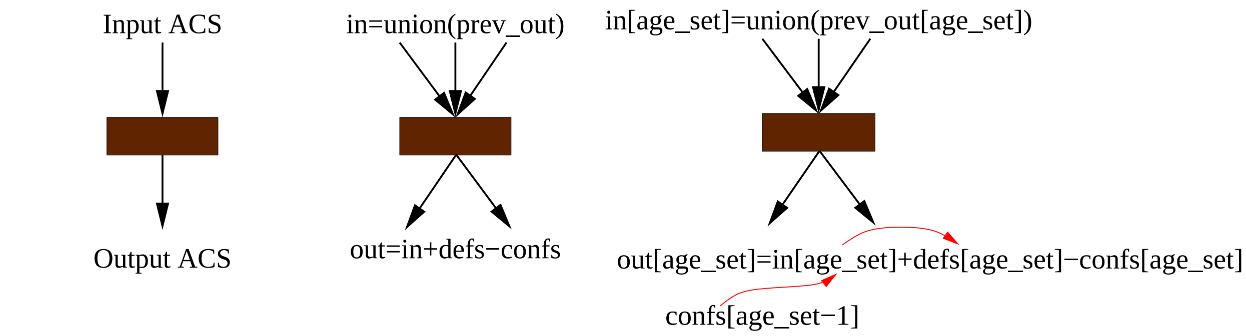
4.1 Instruction Caches

Static instruction cache simulation constitutes a method to reason about the cache content for an execution point within the program on an abstract level. The method is formulated through data-flow recurrence equations, implemented by an efficient iterative fixed-point algorithm, proved correct and evaluated for a number of sample programs. Overall, this work provides a comprehensive study of static instruction cache prediction for set-associative caches with least-recently used (LRU) replacement policy.

The idea of using a data-flow approach for cache prediction originates from a study on cache simulation for direct-mapped caches with various applications [43]. Since then, the method has been revised and formalized to simulate associative caches with arbitrary associativity through an extended data-flow framework [52]. Other groups either adopted our idea to develop data-flow frameworks but utilized different semantics [14] or they used other methods for cache prediction, such as integer linear programming or finite automata [36, 23].

Static cache prediction through data-flow analysis abstracts from the possible *concrete cache states* (CCS) that may occur during a program's execution. It reasons about *abstract cache states* (ACS) that reflects for each cache line (a.k.a. cache block) the *set* of program lines (*i.e.*, a block of consecutive instructions) that it may contain. In other words, the ACS provides information about items that *may* be cached while the CCS reflects what *must* be cached.

Both ACS and CCS can be represented as a set of program lines. As such, the ACS can be regarded as the union of all CCS' at a point of execution where the program lines for a cache line are combined in one set. Thus, the storage complexity of the CCS may be exponential while the ACS has a polynomial storage complexity. The ACS information can be constructed over the control flow of the program by calculating an input and output ACS for each basic block as depicted in Figure 4.1(a). The ACS provides information about which program lines *may* be cached at this point. Using a forward data-flow formulation with unions on joining edges in the control flow, the input and output ACS is determined as depicted in Figure 4.1(b). The input ACS is the union of the output ACS of all predecessors in the control flow. The output ACS *out* of the current block is calculated as the input ACS *in* plus all program lines in the current block (*defs*: defined lines) minus all program lines that conflict (*confs*) with the lines in the current block. This data-flow problem for simulating direct-mapped cache is formalized in six recurrence equations.



(a) input/output ACS (b) direct-mapped (c) set-associative

Figure 4.1: Structure of basic blocks

A generalization to set-associative caches with LRU replacement adds two equations in the data-flow specification resulting in separate equations for defining and conflicting sets, depending on the level of associativity. Figure 4.1(c) depicts the calculations pictorially. A cache set itself consists of a set of states for each age level, one for each level of associativity. The input ACS for an age level is calculated as the union of the corresponding sets of the predecessor blocks. The output state still results from the input state plus the defined lines minus conflicting lines but with respect to the age level. Conflicts depend on definitions within the same associativity while definitions depend on the conflicts of the previous age level. This dependency reflects the operational characteristics of LRU caches, where the least recently used reference is removed from cache when a new reference is stored in a cache set while all other lines in cache age. The aging is reflected by dependencies between age levels.

So far, the *intra*-procedural data-flow framework has been described. In an *inter*-procedural setting, one may simply propagate the output ACS for a caller to input ACS of the entry blocks of a callee and the output ACS at exit blocks of the callee to the input ACS following a call. However, calls to the same function from different call sites may expose different cache behavior. For this reason, the following technique is employed. The call graph representing the calling structure of a program is transformed into a function instance tree where for each calling context a different instance is created. This allows the distinction of the cache behavior depending on the calling context. Also recall the assumption from Chapter 2 that there be no recursion, which guarantees that the call graph is non-cyclic, thereby ensuring that a corresponding function instance graph constitutes a tree. If recursion was allowed then the function instance graph would also become cyclic. Direct recursion can be represented by two instances of a function, the

category	1st reference	2nd..n-th reference
always-hit	hit	hit
always-miss	miss	miss
first-hit	hit	miss
first-miss	miss	hit

Figure 4.2: Instruction Categories

first instance representing the initial call and the second instance consecutive calls. One edge would link the first to the second instance, another would lead from the second instance to itself to represent recursions. This would still allow the distinction of cache behavior for initial and consecutive calls. Notice that function instance graphs have also been developed in independent efforts on alias analysis within optimizing compiler [13].

The goal of inter-procedural data-flow analysis is primarily the calculation of the ACS for each block. In addition, other data-flow information is gathered in the same manner. This information provides the grounds to determine the cache behavior of an instruction reference by distinguishing first and consecutive references through the following classification (see Figure 4.2). An always-hit experiences a cache hit on the first and consecutive references. Conversely, an always-miss will result in cache misses for first and consecutive references. A first-hit will hit the cache on the first reference but misses the cache for consecutive references. Conversely, a first-miss will miss the cache on its first reference while hitting thereafter.

This instruction categorization is automatically derived from the data-flow information calculated before. A separate categorization is supplied for each loop nesting level of an instruction. For example, Figure 4.3 depicts two nested loops with instructions a and b within program lines x and y, both mapping into the same cache line. Assuming a direct-mapped cache, only one of these program lines may be cached at any time. Consider instruction b in the inner loop. On the first reference, the instruction will miss the cache since x is cached. On consecutive iterations of the inner loop, b will hit the cache since y is already cached. Hence, instruction b is a first-miss for the inner loop. Now, consider instruction b within the outer loop. The worst-case behavior of b is a miss each time the outer loop is executed, *i.e.*, an always-miss. This is due to the fact that x and y replace each other in cache for each iteration of the outer loop.

A formal specification of the worst-case instruction categorization for set-associative

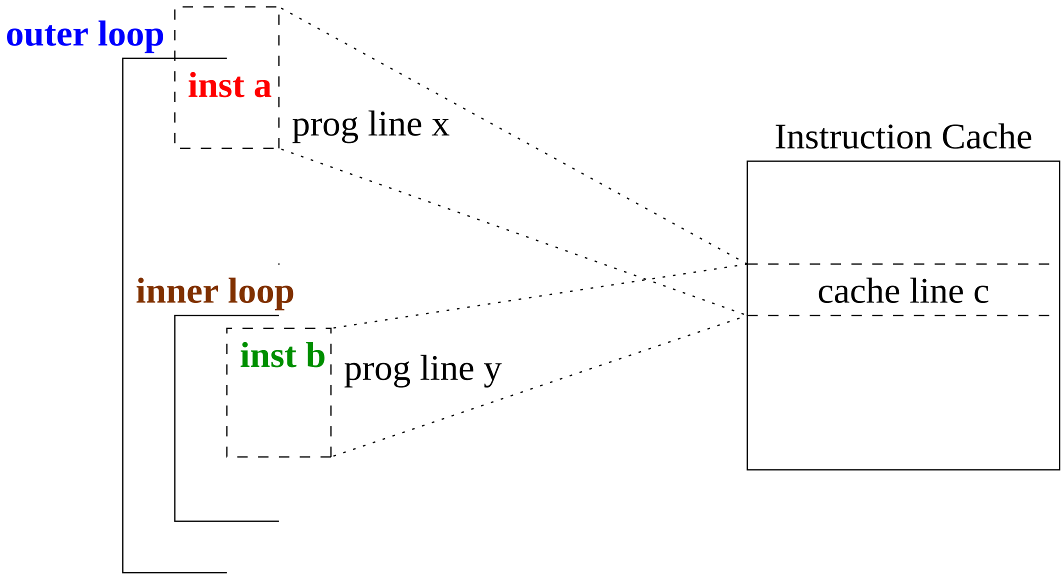


Figure 4.3: Structure of Basic Blocks

is given in [52]. The categories are derived from the abstract, linear, dominator and post-dominator cache states as well as the loop nesting. The abstract cache state has been described above. The linear cache state results from a forward data-flow analysis over the forward edges of the control flow, *i.e.*, only backedges due to loops are excluded. It allows inferences about lines cached in the absence of loops. The dominator cache state informs about program lines certain to have been traversed in the past. Conversely, the post-dominator cache state supplies information about future lines certain to be reached. Nesting levels are distinguished for nested loops and call chains at the same time, which will be discussed in more detail in the next chapter.

The derived cache categories describe the cache behavior of instructions in sufficient detail to take them into account during pipeline simulation, also discussed in more detail in the next chapter. Experiments conducted to evaluate the quality of these cache predictions show that the estimates provided by static cache simulation are within 1% of the actually observed number of cycles during program execution, except for cases with data dependencies where results vary between 3% and 14% relative to the actual value. Furthermore, the results of static cache simulation either match or exceed the actual values for worst-case predictions. Thus, this approach provides a safe, conservative approximation of the cache behavior with tight bounds on the WCET for real-time applications.

4.2 Data Caches

In the following, a brief description of static data cache simulation will be given. The details of this topic are beyond the scope of this work as they have been investigated in a publication [69] not considered to be an integral part of this work. However, the data cache work is considered in the overall tool set, which justifies a brief description.

Static data cache simulation is based on the same ideas as the aforementioned work on instruction caches. Unless instruction references, data reference are much harder to predict. Data references often refer to non-discrete values. For example, an array reference within a loop often refers to a range of addresses depending on the indexed value. The index may be a constant, an induction variable or even an arbitrary expression. Other data references may include references to variables on the stack or deference of pointers to access values. Many of these references complicate static analysis of data caches.

In Chapter 2 it was assumed that heap allocation not be permitted within hard real-time applications. By going one step further, one may assume that pointer dereferences do not occur at all without restricting the expressiveness of a programming language. Thus, ranges of data references may be obtained, which can be translated into relative and then virtual addresses before cache categorizations can be determined.

Relative addresses may be obtained during compilation by expanding expressions describing the referenced address according to the surrounding loop information. This expansion results in discrete values of loop invariant base addresses and offsets but excludes loop dependent portions, in particular dependencies on induction variables. These relative addresses are subsequently transfered into virtual addresses by distinguishing global base references from local references on the stack. The former impose little overhead since they have known absolute base addresses. The latter require that the stack height be calculated. Recall that functions are already distinguished by instances indicating their calling context. Based on the calling sequence, the stack height for a function instance may be calculated and then simply added to the offset (replacement) for local variables.

So far, virtual addresses have been calculated but there is no information on how these addresses are mapped to the physical address space. Notice that virtual and physical addresses are identical if the system page size is a multiple of the data cache size, which shall be assumed in the following. For example, the MicroSPARC I has a 2 KB data cache and 4 KB pages.

Scalar data references are categorized within the four categories of the last section.

Array references, however, require knowledge about the initial index value and the stride. Based on this information and the above data-flow framework extended by range references, both special and temporal locality of array references can be inferred. For this purpose, another category is introduced that is called *calculated* and supplies a number n. This number indicates the maximum number of cache misses for a data references in the worst case. By taking calculated references into account during pipeline simulation, tight bounds for data caches can be obtained, even for non-scalar references.

Chapter 5

Pipeline Simulation and Path Analysis

The approach of simulating hardware components separately taken in this work actually combines pipeline simulation and path analysis. Pipeline simulation determines the temporal overlap of consecutive instructions within a processor pipeline. Path analysis identifies the shortest paths for BCET and the longest paths for WCET within the control flow. While pipeline simulation is restricted to a single path, path analysis combines two or more paths, first within the intra- and then within the inter-procedural control flow. These two tasks are integrated since combining paths involves the same considerations as adding another instruction to a partially simulated single path. In the following, a brief outline of pipeline simulation and path analysis will be given. A complete description can be found in another publication [19], which constitutes an integral part of this work. After the general overview, the following discussion will focus on cache-related issues developed by the author of this document.

5.1 Pipeline Simulation for Blocks

The temporal impact on caching and pipelining are dependent on each other. For example, a cache miss on an instruction fetch stalls the pipeline for the latency (miss penalty) of a reference through the memory hierarchy. The last chapter illustrated how cache behavior of references can be categorized. These categories, in turn, provide the means to perform pipeline analysis while taking cache effects into account. Hence, cache simulation and pipeline simulation can be separated although they are related to one another.

In the following, simple scalar pipelines are assumed that do not execute instructions in parallel during any stage. The temporal characteristics of a path are simulated by concatenating instructions within one path. For each instruction, cycle information for its edges are calculated, *i.e.*, the cycle time when it enters the first and exits the last pipeline stage are determined. For a partial path, the next instruction is added at the earliest possible time taking structural hazards, data hazards and resource requirements of the new instruction into account. In addition, cache categories are considered, *i.e.*, in the event of a possible cache miss the latency of a memory reference will be considered as a pipeline stall. Since hazards, multi-cycle instruction executions and cache misses may occur simultaneously, their latencies are overlapped to reflect the maximum latency incurred over a set of instructions.

