A Top Layer Design Approach to Complex Real-Time Software

Jair Jehuda

RAFAEL Electronic Systems Division (80)
P.O.B. 2250, Haifa 31021, ISRAEL
jehuda@tx.technion.ac.il

Daniel M. Berry

Department of Computer Science
Technion - Israel Institute of Technology
Technion City, Haifa 32000, ISRAEL
dberry@cs.technion.ac.il

January, 1995

Abstract

In this thesis we introduce a *top-layer* design approach to *complex* real-time software, so called because it is designed to produce and support *platform-independent* software for extremely complex system requirements, e.g., concurrent, best-effort hard real-time systems running on shared-memory multi-processor platforms.

The keys to our approach are simple, yet flexible platform and job-oriented program models that enable decomposing larger, generally highly complex, scheduling problems into several smaller, generally simpler, scheduling problems, called *jobs*. Internal job scheduling problems are independently resolved offline and encapsulated in a platform-independent manner, creating reusable real-time software objects. A hard real-time *job time-sharing* scheme is then used to incorporate these jobs into programs without having to deal with the scheduling details within them. An additive schedulablility criteria suffices for determining whether a given set of jobs can reliably time-share a given processor, thereby greatly reducing the complexity of *load-balancing* over available platform resources.

To best utilize any given platform, each job may also support several *modes* of operation, each mode requiring a different measure

of resources and contributing a different reward to an application-specific system value function. Job mode alternatives, required mode resources, and contributed rewards may also be state-dependent, varying with computational circumstances. An appropriate software *meta-control* algorithm must therefore occasionally recompute the job modes which maximize the system value per time unit for evolving runtime circumstances without overloading the system resources, i.e. without jeopardizing any hard deadlines within any of the jobs on any of the platform processors. This multi-mode job-oriented program model thus introduces a novel notion of *best-effort* meta-control in a hard real-time context. The same model can also be used to enable several *concurrent* hard real-time applications to reliably share appropriate multi-processor platforms.

Appropriate algorithms have been devised, several policy making issues have been addressed, and all of the above have been incorporating within a comprehensive top-layer architecture. A realistically complex musical real-time application has also been implemented to demonstrate the feasibility of our approach. Results are very promising, clearly indicating that platform-independent, best-effort hard real-time software is a viable and realistic goal for many products, with great potential for significantly reducing real-time software development and maintenance costs.

1 Introduction

As hardware costs shrink and software requirements grow, the unwavering high cost of software development and maintenance has become a very powerful driving force behind the development of "write once, run anywhere" technologies such as Ada, Postscript, HTML, SQL servers, OpenGL, Java [Nil95], CORBA and many others. Such open system technologies essentially enable the programmer to concentrate on the *top* application layer only, with lower firmware and hardware layers requiring only minimal attention, thereby significantly cutting costs by substantially simplifying the software development and maintenance process. But perhaps even more important, the software produced by these technologies is essentially *platform-independent*, enabling it to run on a wide variety of available platforms, to be readily ported to new platforms when old ones get obsolete, and to consistently perform better as

newer platforms grow faster and more powerful.

Unfortunately, the impact of these rapidly developing platform-independent software technologies has hardly been felt in the area of real-time computing. It is widely assumed that real-time software and platform-independence are inherently mutually exclusive. After all, *real-time* systems are those which must interact with their environment in a timely manner. Untimely interactions in *hard* real-time systems, e.g. those found in defense, transportation, aerospace, energy, and health care, can be very hazardous and costly, and are therefore not to be tolerated. The real-time software within such systems is typically responsible for carrying out a substantial set of ongoing activities which must be appropriately scheduled to complete their functions within carefully crafted time intervals. The reliable scheduling of these perpetual activities clearly requires complete *a priori* knowledge of the available run-time resources as well as the computational and other resource requirements for each activity. Typical scheduling solutions are therefore thought to be inherently platform-dependent.

There are, of course, exceptions to this rule. In very *simple* real-time systems, such as those in which all activities are purely periodic, completely independent of each other, and all running on a single processor, relatively simple schedulability criteria and dynamic real-time scheduling policies can be applied uniformly to all candidate platforms. The hard real-time applications which we wish to address in this thesis are typically much too *complex* to be scheduled in a platform-independent manner, thereby causing them to be prohibitively expensive to develop and maintain.

