An Overview of Practical Research Approaches to Real-Time System Engineering

Lars Albertsson
Computer and Network Architectures Laboratory
Swedish Institute of Computer Science
lalle@sics.se

September 1, 2001

1 Introduction

Real-time applications executing in general-purpose computing environments have become popular. Examples of such applications are games, support software for telecommunication systems, and encoders and decoders of streaming media. The applications must provide throughput and response time guarantees. They should also provide high average throughput, and must therefore execute on commodity high-performance platforms.

It is inherently difficult to construct computer systems providing performance or response time guarantees. The difficulty is compounded by the focus in computer system design; optimising common execution paths, thereby achieving good average, but also unpredictable computing performance. The variations in execution time may be of several orders of magnitude, and it is hard to predict the worst case execution time.

The real-time research community is predominated by theoretical approaches to real-time system construction. Analytical models of both hardware and software components are combined to form a proof that the modelled system will always perform its task within stipulated time. Even though theoretical methods are popular and in some cases effective, they have a number of drawbacks:

- Theoretical methods rely on a good model of the system under study. As a model is an approximation of reality, it is not exact and always contains approximation errors. Proofs obtained can only be as solid as the accuracy of the underlying model.

- In order to reason about worst case execution times, it is often necessary to use very pessimistic performance models. Hence, a real-time program cannot both be proven to be correct and use the full performance provided by modern hardware.

- Proving properties of theoretical models is a task with very high computational complexity. Analytical methods are therefore limited to analysing simple systems, or to using simple models of complicated systems.
Because of these properties, theoretical methods are effective for designing small embedded systems, and less effective for real-time system design based on the more complex general-purpose systems. Successful construction of real-time applications in commodity environments therefore requires different approaches. This survey is an attempt to summarise practical methods for designing, constructing, and evaluating real-time systems. Practical design methods for real-time systems are discussed in Section 2. Methods for monitoring running systems are described in Section 3. Section 4 describes methods for predicting properties of designed systems, focusing on simulation methods.

2 Design methods

Real-time system design is more complex than traditional system design, as components cannot be regarded in isolation. Every active component affects execution time, and the system must either be analysed in concert, or partitioned in the time domain.

2.1 Programming support

There are design tools that provide a language and environment for expressing temporal constraints during design as well as support for verification or validation. The tools cannot guarantee that the application designed is correct unless the entire system is designed with a single tool, which is impractical for large systems. Instead, the purpose of such tools is usually to support automated checking of real-time properties, using either monitoring techniques, described in Section 3, or prediction techniques, described in Section 4.

Design tools that support automated monitoring allows the programmer to express temporal annotations, either with extensions to existing programming languages [KL91, Ger96], or with domain-specific languages [PW93, BJHL96]. The compiler uses the annotations to automate monitoring or execution time prediction.

The PERTS [LRD93] system is an example of a prediction-based design tool. The programmer builds prototypes, and DRTSS [SL94], a component of PERTS, helps the programmer validate the prototypes. DRTSS consists of an environment for discrete event simulation, enabling the user to validate a coarse model of his system. There are also many other similar tools, and their usefulness is limited to small systems, where the model corresponds well to the real system.

2.2 Operating system support

It is possible to construct operating systems that control application execution time by partitioning computing resources. For a very long time, the research community has striven to construct commodity operating systems that support controlled partitioning in the time domain. This is often referred to as providing quality of service (QoS) guarantees. Despite vast efforts, little progress has been made.
2.2.1 Real-time schedulers

The most obvious way of improving real-time support is to replace the priority-based scheduler that is found in general-purpose operating systems. Several schedulers have been tried in different operating systems [Gol94, Sri98], and some have made their way into commercial systems [KSZ91, HP97]. A good description of a real-time scheduler and implementation issues is given by Khanna et al. [KSZ91].

Researchers in the KURT project at Kansas University have evaluated a family of schedulers for Linux [Sri98]. In order to provide accurate timing, the Linux system timer has been replaced with a high resolution timer. They have also discovered that scheduling is disturbed by interrupt blocking due to disk activity, and have attempted to address this [Hil98].

2.2.2 Microkernels

Unfortunately, simply adding a real-time scheduler to an operating system does not render sufficient QoS guarantees. In a monolithic kernel, many activities are performed that are not possible to map to a thread. Thus, the kernel cannot distinguish between activities that should have different priorities.

In order to give appropriate service to prioritised applications, there are designs that move kernel services to user space, where they may be scheduled as desired. An example of this approach is real-time upcalls [GP98], where applications may provide network protocol processing code, which is called by the kernel when a packet arrives. This method requires that the kernel is able to map network packets to applications at an early stage.