5.2 Path Analysis

During loop analysis, consecutive paths within a loop or a function instance are combined into longer paths. When two paths are concatenated, the same rules of pipeline simulation apply, *e.g.*, the temporal effect of hazards crossing paths have to be considered. This is achieved by using the pipeline simulation framework on path edges that only represent the leading and trailing edges for each pipeline stage. Trailing edges of the prior path and leading edges of the next paths will be maximally overlapped where permissible.

Loop analysis is also subject to multiple traversals of already encountered paths. For a loop without conditional execution, the loop body may simply be concatenated several times. In a naive approach, $n - 1$ concatenation would have to be performed for a loop with an upper bound of n iterations. Instead of using such an approach with high complexity, a fixed-point algorithm was derived based on the observation that the timing predictions for paths in a loop may only change over iterations until certain cache references have been encountered. The temporal changes between iterations stem from first-hits and first-misses, which expose different behavior for first references than consecutive references. The fixed-point algorithm concatenates the longest paths starting with the first loop iteration for as many more iterations as paths with new first-hits and first-misses are encountered. The cycle time of the last path then constitutes the longest path for any consecutive iteration for the WCET. Hence, after reaching a fixed point, the remaining worst-case iteration time can be approximated as the length of the last path for the remaining number of iterations. For the BCET, the shortest initial path is determined with first-misses and first-hits interpreted as misses and hits, respectively. Then, the shortest path for consecutive iterations is determined with inverse interpretations for first-misses and first-hits. These interpretations reflect the best-case behavior for the given non-regular cache categories. The initial path and $n-2$ instances of the consecutive path are then concatenated, where n represents the minimum number of loop iterations. Notice that exit paths are treated separately as premature loop exits during the last iteration.

5.3 Program Analysis

Program analysis utilizes timing predictions of loops and function instances to obtain timing predictions for a task or program. Initially, a timing tree is constructed that represents the inter-procedural calling structures of the code subject to analysis. Nodes in this tree represent loops or function instances while edges correspond either to loop nesting or function calls. Program analysis comprises a bottom-up traversal of the timing tree. Leaf nodes are subject to path analysis as described in the last section. If timing predictions have been obtained for all children, the analysis may proceed with a non-leaf node. The paths through the current node are analyzed just as in the last section, except that cycle times for called functions and lower loop nests are supplied by a child node. At the same token, pipeline information of the child node is combined with the path of the current node.

Certain cache categorizations are subject to an additional adjustment (see Table 5.1). For example, a first-miss at a child loop also classified as a first-miss within the current loop only incurs a miss on its first reference. When n iterations of the current loop call n entries to the the child loop, then n considerations of the child's cycle time will include n first-misses counted as a miss. This value is later adjusted by subtracting $n-1$ miss penalties from the predicted time of the current node for each first-miss to first-miss transition. This adjustment prevents overestimations and contributes significantly to tightened timing predictions. Similarly, transitions from always-misses within the child to first-hits within the parent require a single adjustment for each child invocation, *i.e.*, the miss penalty is subtracted once.

Child $\rightarrow$ Parent	Action to Adjust Child Loop Time
first-miss $\rightarrow$ first-miss	Use the child loop time for the first iteration. For all remaining iterations subtract the miss penalty from the child loop time.
always-miss $\rightarrow$ first-hit	For the first iteration subtract the miss penalty from the child loop time. For all remaining iterations use the child loop time directly.

Table 5.1: Adjustment of Transitions within the Timing Tree

Figure 5.1 depicts a sample tree to illustrate the bottom-up traversal for WCET prediction. Initially, the timings for the leaf nodes of function value with instances 1

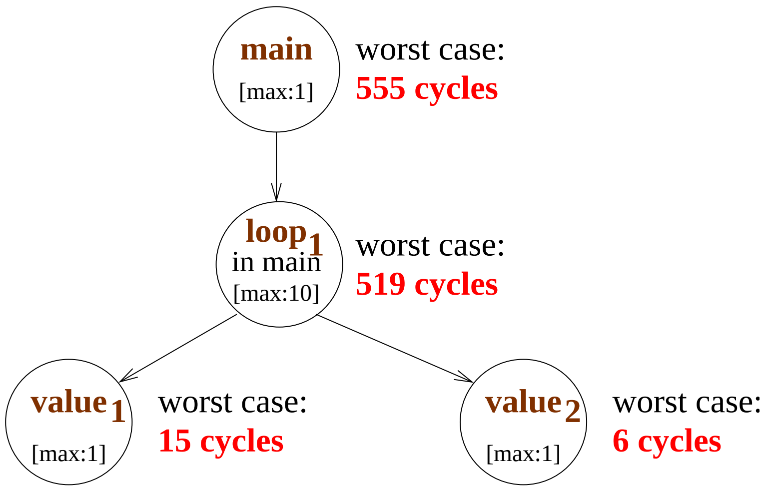


Figure 5.1: Timing Tree Traversal

and 2 are calculated. Then, paths in loop 1 are analyzed and any calls to the function instances cause the child timings to be considered within loop 1, potentially subject to the previously mentioned adjustments. Notice that loop 1 has an upper bound of 10 loop iterations, which will cause 10 invocations of each function instance assuming an unconditional execution of these instances. The prediction for main comprises its paths and the loop, whose timing prediction is again reused.

5.4 Results

Measurements for a simulated MicroSPARC I pipeline and a 128B direct-mapped instruction cache with a 16B line size were conducted for a number of test programs. Most results were very tight, in particular for the WCET prediction. The results obtained for a sorting program where less tight, which is attributed to loop nesting where the inner loop bound depends on the value of the outer loop iterator. This problem was later addressed within the pipeline simulator [20]. Tightness may also be compromised where data dependencies between execution paths exist, as found in another program. Furthermore, while most BCET predictions were tight, the same outliers as for WCET can be observed due to the above reasons.

Overall, the quantitative results show tight predictions of the WCET. These results were obtained through efficient analysis methods by algorithms whose correctness is shown in [19]. This paper provides the details of the techniques and the evaluation while this chapter summarized these issues. The analysis methods accurately predict the timing behavior of architectural components. The measurements also indicate that pipeline analysis by itself and cache analysis by itself (*i.e.*, without simulating the other component) result in just as tight results as a combination of cache and pipeline analysis. This clearly indicates that separate analysis of architectural components does not adversely affect the quality of the results but rather provides an efficient and systematic solution to the approach of static timing analysis.

Chapter 6

Comparison with Dynamic Timing Analysis

A comprehensive evaluation of static timing analysis should also include a comparison with dynamic analysis methods to validate the results. This chapter provides a quantitative comparison between static timing analysis (SA) discussed so far and one dynamic testing method, namely evolutionary testing (ET), which has shown to yield the best results of all dynamic techniques. In the following, static and dynamic analysis methods are contrasted. Then, a brief description of ET will be given followed by a concise presentation of the results obtained in a comparison with SA in a separate publication [68], which constitutes an integral part of this work. Another central part of this study is its implications on general testing procedures for soft and hard real-time applications, which is also addressed here.

6.1 Dynamic Timing Analysis

The execution time of code portions is traditionally gathered by direct timing, *i.e.*, a *dynamic* timing analysis is performed during task execution. The time may be measured by an existing timer chip, a passive logic analyzer or by cycle counting for executed instructions. The BCET and WCET should be triggered by feeding the task with the best-case and worst-case inputs, respectively.

Several problems arise with this approach. First, cycle counting based on instruction execution does not cover latencies imposed by pipelining or by references of values stored within the memory hierarchy. In general, architectures are becoming more complex causing this approach to become less and less applicable. Second, even for moderately complex tasks, the best-case and worst-case inputs may not be known. But even if they were known, they typically reflect the best and worst case in terms of algorithmic complexity. However, due to architectural complexities, the best and worst *temporal* behavior may be experienced under different circumstances (different execution paths) than the corresponding cases for algorithmic complexity.

Alternatives are provided by exhaustive testing, random testing and evolutionary testing. Exhaustive testing measures the execution times for all valid inputs according to an input specification of a task. Random testing measures execution times for randomly generated inputs that comply with the input specification of a task. Evolutionary testing is a feed-back method where the execution time for an input is measured and, based on the "quality" of this result, another input is generated.

Exhaustive testing is generally infeasible due to the exponential size of the input space even for small pieces of code but this method is guaranteed to find the BCET and WCET. Random testing covers a subset of the possible input, just as evolutionary testing does. However, evolutionary testing is likely to examine more inputs close to the BCET or WCET due to its feed-back mechanism, which guides the search in the input space. Neither random nor evolutionary testing are guaranteed to determine the actual BCET or WCET. However, experiments will show that evolutionary testing may provide extreme execution times close to the actual BCET and WCET for a number of test cases.

Dynamic timing analysis assumes that the code of a task can be executed stand-alone for known input specifications and that the code terminates. For some applications, a single task may not be separable from the rest of the application due to inter-task dependencies or dependencies on the execution environment, such as sensors and actuators. In such cases, dynamic timing analysis requires instrumentation for each task within the application to extract its timing characteristics.

In general, dynamic timing analysis cannot provide extreme execution times suitable for the temporal verification of real-time systems by schedulability analysis. Nonetheless, practitioners often employ these methods and simply try to ensure that the overall system is underutilized.

6.2 Evolutionary Testing

Evolutionary testing is a dynamic testing method originally developed to test the I/O relation of programs. This method provides an alternative to exhaustive testing, which may not be feasible even for moderately complex input specifications, and random testing, which may not always provide the best coverage of the input space for testing. Evolutionary testing is a feed-back method where the output of the program may be evaluated to guide the selection of consecutive inputs during several rounds of testing. For timing analysis, the metric to evaluate the output is given by the execution time for a corresponding input. Based on the "quality" of this metric, another input is generated. The concept behind this method is to successively generate new inputs that represent better combinations in a search for a higher quality output. Hence, evolutionary testing is closely related to genetic algorithms and neural networks, both in terms of its approach and operational description, although its application area is quite unique.

Figure 6.1 depicts the operational stages involved in evolutionary testing. After an

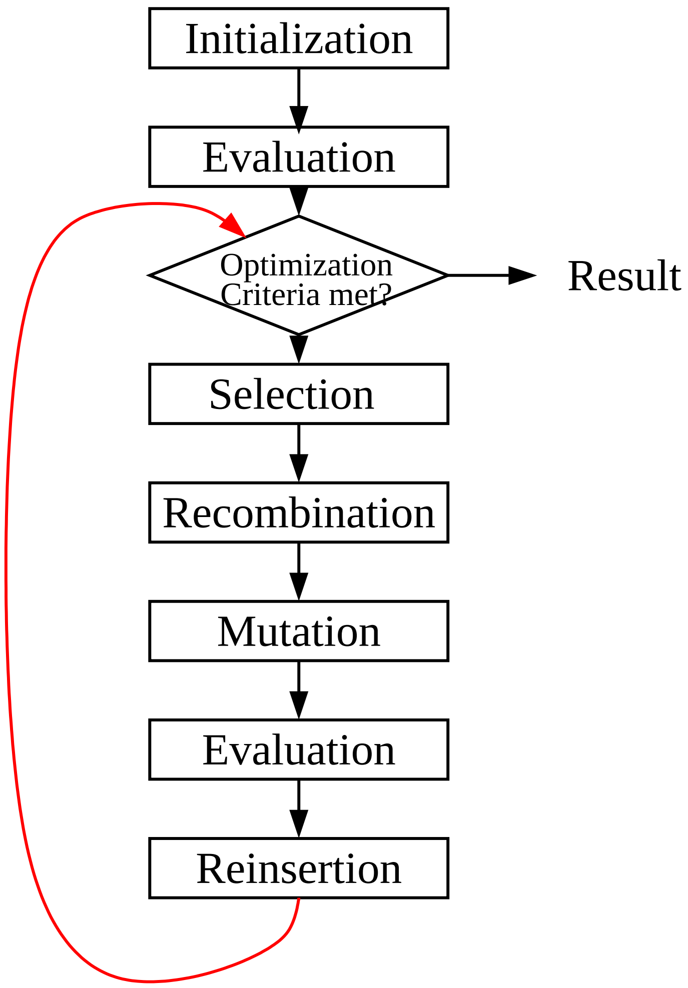


Figure 6.1: Evolutionary Testing

initial trail of an input (a population), its quality (fitness) is evaluated. Then, new inputs are generated repeatedly until either an optimal output quality is reached or another termination condition applies, such as a maximum number of generations has been reached. The iterative steps include three fundamental principles of evolutionary methods, namely selection of individuals (inputs) that should be varied, recombination of values for these individuals and mutation of selected values by limited variation. The resulting individuals then undergo a new evaluation by presenting a test program with the newly generated input. This supplies another output whose quality will be compared to previous outputs. Depending on the fitness of the output, the new individuals may replace a subset of the population with lower fitness values creating a new population with higher fitness. This process reiterates until certain optimization criteria are met.

Evolutionary testing was first developed to provide better coverage of the input space during automated system testing. The method was later applied to dynamic timing analysis by using the execution time of an execution as a metric for the fitness of an output. Hence, evolutionary testing may be applied to guide the search in the input space towards inputs resulting in either short or long execution times, respectively. Notice that two separate tests are required, one for finding the BCET and one for the WCET. Furthermore, recall that dynamic testing (with the exception of exhaustive testing) may not result in finding the actual extreme execution times but may provide close to optimal results.