This paper hopes to convince the reader that appropriate models, control algorithms, and software architectures can indeed produce platform-independent software even for hard real-time systems which are quite complex. To this end, we have developed a novel notion of *best-effort* hard real-time systems, i.e. systems which have hard real-time requirements and are also required to dynamically adapt their behavior to best exploit whatever run-time resources are currently available to them. As we will show, when running on shared-memory, multi-processor platforms, the requirements of such best-effort hard real-time systems are so complex that there are no known methods for addressing these complexities even when the runtime platform is *a priori* known and fixed. The *top-layer* design approach introduced by this thesis shows how such complexities can realistically be addressed, and in a *platform-independent* manner. Furthermore, we show that

hard real-time applications which have adopted the approach can then run *concurrently* and reliably on appropriate multi-processor platforms. We thus demonstrate that platform-independent complex real-time systems are not only possible, but that a top-layer design approach can also help resolve complexities which would otherwise be computationally intractable.

The scope of our research includes

- developing appropriate platform and aggregate job-oriented program models, including a novel notion of platform-independent *best-effort* meta-control in a hard real-time context, which can also be applied to concurrent real-time applications,
- devising practical job load-balancing, hard real-time job time-sharing, and best-effort meta-control algorithms,
- addressing several policy making issues, e.g. how to select an optimal suite of algorithms for a specific application,
- incorporating the above within a comprehensive top-layer architecture,
- demonstrating the feasibility of the approach by using it to implement a realistically complex real-time application which is portable and best-effort, and
- producing a comparative tradeoff analysis of the top-layer approach with existing approaches regarding cost, overhead, the ability to meet scheduling constraints and more.

Results are very promising, clearly indicating that concurrent, platform-independent, best-effort hard real-time software is a viable and realistic goal for many systems, with great potential for significantly reducing real-time software development and maintenance costs. True, in its current form, the top-layer approach developed here does *not* serve as a solution for *all* real-time applications. Nevertheless, by showing how it can be applied, as is, to a very challenging class of real-time systems, and by suggesting how current models and algorithms can be extended to embrace a wider class of applications, we hope to pave the way for future developments which will ultimately increase the portability and adaptability of real-time software.

First, we establish the setting by introducing the complex real-time problem domain in Section 2, and listing primary motivations in Section 3. Section 4 then briefly introduces the top-layer approach, describing the assumed platform and program models, outlining the software control elements required by these models, and using a few brief examples to illustrate how the top-layer architecture might be applied differently to different applications. This is followed by Section 5 which introduces ATLAS, a realistically complex real-time application which has been successfully implemented using the top-layer approach.

This paper is based on the first author's Ph.D. dissertation which contains all th details not presented herein due to space limitations. It can be obtained at **ftp://ftp.technion.ac.il/....**

2 Complex Real-Time Systems

The problem domain of this paper is a class of real-time applications which we call *complex real-time applications* because *the online decisions that they require are of such complexity that optimal decisions are intractable.* More specifically, we wish to address real-time applications which have hard (tight) timing constraints, difficult-to-schedule task characteristics, and dynamic, state-dependent, task sets and execution times. We further require that these applications reliably run, as is, on any of several platforms, each platform having a different number of shared memory processors, and each processor possibly running at a different machine cycle rate. When provided with various alternate mode of behavior, we require that these applications should also online adapt their behavior to best accommodate each platform and each application state. Finally we require that they should also be able to reliably run concurrently alongside additional complex real-time applications on any of these designated platforms. As we later show, *conventional task-oriented design approaches are not equipped to deal with such complexities even in a sub-optimal manner.*

The following subsections introduce real-time systems, describe the scheduling and meta-control decisions required by such systems, and better define the targeted complexities and assumed limitations.