A similar design is often used in novel operating systems designed to support soft real-time applications. The Nemesis [LMB06] operating system provides fair and predictable execution by placing operating system services in libraries executed by the application. In Scout [MMO94], processing is mapped to data paths, which are allocated resources and are scheduled similarly to threads. While new operating systems are interesting from an academic point of view, immaturity and lack of user base have prevented them from becoming widespread.

The RTLinux project [Bar97] and the RTAI project [MBDP00] combine a general-purpose operating system, Linux, with a microkernel supporting real-time services. Real-time programs are scheduled by the microkernel, and therefore get sufficient quality of service. The Linux kernel runs as a low-priority task on the microkernel. The microkernel provides communication channels between Linux applications and the real-time applications.

2.2.3 Resource partitioning

Most of the research in real-time operating system support is focused on how to schedule the CPU. This is not sufficient, however, as applications also compete for other resources. Banga et al. [BDM99] addresses this problem by introducing resource containers. When an application requests a service from the kernel, the resources used are charged to a resource container, which has been
allocated by the application earlier. This allows the kernel to control fairness and predictability with respect to different types of resources.

Waddington and Hutchinson present a design for Windows NT which they call protected virtual machine [WH99]. They partition the resources in several virtual machines, which are scheduled individually. Each thread uses the resources of its corresponding virtual machine, in a similar manner as with resource containers.

Rajkumar et al. provide a similar design called resource kernels [RJMO97], which addresses the problem that there are dependencies between scheduling of different resources. Oikawa and Rajkumar have also implemented a portable resource kernel framework, which has been ported to multiple operating systems [OR99].

Hardware controlled resources, such as caches, are more difficult to partition. Liedtke et al. present a technique for partitioning hardware caches by controlling the virtual memory mapping [LHH97]. They modify the operating system to map virtual memory pages accessed by different processes to different parts of the cache memory. This method only works for physically indexed caches.

Manual partitioning of multiple resources can be difficult. This is addressed by Abdelzaher, who proposes an automated mechanism for determining application reservations based on estimation theory and profiling of running systems [Abd00].

References


[LMB+96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.


3 Monitoring real-time systems

Monitoring is another task that becomes more problematic when applied to real-time systems. In order to measure the state of a running system, it is necessary to insert a probing mechanism, which unfortunately always changes the behaviour of the system. This is referred to as the probe effect [Gai85]. In many cases, the only probe effect observable by an application is a change in execution time. This is acceptable for monitoring of traditional applications, as the correctness of most traditional, sequential programs is unaffected by such perturbation. In contrast, the correctness of real-time programs depends on time elapsed between different points in program execution. With a careful monitor implementation, the probe effect may be very small, and unless the application under study is very fragile, useful results can still be obtained. Nevertheless, probe effect effectively limits the amount of monitoring.

The second major problem with monitoring of real-time systems is the lack of reproducibility. For traditional programs, it is feasible to capture and replay the relevant interactions between an application and its environment. As the
timing of both program execution and the actions of its environment is inherently unpredictable, the time flow of a real-time application cannot be easily captured and replayed. Moreover, if the monitoring configuration changes between experiments, the probe effect will change program behaviour. In this case, even minimal probe effects may result in widely different execution paths.

Lack of reproducibility is also a problem for non-real-time application whose execution is dependent on the time flow. The most common example of such applications is multi-threaded programs, which can produce different results depending on interleavings of the executing threads [YP97]. The parallel processing community has therefore performed research on low-intrusion and reproducible monitoring, replay and debugging. Some of this research is mentioned below.

3.1 Software-based monitoring

The most straightforward method of monitoring computer systems is to insert extra program code to track and log application progress. The extra code can be inserted by the programmer [Sch88, NGM98], preprocessor [GK96], compiler, binary program modification, run-time system [CL95, ML97], operating system [MMS86, TM99], or by an external monitoring program [MCC+94]. In order to obtain accurate information, it is desirable to keep intrusion from the probing mechanism low. In real-time systems, the monitoring software is often defined as part of the production system in order to avoid the probe effect. The problems regarding intrusive monitoring and related research has been further described by Marinescu [MLC+90].

Snodgrass proposes that monitoring should be regarded as queries to a fictional database containing information on program execution [Sno88]. He presents a monitoring system with automatic generation of probing code, based on database queries.

Hollingsworth [Hol94] describes a technique for minimising intrusion in Parodyn [MCC+94], a performance instrumentation tool for parallel programs. Parodyn has the ability to adapt instrumentation dynamically to focus on different performance bottlenecks. The dynamic instrumentation is used to implement an automatic search for bottlenecks. The search is performed along three independent axes: time, program position and cause. The tool adapts the instrumentation to probe for one hypothetical bottleneck at a time. Karanovic and Miller [KM99] propose the use of performance data from previous executions to guide the search for bottlenecks.