6.3 Static Timing Analysis

Static timing analysis does not rely on the actual execution of a task's code. Instead, it involves the analysis of the control flow of a task and the simulation of the hardware components. This approach yields an analytical method to determine the extreme execution times on a cycle level. It completely abstracts from the input. The analysis results in guaranteed predictions of the extreme execution times, *i.e.*, estimates are conservative in that the best case is never underestimated while the worst case is never overestimated. The results are directly applicable to schedulability analysis of real-time systems. If the WCET estimates are tight enough, this method may also allow higher utilization of the system. For exact estimates, full utilization may theoretically be achieved.

Figure 6.2 depicts a range of timing measurements that may be obtained by dynamic methods and discrete values for static timing analysis. Static timing analysis typically results in overestimations of the WCET and underestimations of the BCET. Both dynamic and static methods may or may not find the actual extreme times. Both methods also bound the actual extreme time from opposite sides, which may serve as an indicator of the quality of predictions in the absence of known actual times as discussed in Chapter 6. However, only static timing analysis may provide safe estimates of the execution time suitable for schedulability analysis.

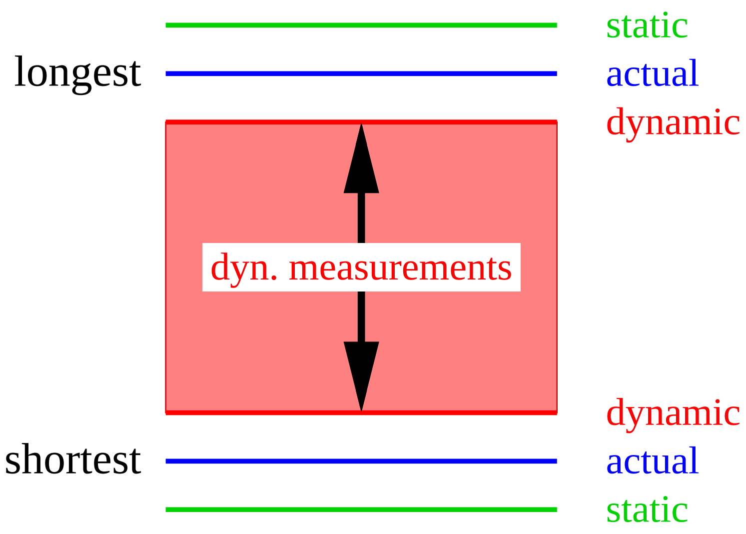


Figure 6.2: Static vs. Dynamic Timing Analysis

6.4 Quantitative Comparison

Measurements were conducted for a number of test programs, including previously used benchmark programs to serve as a reference and industrial real-time benchmarks. The chosen target platform was a SPARC IPX, *i.e.*, a processor with a simple scalar pipeline without caches. Experiments for ET utilized a cycle-level timing tool, Quantify from Rational, which obtains instruction frequencies and calculates the corresponding timings based on instruction execution information. This tool does not take into account the effect of caches or the overhead of operating system calls. Hence, the SPARC IPX was chosen as a target architecture for testing since it does not contain any caches. Consequently, timing predictions for SA were based on pipeline simulation and path analysis but excluded cache simulation for these experiments.

For two test programs, actual extreme execution times could be obtained since the worst-case and best-case input were known for this simple architecture. In theses cases, timing predictions of SA and results from ET ranged from very tight results of less than 1% to a 24% difference when compared with the actual values. The results by ET were tighter but the testing overhead was often more time-consuming. In one case, a special combinator function was introduced to better reflect the correlation among inputs in the program, which are due to the algorithms within the program. In other cases, a large number of generations was necessary to obtain satisfactory results, which, in one case, even resulted in finding the actual BCET.

The industrial test programs are subject to more complex inputs, which did not allow us to determine the worst-case inputs. As a result, the actual extreme execution times were not known. This is the general case for real-world examples with moderately complex inputs. The numbers obtained for SA and ET can, therefore, only be compared with each other. However, we know that the actual execution time is bounded from opposite ends by SA and ET, both for the BCET and WCET. Hence, the difference between SA and ET results may serve as a metric for the tightness of the measurements. Two industrial benchmarks showed differences between 4% and 32%, where the worst-case predictions were generally tighter. The third industrial benchmark experienced larger discrepancies between SA and ET. A thorough analysis revealed that the two methods used different information about the execution length of certain multi-cycle instructions. So far, the discrepancy between the pipeline model of SA and the cycle information used by Quantify could not been resolved. Lack of sufficient information about a processor architecture may inflict such problems, in this case for both SA and ET. Notice though that different timing methods for ET, *e.g.*, using on-chip timers, would not require detailed knowledge about the architecture while SA always requires cycle-level information.

6.5 Conclusion and Guidelines

A qualitative comparison between SA and ET established the importance of both methods. Essentially, the methods are complimentary. They bound the actual extreme execution times from opposite ends. SA provides safe timing predictions that can be directly used within a schedulability analysis framework. ET results in timings that were actually observed during execution and also serves as a means to test the actual application. SA requires a single simulation run whose complexity depends on program structure and size. ET includes many executions of the test object, both for BCET and WCET determination. SA already provides timings for program parts, which can be obtained by ET only through separate executions with appropriate instrumentation. The portability of SA is constrained by the retargeting effort for a compiler back-end and the availability of processor information while ET's portability depends on the actual timing method used. Some small interferences are not covered by SA, such as DMA and memory refresh cycles. ET may experience larger interferences, *e.g.*, due to caching for tools like Quantify or due to operating system overhead when using on-chip timers. Overall, SA can be regarded as a white box test method while ET is closer to a black box method.

The impact of SA and ET on real-time testing can be summarized by distinguishing between soft and hard real-time applications. For the design of hard real-time systems, timing predictions should be derived using SA to ensure that the results supply safe predictions applicable to schedulability analysis. For soft real-time systems, the designer may choose between SA and ET, possibly even using the mean between SA and ET. In general, logical and temporal validity should be checked for any real-time system, which implies that ET or other dynamic testing methods should be used in any case during system testing.

Chapter 7

Task Analyzer

The task analysis component shown in Figure 1.8 represents the missing link in the tool set at this time. The purpose of this component is to perform an interprocedural analysis of the control flow already supplied by the compiler, to identify task boundaries and synchronization points, and to issue timing requests to the timing analyzer corresponding to task phases. The design of the path analyzer includes the specification of a set of primitives that represent synchronization points for a given source language. For example, synchronization points are well-defined language constructs identified in the Ada 95 reference manual [25]. For the source language C, concurrency may be supported by POSIX Threads [64] where similar synchronization points to Ada 95 can be identified, in particular, since there exists a clear correspondence between Ada 95 tasking constructions and their mapping onto POSIX Threads primitives [17]. For each synchronization point, the equivalent primitive called by the runtime system has to be specified. The objective of the task analyzer is to traverse the interprocedural control flow, locate primitives that correspond to synchronization points and infer the corresponding task from the context of these primitives. The task analyzer can then issue timing request for the instruction sequence between two adjacent synchronization points, which corresponds to the phase of a task.

The implementation of the task analysis tool can be conducted by building a task graph from a traversal of the interprocedural control-flow graph. Basic blocks in the control-flow graph representing synchronization points (*i.e.,* calling a runtime system primitive corresponding to a synchronization point) are depicted as a node in the task graph. Edges in the task graph correspond to all possible sequences of control-flow transitions between any two synchronization points. Once the task graph has been constructed, timing requests can be issued for any adjacent pair of nodes in the task graph. The implementation of the task analyzer will then complete the set of tools for automated static timing analysis.

Chapter 8

Future Work

The components of the tool set for timing analysis (see Figure 1.8) have been implemented (with the exception of the task analyzer, as explained in the last chapter) but further development continues in several ways. The compiler modifications of Gnat/Gcc are being extended to automatically infer loop bounds where possible. Right now, the user is required to supply these bounds even if they are apparent from the source programs. Static cache analysis is supported by separate tools for instruction caches and data caches. This allows the analysis of the cache behavior for most level one (on-chip) caches since modern processors typically employ split caches on this level. The analysis of unified caches (such as found on level two / off-chip caches) is currently being implemented by taking the temporal ordering of instruction and data references into account. The implementation of the pipeline analyzer is undergoing various changes to support a portable approach that allows the specification of temporal and functional characteristics of modern microprocessors at the cycle level.

The on-going effort to implement and extend various tools reflect two efforts. First, the flexibility and portability of various tools is being enhanced to reduce the retargeting effort for different architectures. Second, the design and implementation of tools is being extended in order to allow new hardware features have to modeled. Future developments in hardware design may necessitate conceptual extensions of the timing analysis framework potentially resulting in additional components to simulate yet another hardware feature.

Chapter 9

Summary

Schedulability theory of real-time systems for uniprocessors provides a firm foundation to verify the temporal behavior of such a system. Its objective is to determine if a given set of tasks can meet their deadline. Schedulability analysis, however, can only be applied if the worst-case execution time of each task is known. The work described in the first part of this document addresses this issue. A formal framework for cache simulation is derived that provides the base for safe timing predictions for schedulability theory. This approach is transfered into a data-flow framework that has been implemented. Based on the data-flow information, the cache behaviour of memory references can be characterized. These cache categories can be interpreted during pipeline simulation, and results are combined in path analysis to predict the extreme execution times. Measurements show that tight timing predictions are obtained by this approach. Overall, the implementation of this tool set through separate simulation of architectural components results in an efficient, systematic and accurate solution of the problem. Furthermore, static simulation results are compared with dynamic testing. Both methods are found to be complementary in terms of their capabilities to predict extreme execution times. Their merits are such that they should be applied during different stages of the software development cycle of real-time applications. In summary, static timing analysis is shown to yield safe worst-case execution time bounds, even in the presence of caches. The results of static timing analysis are directly applicable to schedulability analysis and should simplify the task of verifying the temporal correctness of real-time applications.

Part II
Distributed Systems

Chapter 10

Schedulability Analysis

Part I of this work motivated the necessity of methods to determine the BCET and WCET of each real-time task to perform schedulability analysis. The methods of static timing analysis and dynamic timing analysis where introduced, implementations were developed and their results were contrasted. The objective of part II is to provide the foundations for transferring these results to distributed systems. The following chapters address this goal by laying out the foundation for supporting priorities in a distributed system such that fairness and responsiveness guarantees can be given for prioritized requests. For this purpose, we develop a novel distributed priority protocol. The performance of our protocol is evaluated in a distributed runtime system, which was also designed and implemented by us. These contributions then allow the application of timing analysis methods to schedulability analysis of distributed systems, as explained in more detail in the following.

Schedulability analysis for uniprocessors provides the means to determine if temporal requirements of a system can be met. This chapter presents several approaches on how to apply schedulability analysis in distributed systems. Distributed systems are loosely coupled systems that are connected via a communication medium but do not share memory between processing nodes. In contrast, shared-memory multiprocessors (SMPs) are tightly coupled systems where processors share memory accessed via a common bus. Parallel execution in SMPs and in distributed systems requires that temporal models be extended to model task interference and communication costs in addition to regular task execution. For this purpose, the results of uniprocessor schedulability analysis can be transferred to parallel systems in a number of ways.

Different components of a system, such as processors, shared memory or communication media, can be regarded as a processing entity subject to scheduling. In other words, schedulability analysis is not necessarily constrained to analyzing execution on a CPU but can be applied to usage of arbitrary processing components. For example, tasks on disjoint processors on an SMP may access memory in parallel, which may result in delays subject to the bus arbitration protocol. This task interference affects the temporal behavior and needs to be reflected in models of schedulability analysis. For a distributed system, the cost of communication between tasks on disjoint nodes also causes delays that have to be accounted for in schedulability analysis. In both cases, the respective component can be modeled as an abstract processor, *e.g.*, shared memory or the communication medium are processors where shared memory accesses or messages are scheduled, respectively. Hence, the term processor does not necessarily have to refer to a CPU in the context of schedulability analysis.

Parallel execution of tasks may also require synchronization between tasks. Synchronization enforces a partial order of execution between tasks if tasks are working cooperatively on solving a problem. Synchronization may also be required if parallel tasks access shared resources, such as common devices, sensors or actuators. In the following, we focus on the presentation of methods for distributed systems since the distributed models are more general and can be reduced to models for SMPs.

10.1 Resource Modeling

For real-time computing, a *local resource model* is commonly assumed, *i.e.*, a set of resources is tied to a CPU, where CPU and the set of resources together comprise a processor. In this model, phases of a task (also referred to as jobs) may require different resources, as discussed in Chapter 1. For parallel execution, a task is statically assigned to a processor. In addition, a phase requesting resource R *is required* to execute on the processor that R is tied to. Hence, a task T on CPU_1 accessing R_2 tied to CPU_2 in phase ϕ requests CPU_2 to execute the critical section ϕ on T's behalf as a *proxy*. Here, ϕ is called a *remote* resource usage in contrast to *local* resource usages that refer to accesses of resources tied to the same CPU as the requesting task. A resource requested by tasks with disjoint CPUs is called a *global* resource and requires proxy execution for remote resource accesses. This model requires protocol overhead for proxy execution, which has to be reflected in blocking times for schedulability analysis. The multiprocessor priority-ceiling protocol and end-to-end schedulability analysis adhere to this local model, as discussed in the next sections. This model reflects the assumption that devices are tightly coupled with CPUs.

An alternative model is the *distributed resource model* where resources are not tied to a CPU. Instead, resources and CPUs are loosely coupled. Communication media connect resources and CPUs in such a way that a resource may be used by a number of CPUs in sequence. Arbitration between tasks with common resource requirements has to be negotiated before critical section entry of such a phase via the communication medium. This model requires protocol overhead for distributed synchronization, which has to be reflected in the blocking times for schedulability analysis. The synchronization protocols of Chapter 12 and their extensions for the priority inheritance and ceiling protocols use this model. The model reflects the assumption that devices may be loosely coupled, *i.e.*, connected to a communication medium instead of being associated with a single CPU.

Distributed execution of a set of tasks also requires that tasks be assigned to a set of processors. A number of constraints are typically considered during task assignment. Tasks accessing common resources should be assigned to the same processor to reduce communication costs, *i.e.*, to the processor where the resource resides for the local resource model. In this case, the interference costs between tasks on the same processor has to be modeled. Tasks using disjoint resources may be placed onto different processors to increase the amount of parallelism and, thereby, the overall throughput. The remaining communication cost between remote tasks has to be modeled. Of course, these two objectives have conflicting goals, and the overall problem can be reduced to the bin-packing problem. Hence, it may not come as a surprise that the general problem of optimal task assignment is NP-hard in the strong sense [15]. Nonetheless, efficient near-optimal solutions exists, *e.g.*, through constraint formulations that can be solved by integer linear programming or by finding partitions of a graph representation of the problem[40]. The following section assumes that task assignment has already taken place. It focuses on the issues of schedulability analysis.

10.2 Multiprocessor Priority-Ceiling Protocol

This protocol adheres to the local resource model. It requires tasks and resources to be statically assigned to processors[57]. Hence, it supports local resource requirements and proxy execution for remote resource requests. A remote resource request should complete without indefinite blocking by lower priority tasks. The protocol takes this idea into account by ensuring that local task execution cannot block any proxy executions on a processor. It defines global critical sections to be scheduled at a priority higher than any local tasks.

Let $P_\top$ be the highest priority of any task in the system and P_i be the priority of task T_i. A phase of T_i comprising a global critical section is then defined to be executed at priority $P_i + P_\top$ on the remote processor, which is higher than the priority of any local task on this remote processor.

The multiprocessor priority-ceiling protocol results in potential blocking of remote requests due to a number of reasons, which can be analyzed to determine the maximum blocking factor. This blocking factor can then be used in the context of uniprocessor schedulability analysis for each processor, *e.g.*, using rate-monotone or response-time analysis as described in Chapter 1. The global blocking factor of a task is the sum of local and remote blocking, local and remote preemption, and deferred blocking, which can be characterized as follows:

Local blocking is due to delays upon resumption from remote execution when local tasks still use resources, *i.e.*, it can be expressed in terms of the local blocking factor B_i.

Remote blocking occurs when a lower priority task local to the remote processor is engaged in the critical section upon a remote request.

Local preemption may result in delays due to proxy executions on the local processor.

Global preemption may cause delays when the proxy request is preempted by other proxy requests from higher priority tasks on the remote node.

Deferred blocking results from execution of higher priority tasks on the local node.

The detailed formulas for the above items as well as adjustments of a task's execution time C_i and local blocking B_i are omitted here (see [57]) since the following chapters do not utilize this approach. The problem with the multiprocessor priority-ceiling protocol is that it does not preserve the original properties of its uniprocessor counterpart. The multiprocessor priority-ceiling protocol either cannot prevent deadlocks upon nested critical sections or it requires that state (priorities) of all tasks be globally visible before the ceiling calculation can be performed dynamically. Notice that these restrictions do not apply to the priority ceiling emulation protocol (discussed in Chapter 12) since it does not determine priority levels dynamically.

10.3 End-to-End Schedulability Analysis and Protocols

End-to-end scheduling in a distributed system assumes that critical sections are not nested and also adheres to the local resource model [63]. A task is assigned to a processor where all phases with or without local resource requests execute. A phase with a remote resource request is executed as a proxy on the remote node. The idea behind end-to-end schedulability lies in reducing the analysis to the uniprocessor problem for each processor. Local and proxy execution are modeled as periodic entities with respect to the executing processor. In addition, phases of a task have to preserve their sequential ordering though synchronization as ensured by protocols.

One protocol, the *phase-modification protocol*, delays the release time for a phase of a periodic task, *i.e.*, a consecutive phase may not be released right after the previous phase has completed execution [5]. Instead, the consecutive phase is released at a fixed time relative to the release of the previous phase. This relative delay is defined as the maximum response time of previous phase. The end-to-end response time of a task is then simply the sum of the maximum response time of its phases. Unfortunately, this protocol requires synchronized clocks, which are not physically present in distributed systems. Even for SMPs, release jitter of tasks may cause phase overruns where a subsequent phase may be released before the previous phase has been completed. A simple modification alleviates these shortcomings. A subsequent phase is released either after a delay of the its maximum response time relative to the release time of the previous phase (as before) or at the completion time of the previous phase, which ever comes later. Notification of completion between remote nodes requires communication, which has to be modeled separately. This modified protocol tolerates jitter and subsequent overruns and does not rely on global state such as synchronized clocks. However, overruns may cause non-periodic differences between the release of the same phase for consecutive jobs of a task. Hence, the distance between phases may be affected adversely such that a minimum cannot be guaranteed.

Another protocol, the *release-guard protocol*, ensures minimum distances between the same phases of a task. For the first job, completion of one phase causes a notification to release the next phase [63]. For each consecutive job of a task, different release conditions are used depending on whether or not the processor had been idle since the same phase of the previous job. Without idling, the release occurs upon completion notification as before or at the previous release time of the same phase plus the task's period, which ever comes later. With idling, the release occurs after notification or upon reaching an idle state, which ever comes later. This protocol requires that release times of phases be recorded over one period. It tolerates jitters and overruns without requiring globally synchronized clocks. In the absence of idle time, it also preserves minimum distances between consecutive phases. The end-to-end response time of a task is then simply the sum of the maximum response time of its phases, as in case of the phase-modification protocol. The maximum response time of each phase can be obtained by response time analysis as discussed in Chapter 1, where each phase on a processor is modeled as a periodic, independent task with a given release time, execution time and period.

10.4 Distributed Resource Protocols

Protocols that follow the distributed resource model where remote requests are not executed as proxies but instead result in resource arbitration due to distributed synchronization take a different approach. The protocols use notification across the communication medium, *i.e.*, through message passing, to ensure the required synchronization state. Chapter 12 discusses such a protocol for mutual exclusion in a distributed system using a token-based approach. The relation between such protocols and real-time systems are threefold. First, the protocols have to be predictable. The token-based protocol ensures predictability by bounding the number of message exchanges per request. Second, the protocols should be deterministic. The absence of global state in a distributed system makes deterministic behavior a challenge. The token-base protocol ensures deterministic ordering between requests according to priority levels and FIFO within one priority level if the minimum temporal distance between requests exceeds a certain threshold. Third, priority inversion has to be prevented. For fixed-priority scheduling with static priority assignments, the token-based protocol is shown to support the priority-ceiling emulation protocol in a distributed environment without additional message-passing overhead. In addition, support for the priority-inheritance protocol is developed, which requires additional messages with a low upper bound on the message complexity. These protocols allow the results of schedulability analysis from Chapter 1 to be applicable to the individual processors of the distributed system. The blocking times are extended accordingly to reflect the upper bounds on communication requirements due to the aforementioned protocols. This provides a new and flexible transition from uniprocessor schedulability analysis to distributed schedulability analysis of real-time systems.

Chapter 11

Distributed Shared Memory: DSM-Threads

This chapter provides a description of a programming environment suitable for distributed execution. This environment is subsequently used for experiments to implement and evaluate distributed communication protocols that are suitable for real-time systems. The protocols are described in the next chapter while this chapter focuses on the description of aspects less closely related to real-time systems.

The following sections summarize several aspects of DSM-Threads described in more detail in [46] and [45], which constitute an integral part of this work. Other aspects are described in [49, 51] and [58], which can be consulted for additional reading.

11.1 Distributed Programming Models

Distributed systems have enjoyed increasing popularity that has reached the scope of real-time systems. While distributed system provide a cost-effective means to exploit parallelism in a network of distributed computing nodes, distributed applications are inherently more difficult to design and implement than shared-memory applications running on shared-memory multiprocessors (SMPs).

In a distributed system, there is no notion of a global state due to the lack of shared memory. As a result, conventional techniques for concurrent programming cannot be applied anymore. Instead, distributed algorithms have to be developed based on explicit communication (message passing). But either such approaches use a centralized approach with servers as a potential bottleneck or decentralized approaches require new and often complex algorithms or extensive program restructuring. As a result, distributed algorithms are often hard to understand.

An alternative solution is provided by *distributed shared memory (DSM)* [54, 56]. DSM provides the logical view of a portion of an address space, which is shared between physically distributed processing nodes, *i.e.,* a global state (see Figure 11.1). Address references can be distinguished between local memory accesses and DSM accesses, *i.e.,* an architecture with non-uniform memory access (NUMA) is created. In addition, the well-understood paradigm of concurrent programming can be used without any changes

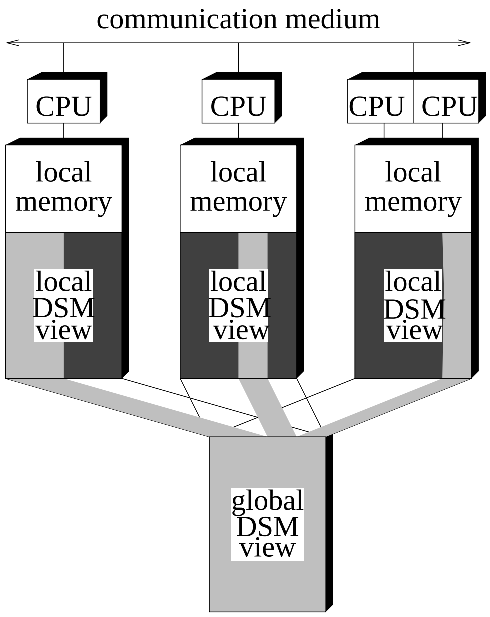


Figure 11.1: DSM Model

since the communication between nodes is transparent to the programmer. The user's view of the DSM address space provides the illusion of a global address space (global DSM view). The local DSM view depicts distinctions between regions with sufficient access rights (lightly shaded) and regions with improper access rights (darker shaded). Access rights are dynamically controlled by the runtime system and are changed when data regions are transfered between nodes. This process is transparent to the user.

This chapter introduces our DSM system called *DSM-Threads*. It provides the runtime support for distributed execution while implementing a shared-memory model in software with regard to real-time properties such as predictable latency. Hence, DSM-Threads facilitates the migration from the uniprocessor over an SMP to a distributed system in the sense that programming models and analysis methods of the temporal behavior can be reused. This chapter focuses on the design and implementation of DSM-Threads while the next chapter relates distributed protocols to real-time aspects.

11.2 System Overview

DSM-Threads utilizes an interface similar to POSIX Threads (Pthreads) [64]. The aim of DSM-Threads is to provide an easy way for a programmer to migrate from a concurrent programming model with shared memory (Pthreads) to a distributed model with minimal changes of the application code. A major goal of DSM-Threads is the portability of the system itself followed by requirements for decentralized runtime components and performance. The runtime system of DSM-Threads is implemented as a multi-threaded system over Pthreads on each node and copes without compiler or operating system modifications. A number of synchronization primitives are supported to arbitrate access to common resources, allow event notification between nodes or synchronize execution phases. Several data consistency models are supported to facilitate ports from Pthreads to DSM-Threads and address the requirements of advanced concepts for multiple writers. The system assumes a point-to-point message-passing protocol that can be adapted to different communication media. It assumes that messages are slow compared to computation and that the size of messages does not have a major impact on message latencies.

DSM-Threads differs from other DSM systems[1] in a number of ways:

- Each node is multi-threaded.
- Communication is asynchronous and tolerates out-of-order message delivery, *i.e.*, two messages from node A to B may arrive in a different order than they were sent.
- Communication is controlled by DSM-Threads to ensure a node's responsiveness and predictable temporal behavior.
- Its interface closely resembles that of POSIX Threads.
- Different memory consistency models are supported with novel protocols for speculative data movement.
- Synchronization protocols are strictly decentralized to avoid bottlenecks and optionally support the notion of priorities for real-time support.

The following sections provide a more comprehensive overview of these features.

11.3 Design

DSM-Threads was designed with a number of major goals motivated by the properties discussed in the previous section. Portability is provided in two ways. On one hand, the system implementation depends on established and emerging standards (*e.g.,* POSIX) and widely ported tools (*e.g.,* gcc and other Gnu utilities). On the other hand, the application program interface (API) of DSM-Threads also resembles that of POSIX Threads.

[1]See [46] for related DSM systems.

Hence, minimal modifications to source programs commonly suffice when migrating from Pthreads to DSM-Threads, which enhances compatibility.

The efficiency of communication is mainly constrained by the number of messages sent assuming that the network presents a bottleneck, as often given in a distributed system. Communication protocols for synchronization and memory consistency have been designed to keep the number of messages low. Furthermore, the protocols are based on decentralized approaches to avoid bottlenecks.

Predictability of the communication scheme in its response characteristics is ensured, *i.e.*, the communication scheme ensures that message reception does not block future incoming messages. For this purpose, the protocols utilize asynchronous communication, dedicate a high priority communication thread on each node and assign workers to handle messages. This ensures predictable timing behavior on message reception for an underlying network with deterministic and predictable latency characteristics. Hence, the response time of communication can be bounded, which is essential for real-time predictability.