3.2 Hardware-based monitoring

In order to avoid intrusion from extra probing software, some researchers have applied hardware solutions for monitoring [Pla84, TFC90, MLC+90, TG92, DR82]. The probing is usually performed by eavesdropping memory traffic or communication between nodes. Some of the hardware-based monitoring implementations are actually intrusive, as they generate extra interrupts or bus contention. Hardware-based methods are expensive, as they require extra hardware and generate large volumes of data. Furthermore, only a fraction of a
program's operations can be traced, and for example memory transactions hitting in a processor's cache cannot be observed by other components.

3.3 Debugging tools

As developers are familiar with symbolic debuggers, it is desirable to present monitor output data by mapping them to a debugger. The debugger can either operate on a trace from an earlier execution [TFCB90, TG92, TBY96] or probe a running system [R2D]. Both methods suffer from the standard problems in monitoring real-time programs: debug sessions are not reproducible and probing affects program execution. These are significant problems, as debugging tends to be performed cyclically; the programmer notices a problem and restarts the debugging session to recreate the problem and examine its cause. This is generally not possible for real-time application debugging, as a problem cannot be reproduced predictably, and the information recorded in a particular trace may be inadequate to debug the problem.

Mueller and Whalley present a debugger for real-time applications, which is complemented by a cache simulator, which predicts program execution time [MW94]. The debugger operates on a running program, and the cache simulator estimates execution time by inspecting memory references. This approach ignores time spent executing the operating system and only works for programs whose execution flow is independent of elapsed time.

Glass discussed and summarised the issues of debugging and testing real-time programs in an article in 1980 [Gla80]. Unfortunately, little progress has been made since then, and most of his conclusions are still applicable.

McDowell and Helmbold discuss the problems of debugging concurrent programs thoroughly, and present a summary of existing debugging techniques [MH89].

3.4 Deterministic replay

In order to provide reproducible debugging sessions for real-time and multi-threaded applications, a technique called deterministic replay has been proposed. A monitoring system collects information on application input and events driven by the clock, such as interrupts and scheduling decisions. When the system is executed in a debugger, the input is taken from the recorded trace, and all clock-based events are replaced with the events recorded in the trace. The timing information and interleavings of events in the original execution are thereby recreated.

The monitoring and replay system can be implemented in the operating system [TH00], run-time system [LMC87, TCO91], or by using hardware support [TFCB90].

References

[CL95] R. Cypher and E. Leu. Efficient race detection for message-passing programs with nonblocking sends and receives. In Symposium on
Parallel and Distributed Processing (SPDP '95), pages 534–543, Los Alamitos, Ca., USA, October 1995. IEEE Computer Society Press.


4 Predicting system behaviour

There are two ways of predicting the execution time of computer programs: analysing a program statically or simulating the execution of a program. Static analysis methods aim at estimating the maximum and minimum execution time for sections of code. While the information obtained is desirable, the analysis does not scale well to complex systems. The simulation approach scales better, but is limited to analysing one execution path at a time. As with any testing method, it also requires the developer to provide representative input data.

4.1 Execution time prediction

The major obstacle for predicting execution time on modern hardware is the use of caches. Caches improve overall throughput performance, but execution time prediction becomes very hard. A memory access operation resulting in a cache miss slows down the access operation by orders of magnitude. Thus, unless cache contents are known, predicted worst case execution time will be much higher than the average case. For programs with small data sets, however, it is feasible to predict cache contents, thus decreasing worst case execution time estimation significantly.

Whalley et al. uses a technique called static cache simulation [MW94]. The control flow of a program is analysed and fed to a cache system simulator. Prior to executing the program, a simulation of the cache is made. It turns out that a majority of the results of cache lookups may be accurately predicted in advance. Thus, it is possible to predict worst-case and best-case execution times with respect to memory hazards. This method works best for direct mapped instruction caches, but work has also been made on set associative caches [WMH+97], data caches [WMH+97], and multi-level caches [Mue97].

The Cinderella tool [LMW95] estimates execution time using formal methods. It transforms the flow analysis and cache content constraints to integer linear programming problems, which are fed to an exterior solver. In order to improve scalability, there is support for programmer annotations regarding possible execution paths. Nevertheless, the tool is still limited to small programs and instruction cache models.
4.2 Computer system simulation

The usefulness of computer system simulation was recognised long ago. Several simulators of popular systems were constructed during the 80’s. Some were built for debugging purposes [HHL84], whereas other supported performance instrumentation as well [CFH+79, DM84].

Program execution can be simulated on many different abstraction levels. The usefulness of a simulator is limited by the accuracy of the model it provides. This discussion focuses on simulators that model a computer at the instruction level. As the instruction level is the border between hardware and software, they execute binary programs with little or no modification. This limits the sources of errors to those introduced by the machine model, and by models of simulation input.