A set of protocols offers users the option to design simple distributed programs whose performance can be enhanced step-by-step by advancing to more sophisticated protocols on demand, thereby providing extensibility and opportunities for application tuning. Different protocols are provided to support multiple paradigms, such as imperative high-level languages as well as object-oriented languages. The modular design of these protocols as well as other independent components of DSM-Threads facilitates the addition of modules to adapt the system, *e.g.*, to add new communication standards or future consistency models.

11.4 Distributed Execution and Synchronization

Execution can be dynamically distributed over a set of nodes by using an interface for remote thread creation similar to POSIX Threads. Remote threads are automatically distributed onto different nodes by default. Optionally, the application can control the distribution scheme by explicitly specifying destination nodes for remote thread creation. This allows network topologies with variable latency or bandwidth characteristics to be considered as part of the application design. Within a remote node, a user thread, a communication thread and a number of worker threads are spawned. The user thread executes part of the application program and may itself spawn additional user threads (POSIX Threads) within the process on the same node. The communication thread receives incoming messages at the highest priority to ensure that message reception does not block any user thread. Messages are then demultiplexed and may be handled by the communication server itself for small protocol jobs. For larger jobs, messages are handed off to a worker thread or directly to the user thread, where they are handled at a lower priority. Figure 11.2 depicts an example of a message exchange due to the memory consistency protocol. A user thread on node 1 experiences a page fault and delegates a

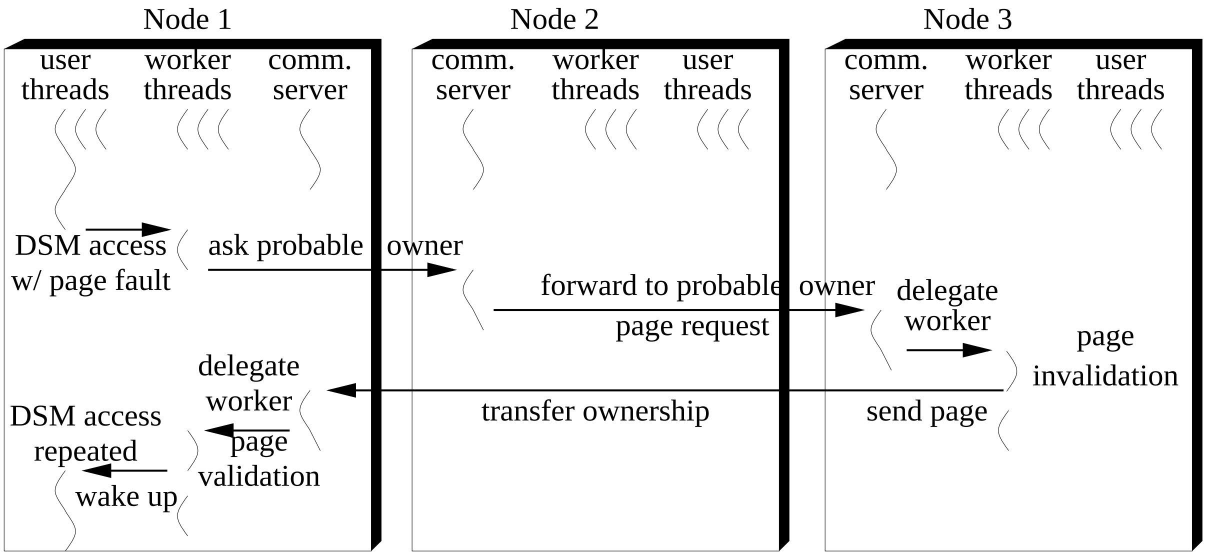


Figure 11.2: Message Exchange between Nodes and Threads

worker to act on it. The resulting request is received on node 2 by the communication server who decides to serve the request himself since he only has to forward it. The communication server on node 3 receives the request and hands it off to a worker since it requires more effort (page invalidation). The response is then directly sent to node 1, where the communication server receives data, passes it to the worker who updates local memory before waking up the user thread.

Synchronization between nodes is supported through barriers, mutual exclusion and condition variables. The design of synchronization protocols was constrained by the use of multi-threading within each node in DSM-Threads. Barrier synchronization provides the means to synchronize the execution stages of a set of tasks. Barriers are realized through a dynamically generated tree for a multicast on a barrier release, *i.e.*, only the nodes participating in the barrier are notified once all of them have reached a barrier. Barriers are realized through an initial registration phase of participating nodes to build a tree of these nodes while consecutive barrier phases reuse the edges of this tree to direct messages to neighbors (see Figure 11.3(a)). The registration phase provides the means to structure the tree such that different latency characteristics can be taken into account in a natural manner. It can also be used to ensure a LogP-like ordering of the tree to ensure a minimal overall latency [12]. If a node is multi-threaded, consecutive barrier requests are communicated to the root in a countdown phase, as depicted by the dashed edges in Figure 11.3(b). Once all threads have arrived at the barrier, each node (and each thread within the nodes) is released from the barrier by sending messages along the edges of the tree constructed during registration, *e.g.*, messages may be sent in the numerical order depicted in Figure 11.3(c). Later rounds of barriers require that each node in the tree waits for its children and its local threads to arrive at the barrier before it sends a wait message upwards within the tree, *e.g.*, wait messages may be sent in the numerical order depicted in Figure 11.3(d).

Mutual exclusion is supported by serving requests in FIFO order of their arrival over a token-passing protocol. This protocol optionally supports priorities of messages with strict priority ordering between distinct priority levels and FIFO ordering within each priority level, as discussed in more detail in the next chapter. Conditional synchronization also adheres to FIFO ordering when queuing and signalling wait requests. Furthermore, conditional synchronization follows the token-passing approach for waits and signals but utilizes a static tree for broadcasts. The decision to use a static tree for conditional broadcasts is shown to be caused by the properties of the POSIX Threads interface inherited by DSM-Threads. These protocols are enhanced to support per-node multi-threading. Several pitfalls and solutions to problems related to data and message races in the presence of per-node multi-threading are discussed as well.

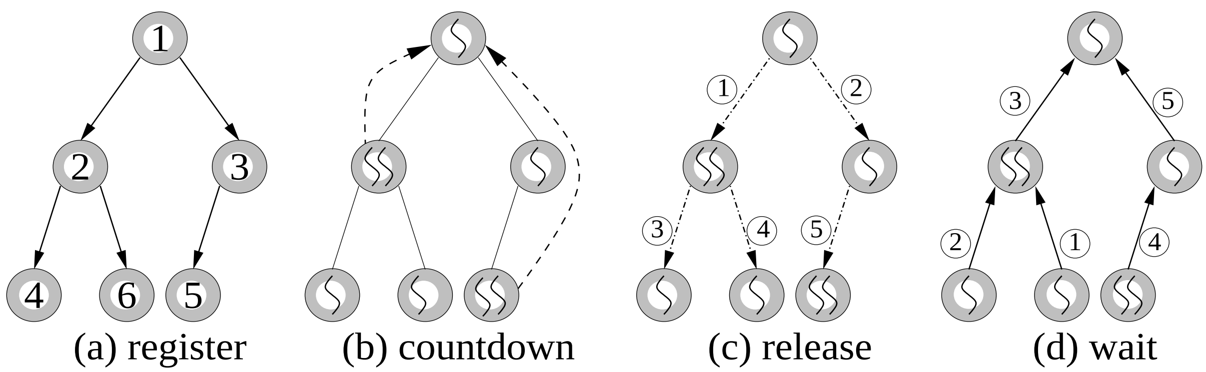


Figure 11.3: Phases of Barrier Synchronization

11.5 Memory Consistency

DSM-Threads supports a shared-memory view of a specified address range for a distributed application. The shared address range may include static global variables specified as shared within their declaration. It may also include globally allocated heap space controlled by the DSM protocols. DSM data may be controlled in software through virtual memory management by altering page protections of virtual memory pages at a coarse granularity, which is utilized for sequential, release and scope consistency discussed hereafter. Software control may alternatively be provided at a finer level, *e.g.*, at the level of variables, which requires explicit calls for DSM accesses as used for entry consistency.

Different consistency models have been proposed for DSMs to reduce the amount of required messages at the expense of additional requirements that a programmer has to be aware of. *Sequential consistency* still provides the conventional programming of SMPs but has its limitations in terms of efficiency for DSM systems. It has been commonly implemented using the dynamic distributed manager paradigm at the granularity of virtual memory pages [34]. We will refer to this implementation as sequential consistency for DSM. An example is depicted in Figure 11.4 where write accesses to variables x and y on node A are followed by an access of y on B and an access of x on C. The latter accesses occur to write protected pages causing page faults (depicted by a lightning symbol), which are handled such that requests to node A are issued. Page invalidations on A then occur before the page is forwarded to the requesters and the access is reissued with an updated page content and proper access rights.

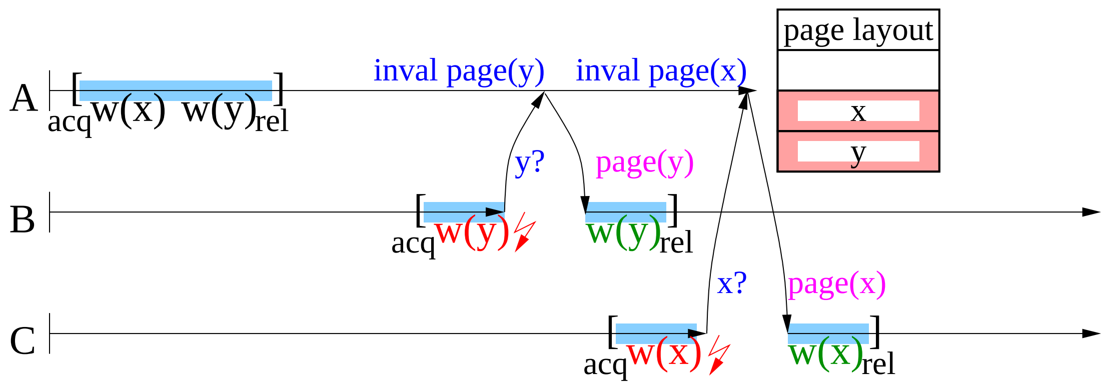


Figure 11.4: Sequential Consistency

Entry consistency requires explicit association between locks and data that is either provided by a compiler or by the programmer with a granularity of arbitrary sized objects [4]. Hence, instead of pages, only the variables x and y would be forwarded in the previous example.

Release consistency is a weak consistency model, *i.e.*, it allows multiple writes on different nodes to a page at the same time as long as the programmer ensures that the writes occur to distinct parts of this page for each node. Implementations include *eager* release consistency (ERC) where data is broadcasted at lock releases and *lazy* release consistency (LRC) where data is requested upon demand when an attempt to acquire a lock is made [26]. Figure 11.5 shows that under ERC, the previous example results in updates being broadcasted upon releases while lock request and grant messages are still required when a lock is acquired. Notice that upon a write access, page contents are mirrored in a twin page, which allows a later scan through the memory content to detect modifications (diffs). Only these modifications are then exchanged. Figure 11.6 shows that under LRC updates are piggybacked with lock grant messages. In addition, time stamp vectors t(N) are required whose components denote for each remote node which updates have been seen. These time stamps are piggybacked on each message, which allows the receiving node to update his time stamp accordingly. The lazy handling of updates has been shown to result in superior performance of LRC when compared to ERC.

Scope consistency combines the advances of entry and release consistency by implicitly associating data with held locks when a page fault occurs [24]. This may allow false sharing to be reduced while potentially incurring fewer and smaller data transfers. An example is depicted in Figure 11.7. Updates of x and y are forwarded to a home node

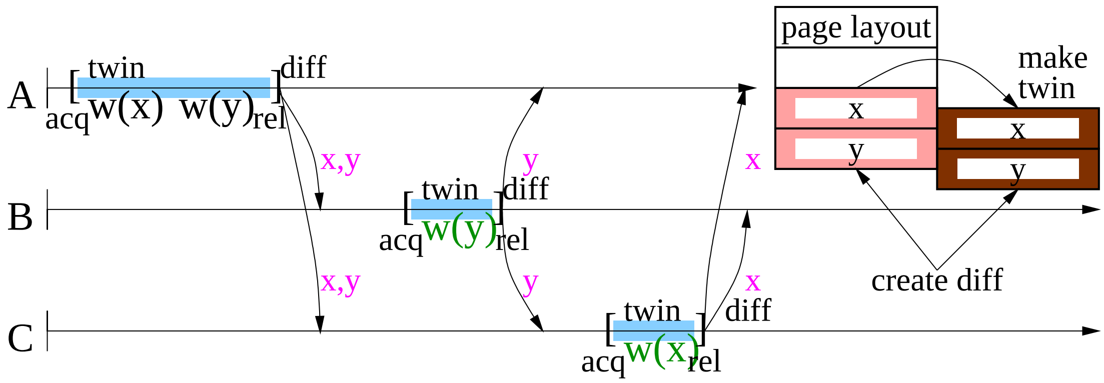


Figure 11.5: Eager Release Consistency

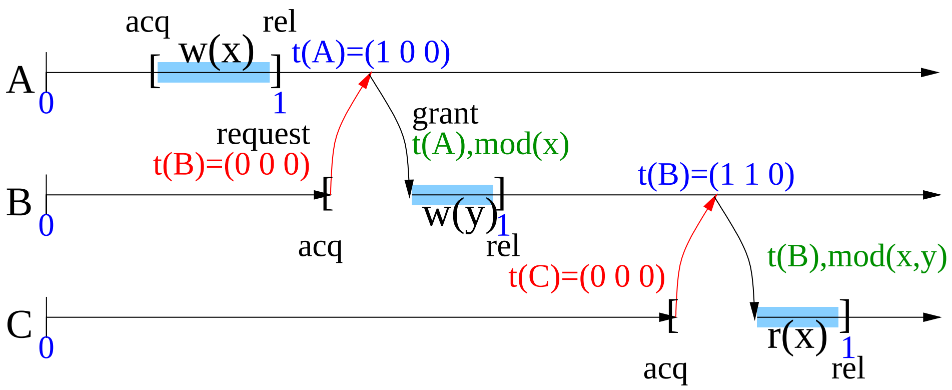


Figure 11.6: Lazy Release Consistency

that keeps the latest copy of the pages containing the variable y. An acquire of lock L2 on B causes the information about modifications of y to be piggybacked with a lock grant message so that read access on B results in a request for this page to the home node. A consecutive lock of L1 and write of z (within the same page as x) does not require an updated version of this page since x and z are protected by distinct locks, *i.e.*, consistency updates due to false sharing, which would occur under LRC, are avoided. These are the consistency models considered by DSM-Threads.

The design of DSM-Threads as a multi-threaded system per node results in extensions to the base protocols for memory consistency. Sequential consistency generally distinguishes three modes of access for a page, namely read access, write access and no access. Transitions between these modes reflect state changes within the sequential consistency protocol due to page faults. A page fault under insufficient access rights then results in message exchanges to obtain the up-to-date page content before raising access rights

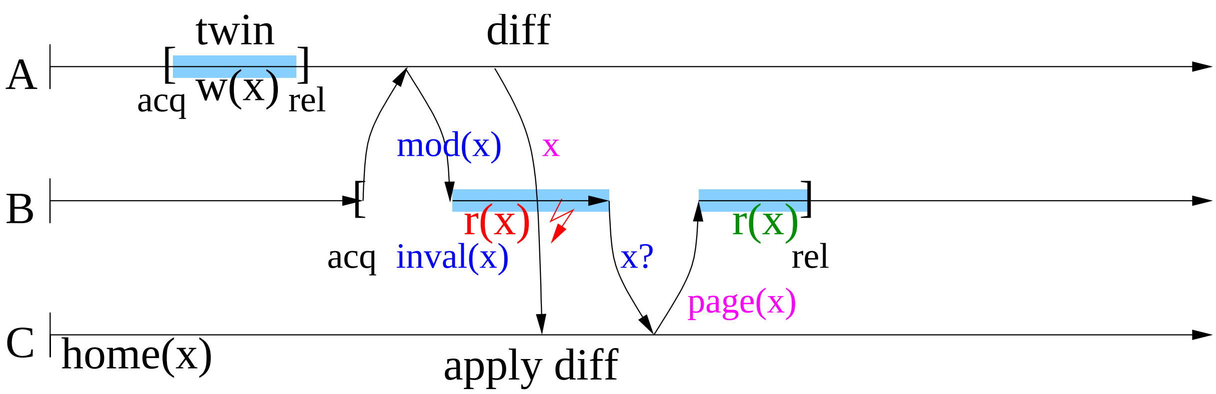


Figure 11.7: Scope Consistency

(recall Figure 11.2).

DSM-Threads enhanced this three-state model to an eight-state model to support per-node multi-threading, handle asynchronous communication and tolerate out-of-order message delivery. New states denote that an asynchronous request has been sent but the reply is still pending. This ensures that after sending a request, the node may still receive other responses intended for different threads within a node. Under tradition protocols with a three-state model, synchronous communication would block the sender such that other threads cannot be served meanwhile. The enhanced model of DSM-Threads adds states for pending actions (read, write and invalidate) and their combination. In addition, the number of transitions between states increases, which complicates the protocol but ensures that a node remains responsive with regard to incoming messages for other threads.

The support for entry consistency is combined with lock synchronization. During lock creation, data may be associated with a lock, which is then piggybacked with messages that transfer the right of mutual exclusion between nodes. Multi-threading requires that requests from local and remote threads be distinguished, which is subject to the lock protocol. Hence, entry consistency can be supported as a straight-forward extension of the API of lock synchronization.

Release and scope consistency also rely on a combination of consistency and synchronization protocols. Page faults trigger data requests as seen for sequential consistency but data updates may only include a subset of a page, according to the requirements of weak consistency that guarantee only up-to-date data when partial ordering between memory accesses is forced through proper synchronization.

Optimizations for speculative data motion between remote nodes is further formulated as an extension to sequential, entry and scope consistency. These extensions exploit regular access patterns of distributed applications by recognizing repeated patterns and then forwarding data and synchronization facilities ahead of an anticipated request. Lock speculation for entry consistency enhances the lock protocol to speculatively forward the right for critical section entry to a past requester upon leaving the critical section. The data associated with the lock is piggybacked as well so that the receiving node may not have to issue requests anymore. For sequential consistency, DSM pages are associated with a held lock on a page fault. Speculative lock forwarding to past requesters will then result in piggybacking of entire page contents. This may avoid messages for lock requests, messages for data requests and page faults under sequential consistency, as depicted in Figure 11.8. Speculation for release consistency may also exhibit lock speculation

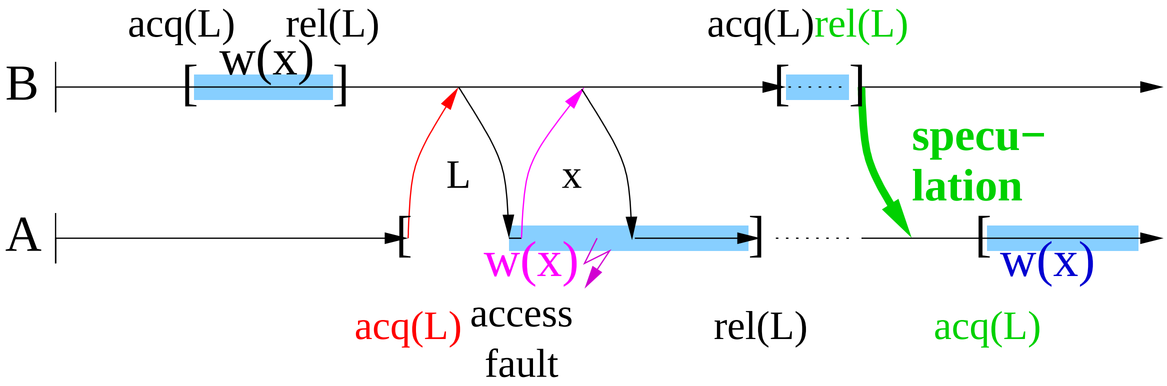


Figure 11.8: Speculative Page and Lock Migration

in conjunction with piggybacking internal protocol data used by release consistency to record the locations of data copies.

11.6 DSM-Threads as an Ada Runtime Extension

DSM-Threads provides the means to migrate from an SMP to a distributed system while retaining temporal predictability. In essence, a multi-tasking uniprocessor application may gradually be transformed to an SMP application where different task subsets execute on distinct processors sharing physical memory. In the next step, task subsets may be distributed on distinct nodes in a network under DSM-Threads while shared memory is provided through the DSM software abstraction. This model is not limited to a certain language, such as the language C, although the API of DSM-Threads and the underlying POSIX Threads are defined in C. The following description gives an account on the efforts to couple DSM-Threads with Ada 95.

The distributed systems annex of Ada 95 [25] supports the abstraction of global data shared between processing nodes *via shared passive library units* associated with a certain node (called a *passive partition*). Passive partitions can be supported in various way. For example, message passing or remote procedure calls (RPC) could be employed to fetch data of passive libraries from a certain node each time it is accessed. A much more efficient alternative is provided by implementing passive partitions *via* DSM. Here, only the first data access results in communication overhead to obtain ownership of the data. While a node owns the data, consecutive accesses are as fast as local memory accesses.

A sample implementation shows that the design of DSM-Threads allows a simple integration into an existing Ada 95 runtime system, namely the Gnu Ada runtime library (Gnarl) of the Gnu Ada Translator (Gnat) [67]. Identifying the DSM-controlled memory ranges at initialization time suffices to provide memory consistency, for example, for the sequential consistency model. The language definition of Ada 95 restricts passive partitions such that they may only contain constants, types, data and protected types without entries. Simple data accesses require sequential consistency while protected functions and protected procedures may be realized through entry consistency, according to the language rationale. These mappings from Ada language constructs to DSM consistency models provide sufficient support for passive partitions.

In Ada 95, *active partitions* are equivalent to a regular Ada 95 program with one or more threads of control (environment task plus user-defined tasks). Active partitions are equivalent to the distribution of execution over different nodes. The Gnat compiler already supports active partitions through a library called Garlic, which is part of the distributed system annex environment (DSAE) for Gnat [27, 28]. Hence, the remote execution facilities within DSM-Threads are disabled to allow a simpler implementation within the existing environment, as depicted in Figure 11.9. Alternatively, Garlic could

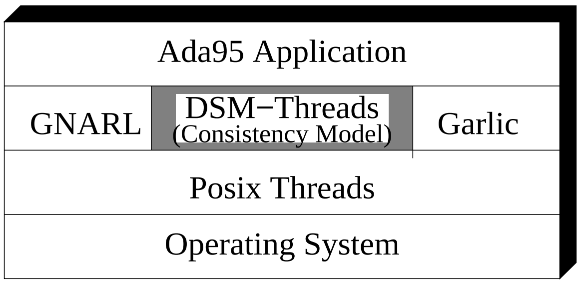


Figure 11.9: DSM-Threads Integration within the Gnu Ada Runtime-System

be completely replaced by DSM-Threads with remote execution support enabled. In this case, an extension to support the partition communication subsystem (PCS) would be required. The PCS provides an implementation of an API for remote procedure calls between active partitions, which can be implemented as an additional Ada package by reusing the DSM-Threads communication facilities. This would have the advantage that communication remains predictable while a coexistence with DSAE cannot guarantee predictability due to a lack of priority support for its communication subsystem.

Chapter 12

Priority Support for Distributed Synchronization

In a distributed system, different nodes of execution often require synchronization. Synchronization often ensures a partial ordering between execution stages of nodes in a distributed application or it may arbitrate the access to common resources. The absence of shared memory requires that distributed synchronization rely on a paradigm such as message passing to communicate between nodes. In the following, a protocol is derived that provides an efficient solution to the problem of distributed mutual exclusion over message passing while supporting the notion of priorities in a distributed system. This protocol is proved correct and extensions to prevent priority inversion are provided.

The following sections summarize the approach and the results of the protocol and its extensions as described in [48] and [50], which constitute an integral part of this work.

12.1 Token-Based Mutual Exclusion

The following protocols rely on a model of a distributed system with a fully connected network. The network should support reliable message passing. The protocols tolerate out-of-order message delivery, *i.e.*, two messages from node A to B may arrive in a different order than they were sent. The overhead of message passing is constraint by the network latency, which is assumed to be several orders of magnitude larger than the overhead of local computation. Hence, the protocol aims to reduce and bound the number of messages sent.

Several algorithms to solve the problem of mutual exclusion have been developed, distinguished by token-based and non-token based approaches, where the latter require broadcast support [11]. This work relies on a token-based approach, where possession of the token indicates the right to enter a critical section. It does not make restrictive assumptions, *i.e.*, it does not rely on broadcast support. The approach by Naimi *et al.* provides an efficient solution of the mutual exclusion problem [53]. It uses a logical tree structure where the root possesses the token and edges point towards the last requester via a probable owner chain, as depicted by the solid edges in Figure 12.1. The edges are adjusted dynamically to provide path compaction during requests. For example, a

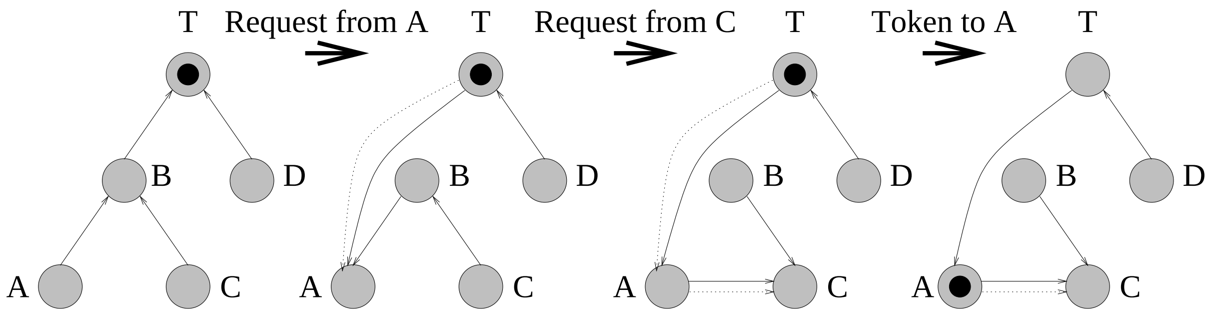


Figure 12.1: Token-Based Mutual Exclusion Protocol

request from A via B to T results in the deletion of the propagation edges and in addition of edges to the requester A from nodes along the propagation path. Another set of edges, the *next* edges, depicted as dotted edges, serve as a distributed queue of requesters, which are reduced upon token passing one at a time. In the example, the request from A results also in a next edge from T to A. A second request from C propagates via B to A resulting in new probable owner edges to C and a next edge from A to C. When T exits his critical section, the token is forwarded along the next edge to A. This next edge is deleted and A enters his critical section. The approach has a message complexity of $O(log\ n)$ on the average and a worst-case complexity of $O(n)$, where n denotes the number of nodes.