As hardware became more complex, the performance of instruction level simulators decreased, limiting the size of systems that may be studied. Although the processors that are used to run the simulators have gotten faster rapidly, the complexity of software has increased almost as fast. Thus, execution time of programs has decreased only slowly. It is still possible to accurately simulate less complex embedded processors, using modern workstations. Several vendors of embedded processors also provide cycle-accurate simulators [And94, Win]. High-performance embedded processors can be simulated with special hardware support, increasing simulation performance [Lau].

There is a class of programs called emulators, which are similar to simulators.\footnote{There is no consensus on the distinction between emulator and simulator, and other sources may disagree with the definition presented here.} The purpose of an emulator is to run programs in alien environments, such as a different processor or operating system, whereas simulators are used for hardware development, debugging and performance evaluation. Emulation shares many properties with simulation, for example implementation techniques. Emulators are not useful for analysing real-time programs, however, and will not be discussed further here.

4.2.1 Implementation techniques

Naïve implementations of instruction set simulators are too slow to execute workloads of realistic size. In order to overcome the problem, techniques for more efficient simulation have been explored.

Target to host instruction mapping, also called direct execution, is a commonly used trick for improving simulation performance. If the target and host architectures are compatible, many instructions need not be translated. In the common case, it is sufficient to insert instrumentation code and change address values where appropriate. This technique limits portability and is inappropriate when simulating non-general-purpose architectures.

Bedichek introduced an intricate technique called threaded code: the simulator generates a code snippet for each instruction instead of using a common decoding routine. This technique minimises simulation overhead and reduces the number of branches, and therefore improves performance. The technique was implemented in two simulators [Bed90, Bed95].
Several researchers have explored the use of trace-driven simulation for analysis of parallel programs. Traces are usually generated by intercepting library calls or preprocessing the application in order to insert logging within the program. The traces obtained are fed to a simulator, which is used to detect serial bottlenecks, evaluate modifications to hardware or software, or to predict parallelisation speedup.

The MPTrace tool [EKKL90] operates by modifying binaries to generate an execution trace. ATOM [Fra96] is capable of modifying and tracing programs as well as most of the operating system. The Shade tool [CK94] also aims at trace generation, but uses dynamic translation. The Shade paper includes a presentation of implementation techniques as well as a good overview of related research in computer system simulation.

Researchers at Massachusetts Institute of Technology built a simulator called PROTEUS [BDCW92], aimed at parallel program performance analysis. It was extended by researchers at University of Southern California to provide more accurate models, including models of virtual memory [PS90].

4.2.2 Complete system simulation

The tools mentioned above simulate parts of a complete computer system. Most of them model only the CPU and cache memory system. In order to avoid modelling of peripheral units, they simulate user level execution only. Calls to the operating system are passed unmodified to the host operating system. Ignoring operating system execution, however, may have severe impact on the accuracy of instrumentation, as demonstrated by Casmira et al. [CFKM98].

In order to minimise deviation between the real system and the simulator, some groups have implemented tools simulating complete hardware systems. These simulators model the hardware in enough detail to run unmodified programs, including operating systems. The tools are referred to as complete system [RHWG95], full system [MDG+98] or faithful [DM84] simulators. Although these simulators provide an almost exact functional model of the hardware, they use approximative timing models. This is necessary in order to achieve reasonable simulation performance. The timing models are focused on the devices known to be performance bottlenecks in modern computer systems, for example the memory hierarchy and I/O devices.

The first complete system simulator was presented by Doyle and Mandelberg [DM84]. It simulated a PDP-11 and was able to boot an unmodified Unix distribution. It provided some basic instrumentation and executed approximately 120 times slower than the host machine.

As the hardware complexity increased, it became more difficult to build complete system simulators providing useful timing models. In 1998, Magnusson et al. presented SimICS, a simulated multiprocessor Sparcstation running Solaris or Linux [MDG+98]. The simulator supports efficient programming of timing models of peripherals and cache hierarchies, including multiprocessor coherence protocols. The simulator is implemented using techniques described by Bedicheck [Bed90] and Magnusson et al. [Mag93, MW95, Mag97]. It executes at an approximate slowdown of 100.
The SimOS group at Stanford University has built a machine simulator providing enough detail to run operating systems, requiring only small operating system modifications [Her98]. The simulator provides detailed instrumentation as well as support for programming event filters. It is also possible to program detailed timing analysis by associating scripts with events. The simulator supports switching between multiple timing models, thereby trading simulation performance for accuracy. The achievements of SimOS are similar to those of SimICS, although the simulator designs are different [RHWG95]. The SimOS group has taken shortcuts by mapping hardware events to services in the underlying operating system. Although the simulator is not strictly faithful because of these shortcuts, it has proved useful and accurate for simulating large systems nevertheless [RBDH97]. Gibson et al. have validated the SimOS simulation models to performance measurements on real machines [GKO+00].

References