12.2 Priority Support

Priorities of concurrent threads or tasks within a node indicate the importance of a certain task. Strictly prioritized scheduling for real-time systems gives precedence to the highest priority task for execution in a uniprocessor system, as discussed in the first part of this document. In a distributed system, the relation between priorities of tasks residing on distinct nodes is not clear. On one hand, a global notion of precedence constraints between prioritized tasks, regardless of their location (node), would be desirable. On the other hand, the absence of global state in a distributed system prevents such a global notion. The following protocol fills this gap by providing enhancements for synchronization protocols to support priorities in a distributed system, exemplified by a prioritized protocol for token-based mutual exclusion.

Requests in the distributed system are assumed to be associated with a priority, where a larger value indicates higher importance. Requests are ordered by priority and FIFO within each priority level. The basic idea of the token-based protocol consists of an accumulation and logging of priority levels during request forwarding along a probable owner chain. Hence, edges are weighted with priorities. For example, when node B receives a request from A at priority 1 and forwards the request, then the edge $A \rightarrow B$ is tagged with > 1 and edges $B \rightarrow A$ and $A \rightarrow A$ with weights ≤ 1 are created (see Figure 12.2). This request then propagates to T where it is queued locally while edges are

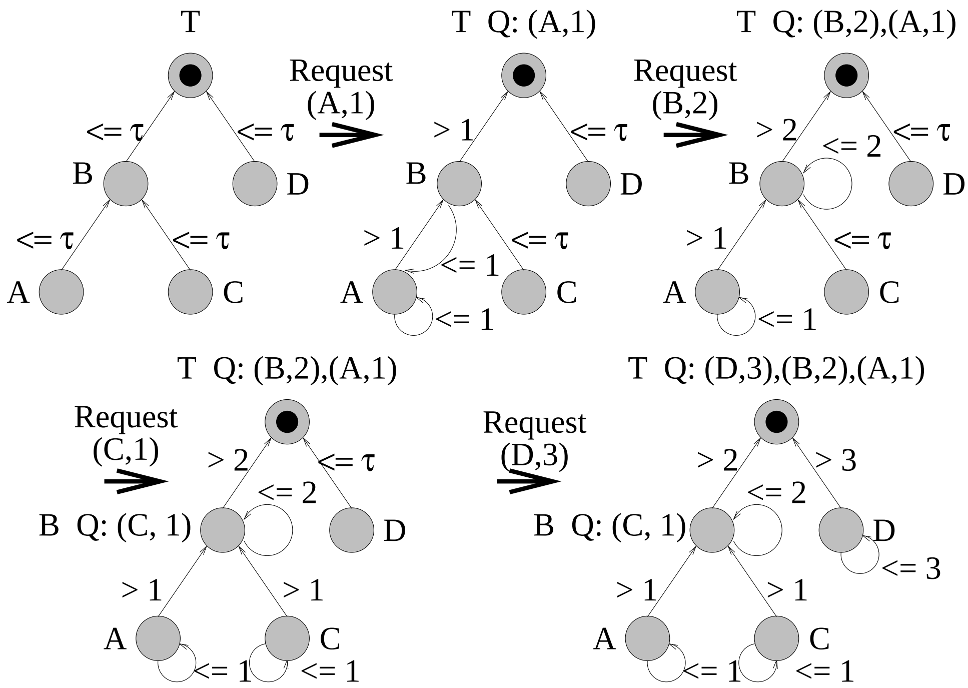


Figure 12.2: Prioritized Token Request

adjusted as before. A subsequent request from B priority 2 is handled similarly. If a later request from C arrives at B with a lower priority 1 than a previous request, the request can be queued locally since B has to receive the token before C. Conversely, higher priority requests are passed along existing edges, as seen in case of the request from B, and weights are adjusted to the higher priority. This allows requests with different priorities to be served in a strict priority ordering but it does not address the issue of FIFO queuing within a priority level.

FIFO queuing presents a problem since the temporal ordering of two requests in a distributed system may be ambiguous in the absence of global state. One solution to the problem is given by a global logical clock and the usage of timestamps but this approach incurs considerable overhead in terms of message passing [32]. Instead, a more efficient solution is employed that relies on local time facilities and estimates the remote time. It approximates remote token usage times based on its delay since the time that the request arrived at a node relative to the proportion of accumulated token usage time of a prior local request and the request latency when the token arrives. This approximation allows the formalization of a property called *relative fairness* that describes the stipulations for FIFO ordering. In essence, if two requests at the same priority on distinct nodes are issued far enough apart in time, then FIFO ordering can be guaranteed. The required distance between requests is a function of the latency characteristics of the network, *i.e.*, for deterministic latencies (such as typically found in real-time networks), relative fairness is provided if the time difference between requests exceeds a certain threshold.

Token forwarding under priority support requires that the token is passed to the head of the local queue. The tail of the local queue is piggybacked with the token and an edge with a maximal weight is created for future requests (see Figure 12.3). The receiver

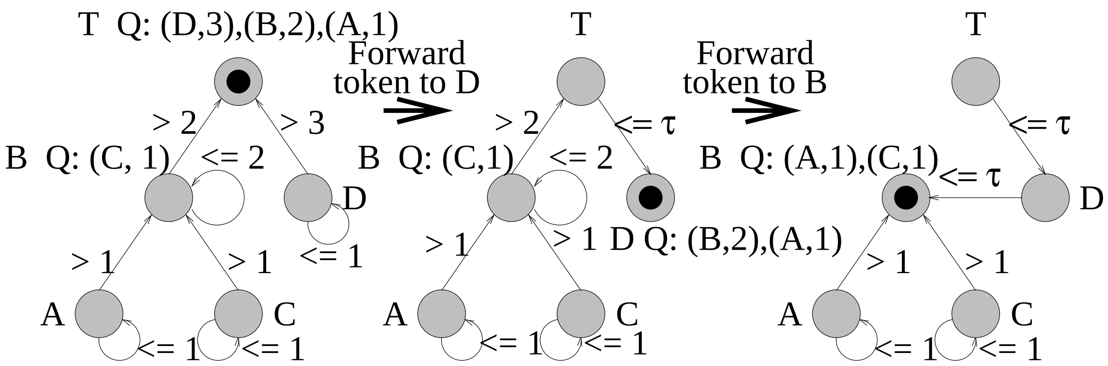


Figure 12.3: Token Forwarding

merges the received queue with its local queue according the priority levels of requests and FIFO within one priority level based on approximations of token usage times, as described above. If a local request is at the head of the queue, then the critical section may be entered. Otherwise, the token is forwarded as before.

Additional information is relayed with the token to prevent message races between high priority requests trailing lower priority token usage due to forwarding. Such message races would result in circular request propagation, which would no longer result in a linear worst-case bound of the message complexity. Hence, requests are stored along their propagation path such that token forwarding to a node along this path ensures that higher priority requests in transit are recognized early. Furthermore, requests can be aborted before they propagate in circles since such a cyclic behavior implies that the token has already reached the propagation path, thereby queuing the current request. The additional information piggybacked with the token in a so-called kill list serves the purpose of releasing locally stored requests after they have been served.

12.3 Complexity and Correctness

The prioritized protocol still has an average message complexity of $O(log\ n)$ and a worst-case complexity of $O(n)$. If the protocol was exposed only to uniform priority values, it would resemble an unprioritized protocol with reduced message passing overhead, *i.e.*, it would result in fewer messages than Naimi's approach due to its local queuing. The correctness of the prioritized protocol is shown as follows. Mutual exclusion is guaranteed by ensuring that only one node may hold the token at any time. Deadlocks are prevented since the probable owner chain always leads to the token or a higher priority requester. Once a request is queued, this request will be seen by the current or future token holder without delay (other than token usage) since circular request and token propagations are prevented. The protocol does not impose starvation since there can only be a finite number of requests ahead in the queue at one priority level whose number is monotonically decreasing once new requests are subject to the relative fairness constraint. Furthermore, relative fairness also ensures that requests be served in a strict priority ordering.

12.4 Quantitative Results

Experiments where conducted to compare Naimi's unprioritized algorithm to the prioritized protocol. In addition, the overall performance of the prioritized protocol was measured. The measurements where obtained by implementing the protocols within the distributed execution environment DSM-Threads. The experiment was designed to measure the number of messages exchanged per request and the response time between request issue and token reception. Three experiments where conducted.

First, eight threads were competing to enter critical sections on a uniprocessor simulating communication by local synchronization, which simulated physically shared memory and revealed the computational overhead of the prioritized protocol. The results indicate that Naimi's approach has about one third of the computational complexity of the prioritized protocol. This indicates the cost required for the additional functionality of priority support of processing larger messages.

Second, eight processes were competing on a dual shared-memory processor using TCP for communication, which simulated a medium-speed network. The results showed a reduced message overhead of the prioritized approach over Naimi's protocol reflecting the savings due to local queuing for frequent requests. The response time of Naimi's protocol increased to about three forth of the response time of our approach.

Third, eight processes were distributed over distinct nodes communicating via TCP over slow Ethernet, which resembles a low-speed network. Here, almost the same number of messages were observed for both protocols. The response time of Naimi's solution varied between half and two thirds of the response time of the prioritized protocol for higher and lower request intervals, respectively.

In summary, the prioritized approach results in the same or slightly lower message overhead than Naimi's approach but incurs additional computation overhead resulting in about 30% longer response times. Therefore, it should be utilized when strict priority ordering of requests is required in a distributed environment.

12.5 Preventing Priority Inversion

Recall the problem of priority inversion from Chapter 1. A medium priority task could execute until completion while a higher priority task waits for a resource held by a lower priority task. The approaches described by the priority-ceiling and priority-inheritance protocols were introduced to counter this problem.

In a distributed environment, the problem of priority inversion also extends to prioritized message passing. The mutual exclusion protocol developed in this work can thus be subject to long blocking periods. Hence, extensions to the protocol to support priority ceiling and priority inheritance are provided.

Priority-ceiling emulation, an efficient variant of the dynamic priority-ceiling protocol, is realized as a straight-forward extension of the base protocol. Ceiling values of resources are calculated statically as the maximum priority of any task potentially requesting this resource. Upon issuing a lock request and receiving the token, the effective priority is boosted to the maximum of the base priority and the ceiling of the new lock. Upon releasing the lock, the priority is lowered to the maximum of the base priority and the ceilings of all locks still held thereafter. These enhancement provides full support for ceiling emulation with the general advantages and disadvantages of the approach, *i.e.*, while preventing chained blocking, avoiding priority changes in critical section and reducing the number of context switches, false blocking may be incurred where the blocking task has no relation to the blocked victim. These properties are generally known and represent a trade-off between efficiency and functionality. Overall, the ceiling extensions do not require any additional messages while avoiding priority inversion for static priority assignments in distributed systems.

The second approach supports priority inheritance by allowing dynamic priority changes. Upon contention for a resource, the priority of the task in a critical section may be boosted to the level of the requesting task for the remaining execution within the critical section, thereby preventing priority inversion. The prioritized message protocol for mutual exclusion can be extended as follows. Requests are re-issued at the higher, inherited priority. The request then travels along the same propagation path again, and its logs are updated with respect to the higher priority. This causes the request to reach the token holder more quickly and to be queued closer to the head of the queue, *i.e.*, the higher priority ensures that the token be received more quickly.

Figure 12.4 depicts an example. Initially, B enters the critical section of $L2$, the token

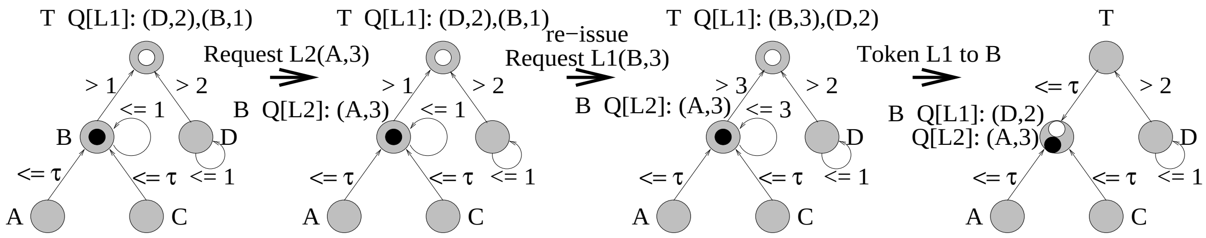


Figure 12.4: Distributed Priority Inheritance

(depicted as a black circle) is present at B. Then, requests for lock $L1$ from B and D are issued at priorities 1 and 2, respectively. The edges and weights of the first picture represent the prioritized tree structure for lock $L1$, whose token (depicted as a white circle) is present in node T. A new request from A for $L2$ at priority 3 propagates to B, the current token holder. Since B holds $L2$ and has requested $L1$ at a lower priority, the request from B for $L1$ is re-issued at priority 3 and propagates to T. This results in a changed order of the queue for $L1$ in T. When T releases the token, it is sent to the highest priority requester, namely B. Hence, B receives the token before D due to priority boosting by inheritance although its original request was at a lower priority.

The details of the protocol changes include priority adjustments upon reception of a re-issued request, priority boosting of tasks engaged in a critical section and lowering of priorities upon leaving the critical section. The additional message overhead depends on the blocking circumstances. Priority boosting may involve transitive inheritance also known as chained blocking. In the absence of chained blocking, only re-issue overhead is incurred with the general message complexity of requests, *i.e.*, $O(log\ n)$ on the average and $O(n)$ in the worst case. In the presence of chained blocking, the worst-case can be bounded tightly by $O(d * n)$, where d denotes the distance on the resource dependency graph from the current resource to the root, *i.e.*, the height indicates the number of transitive inheritance steps that may potentially by caused.

In summary, a protocol for distributed mutual exclusion with priority support has been formulated that supports nested critical sections. Its message complexity of $O(log\ n)$ on the average and $O(n)$ in the worst case matches that of the best known unprioritized counterparts. It has been proved correct and shown to preserve message ordering under the property of relative fairness, which has also been formulated. A quantitative analysis indicates an overhead of 30% longer response times imposed by the support of additional functionality, namely priority support, when compared to Naimi's protocol. Extensions to prevent priority inversion have been developed, which either incur no message overhead in the case of priority-ceiling emulation or impose the overhead of request re-issuing for priority inheritance. This provides the means to incorporate priorities into distributed synchronization protocols, which, amongst other applications, may be used to support distributed real-time scheduling.

Chapter 13

Future Work

Schedulability analysis for shared-memory multiprocessors and distributed systems has not reached the maturity of its uniprocessor counterpart. It still remains to be seen which approaches will become widely accepted, and there may be more than one approach, just as in uniprocessor schedulability theory. The multiprocessor priority-ceiling protocol seems best suited for SMPs but constrains resource allocation to be static. End-to-end analysis is applicable to distributed systems as well as SMPs and provides a convincingly simple analysis model based on the response time of execution phases. Distributed resource protocols, as promoted by our work, provide a decentralized approach, allow dynamic resource allocation and are based on the results of uniprocessor schedulability theory. Future work should focus on the analysis of blocking times for distributed resource protocols. In addition, a mixture of two approaches may also be suitable to distinguish static and dynamic allocation of resources. In such a case, the interaction between these models has to be formalized.

Another area of future work involves an investigation into the suitability of DSM-Threads for real-time environments. The use of distributed shared memory to support the traditionally well understood programming paradigm of shared memory in a distributed environment provides an attractive alternative to programming through explicit message passing. However, page-based DSM systems that utilize virtual memory protection are subject to page faults. First, hard real-time systems often do not utilize virtual memory but use physical memory mappings instead. Hence, DSM-Threads may be more suitable for soft real-time systems where virtual memory is utilized. Second, it remains to be seen how the temporal effects of page faults can be taken into account in a real-time environment.

Furthermore, the token-based protocol for distributed mutual exclusion may be extended to support fault-tolerant behavior. A preliminary study indicates that node failures may be compensated by a fault-tolerant extension of the protocol. The combination of fault tolerance and support for deterministic temporal behavior could thereby provide the property of responsiveness in a distributed real-time system.

Finally, experiments on timing analysis should be conducted to evaluate the merits of our approach within distributed systems. Our distributed priority protocol should be combined with the methods of timing analysis for uniprocessors and schedulability analysis for distributed systems to demonstrate the feasibility of the overall approach.

Chapter 14
Summary

Schedulability theory for distributed systems enhances models for uniprocessors to a parallel environment. The results of uniprocessor schedulability analysis are still applicable to a parallel environment in the sense that schedulability may be determined separately for each CPU under the assumption that nodes are independent. In order to incorporate interdependencies between nodes, common resources, such as the communication medium, may also be treated as a special processor where uniprocessor schedulability theory is applied to analyze the temporal behavior of messages. To cover temporal effects due to remote blocking, the synchronization between different nodes of execution is incorporated into schedulability analysis. Several approaches exist to address these issues. The multiprocessor priority-ceiling protocol assumes that resources are statically assigned to CPUs and that shared memory physically exists. End-to-end schedulability analysis, *e.g.*, through the release-guard protocol, does not require shared memory, which makes it suitable for distributed real-time environments. These protocols require proxy execution for remote resource requests. Distributed resource protocols allow resources to be dynamically re-assigned to nodes and do not require shared memory. Our token-based prioritized protocol for mutual exclusion is an example of such a protocol. This protocol is defined, shown to be correct and experimentally evaluated. In addition, extensions to support priority ceiling and priority inheritance for real-time systems are provided for this protocol. The quantitative analysis of the protocol is based on an implementation over DSM-Threads, a distributed runtime system developed by the author.

DSM-Threads provides an environment for distributed execution and ensures memory consistency in a distributed system through distributed (virtual) shared memory. This allows the user to exploit the shared memory programming paradigm in a distributed environment. DSM-Threads distinguishes itself from other DSM systems by its support for per-node multi-threading, asynchronous communication, predictable response characteristics in terms of temporal behavior, a close resemblance of standardized APIs for parallel execution, support of different memory consistency models, and decentralized synchronization protocols.

The token-based prioritized protocol for mutual exclusion has been implemented in DSM-Threads to demonstrate its applicability to distributed environments and to provide an environment for quantitative evaluation. The protocol enforces strict prioritized message handling and FIFO handling within one priority level. It guarantees ordered request handling if requests comply with the property of relative fairness, *i.e.*, if two requests are known to be issued a certain number of time units apart. Performance results indicate that the protocol exhibits 30% longer response times than an unprioritized protocol, *i.e.*, the expense of priority support is quantified. The protocol is shown to be correct and extensions are provided to prevent priority inversion. In particular, extensions for the priority-ceiling emulation protocol and the priority-inheritance protocol are provided. Similar extensions for the dynamic priority ceiling protocol require an approach similar to the priority-inheritance extensions. These protocols provide support for real-time systems in a distributed environment. They also allow conventional schedulability analysis for uniprocessors to be applied to each component of a distributed system. The blocking times then have to reflect the deterministic upper bounds on communication required by the resource allocation protocols, such as our token-based protocol.

Conclusions of Parts I and II

This work provides a comprehensive study of the state of the art in schedulability theory, worst-case execution analysis, and synchronization support for real-time systems.

First, the results of schedulability theory for uniprocessor real-time systems and embedded systems are summarized. These results are based on the assumption that the worst-case execution time (WCET) of a task be known a priori. A framework for WCET analysis is developed in the following. The framework is formally derived, an implementation is provided and results on timing analysis for uniprocessors are presented. It is shown that a component-based approach separating the process into several steps of the analysis based on the modeled entities results in an efficient and practical solution of the problem. Then, an introduction to the overall toolset for static timing analysis is given followed by a more detailed derivation of methods for performing the static cache analysis. For this purpose, a data-flow framework is developed that provides classifications of cache references suitable for the static prediction of the WCET. The integration of these results into pipeline simulation and path analysis provides cycle-level timing predictions for real-time tasks in a uniprocessor environment. Future work includes a task analyzer to extract task boundaries during compilation automatically, which would complete this framework for uniprocessor timing analysis.

Second, the problem of transferring the results of timing analysis for uniprocessors to distributed systems is discussed. Key services of distributed systems are identified that require additional protocols to provide guarantees for real-time systems. An environment of distributed execution with support for memory consistency, DSM-Threads, is introduced that is utilized in the following to evaluate distributed synchronization protocols. Next, a novel distributed protocol to arbitrate resource requests is introduced that provides a decentralized solution to the mutual exclusion problem while supporting system-wide priorities. Protocol extensions prevent priority inversion by means of priority ceiling and inheritance in a distributed environment. Finally, open problems for distributed real-time systems are identified and a road map for future work on schedulability analysis and synchronization using static WCET prediction is presented.

In summary, our unique contributions are threefold: First, we provide a general framework for static timing analysis with compiler integration and support for the analysis of set-associative caches, integration into pipeline simulation and path anlysis. Second, we compare this approach to dynamic timing analysis and derive usage guidelines. Third, we develop a novel distributed synchronization protocol with fairness properties, show its correctness and evaluate its performance in a distributed runtime system that we also developed. Tools for timing analysis of uniprocessor tasks can be applied in the context of schedulability analysis of distributed real-time systems through the support of our distributed synchronization protocol.

References

[1] N. Audsley, A. Burns, M. Richardson, and A. Wellings. STRESS: A simulator for hard real-time systems. *Software Practice & Experience*, 24(6), June 1994.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[3] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.

[4] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *COMPCON Conference Proceedings*, pages 528–537, 1993.

[5] R. Bettati and J. W.-S. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *12th International Conference on Distributed Computing Systems*, pages 452–459, Washington, D.C., USA, June 1992. IEEE Computer Society Press.

[6] A. Burns, B. Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Ada Europe*, pages 263–275, 1998.

[7] A. Burns and A. Wellings. Synchronous sessions and fixed priority scheduling. *Journal of Systems Architecture*, 44:107–118, 1997.

[8] J. V. Busquets-Matraix. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 204–212, June 1996.

[9] J. V. Busquets-Matraix. Hybrid instruction cache partitioning for preemptive real-time systems. In *EuroMicro Workshop on Real-Time Systems*, pages 56–63, June 1997.

[10] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[11] Y. Chang. A simulation study on distributed mutual exclusion. *J. Parallel Distrib. Comput.*, 33(2):107–121, March 1996.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[13] M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, September 1993.

[14] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3):131–181, November 1999.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[16] F. Gasperoni. Embedded opportunities. In *Ada Europe*, pages 1–13, June 1998.

[17] E.W. Giering, Frank Mueller, and T.P. Baker. Implementing Ada 9x features using POSIX threads: Design issues. In *TRI-Ada '93 Proceedings*, pages 214–228. ACM, September 1993.

[18] T. Hand. Real-time systems need predictability. *Computer Design (RISC Supplement)*, pages 57–59, August 1989.

[19] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[20] C. A. Healy, M . Sjödin, and D. B. Whalley. Bounding loop iterations for timing analysis. In *IEEE Real-Time Technology and Applications Symposium*, pages 12–21, June 1998.

[21] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[22] M. Humphrey and J. A. Stankovic. CAISARTS: A tool for real-time scheduling assistance. In *IEEE Real-Time Technology and Applications Symposium*, June 1996.

[23] Y. Hur, Y. H. Bae, S.-S. Lim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of RISC processors: R3000/R3010 case study. In *IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.

[24] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. In *Symposium on Parallel Algorithms and Architectures*, pages 277–287, June 1996.

[25] ISO/IEC. *ISO/IEC 8652L 1995 (E), Information Technology – Programming Languages – Ada*, February 1995.

[26] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. *J. Parallel Distrib. Comput.*, 29:126–141, September 1995.

[27] Y. Kermarrec and L. Pautet. Gnatdist: A configuration language for distributed Ada 95. In *TRI-Ada*, 1996.

[28] Y. Kermarrec, L. Pautet, and S. Tardieu. Garlic: Generic Ada reusable library for interpartition communication. In *TRI-Ada*, pages 263–269, 1995.

[29] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Technology and Applications Symposium*, June 1996.

[30] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Real-Time Systems Symposium*, pages 229–237, December 1989.

[31] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Giude to Rate Monotonic Analysis for Real-Time Systems*. Kluwer, 1993.

[32] L. Lamport. Time, clocks and ordering of events in distributed systems. *Comm. ACM*, 21(7):558–565, June 1978.

[33] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *IEEE Real-Time Systems Symposium*, December 1996.

[34] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems*, 7(4):321–359, November 1989.

[35] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–397, December 1995.

[36] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[37] J. Liedke, H. Härtig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 213–223, June 1997.

[38] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[39] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[40] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[41] J. W. S. Liu, J. L. Redondo, Z. Deng, T. S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W. K. Shih. PERTS: A prototyping environment for real-time systems. In Susan Davidson and Insup Lee, editors, *Proceedings of the Real-Time Systems Symposium*, pages 184–188, Raleigh-Durham, NC, December 1993. IEEE Computer Society Press.

[42] T. Lundquvist and P. Stenström. An integrated path and timing analysis method based on cycle-level sympolib execution. *Real-Time Systems*, 17(2/3):183–208, November 1999.

[43] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.

[44] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 137–145, June 1995.

[45] F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).

[46] F. Mueller. Supporting Ada 95 passive partitions in a distributed environment. In *Reliable Software Technologies – Ada-Europe '97*, pages 218–229. Springer LNCS 1251, June 1997.

[47] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, June 1997.

[48] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, 1998.

[49] F. Mueller. Adaptive dsm-runtime behavior via speculative data distribution. In J. Jose et al., editor, *Parallel and Distributed Processing – Workshop on Run-Time Systems for Parallel Programming*, volume 1586 of *Lecture Notes in Computer Science*, pages 553–567. Springer, April 1999.

[50] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 340–349, December 1999.

[51] F. Mueller. Decentralized synchronization for multi-threaded dsms. In *Workshop on Software Distributed Shared Memory*, page (accepted), May 2000.

[52] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[53] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, April 1996.

[54] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[55] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[56] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, pages 63–79, Summer 1996.

[57] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–269, December 1988.

[58] T. Röblitz and F. Mueller. Combining multi-threading with asynchronous communication: A case study with dsm-threads using myrinet via bip and madeleine. In *Myrinet User Group Conference*, page (accepted), September 2000.

[59] D. Simpson. Real-time RISCS. *Systems Integration*, pages 35–38, July 1989.

[60] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., 2.7.2 edition, November 1995.

[61] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems.* Kluwer, 1998.

[62] A. Stoyenko. A schulability analyzer for real-time euclid. In *IEEE Real-Time Systems Symposium*, pages 218–227, December 1987.

[63] J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. In *Int. Conference on Distributed Computing Systems*, pages 38–45, May 1996.

[64] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].

[65] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):149–171, May 2000.

[66] Hideyuki Tokuda. Arts real-time scheduler analyzer/debugger. *IEEE Software*, 5(3):29, May 1990. [Abstract].

[67] New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.

[68] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, page (accepted), 2001.

[69] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, November 1999.

[70] A. Wolfe. Software-based cache partitioning for real-time applications. In *Workshop on Responsive Computer Systems*, 1993.