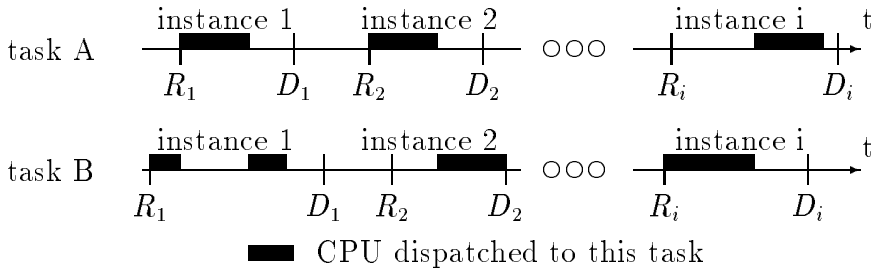

Figure 1: EDF Task Scheduling

2.1 Real-Time Systems

Real-time systems are those that must interact with their run-time environments in a timely fashion. Such systems are typically modeled as running *perpetually*, comprising a well-defined set of software modules called *tasks*, many of these tasks being invoked periodically or sporadically an infinite number of times to carry out their particular functions. As depicted in Figure 1, each task invocation is called an *instance*. Each time a task instance i is invoked, it must usually complete its function within a well-defined time interval, the beginning of that interval being referred to as the task *release* time, R_i, the end of that interval being referred to as the task *deadline*, D_i.

2.2 Scheduling and Schedulability

Intervals for several concurrent tasks will often overlap, so an appropriate real-time task *scheduler* must determine when each of the invoked tasks instances will be provided with the resources needed for task execution. The computational resources required by each task are usually provided in units of time, indicating how much time it would take for the current run-time processor to complete the task. Given the task computation times, release times, and deadlines, the real-time scheduler attempts to dispatch the processor to the various task instances in a manner which enables them to meet their deadlines. One effective *policy* for doing this is to always dispatch the task which has the earliest deadline. Such a policy is referred to as EDF (earliest deadline first) scheduling and Figure 1 demonstrates how such a policy would successfully dispatch a shared processor to the sample tasks, A

and B without missing any deadlines.

Given a set of tasks, a schedule is *feasible* if all tasks can be scheduled such that all timing constraints and resource requirements are met. Major scheduling issues include whether a feasible schedule *exists* for a given task set, whether such a schedule will be *found* by a given scheduling policy, and whether these questions can be resolved by a reasonable *schedulability* test or criteria.

2.3 Periodic and Sporadic

As depicted in Figure 1, task release times, deadlines, and computation times can be quite irregular, but this complicates scheduling and schedulability. *Periodic* tasks, on the other hand, greatly simplify scheduling, so real-time systems are modeled as sets of *periodic* tasks whenever possible. *Periodic* tasks are invoked repeatedly at fixed time intervals, i.e., letting T denote the fixed time period between instances, then $R_{i+1} = R_i + T$ for such periodic tasks. Periodic task instances must typically complete before their next instance release, so we also assume that $D_i = R_i + T$ for such tasks, unless otherwise specified.

Real-time systems, however, must almost always react to aperiodic events as well, e.g., external interrupts and internal signals. Such events invoke the execution of event-driven tasks which are referred to as *sporadic* tasks. Sporadic task release times are therefore not *a priori known*, but they can still be analyzed for schedulability, if they are invoked at rates which never exceed a specified upper-bound. Thus we require that all sporadic tasks in our applications have a known lower-bound, T, such that $\forall i : R_{i+1} - R_i \geq T$.

2.4 Hard and Soft, Values and Gain

Deadlines may be hard or soft, whereupon tasks are referred to as hard or soft in accordance with the nature of their deadlines. *Hard* deadlines are those for which the system fails to meet its requirements if a deadline is not met, while missed *soft* deadlines merely cause system degradation. The measure of degradation is often defined by a time-dependent *value* function attached to each soft task, which represents the contribution of each task instance to an aggregate *system value* as a function of *when* the task instance completes relative to its deadline. A common scheduling criteria for such soft tasks is

to maximize the aggregate system value per time unit, henceforth referred to as the system *gain*.

2.5 Processor Utilization

When seeking a cost-effective processor for a given set of perpetual real-time tasks, the primary criteria is the schedulability of hard tasks and an acceptable system gain for the soft tasks. A common *secondary* criterion is to maximize the processor *utilization*. When dealing with a set of purely periodic tasks, τ, each task $i \in \tau$ having a task period T_i, and a fixed computation time, C_i, for all instances of task i, then the processor utilization, $u(\tau)$, is readily computed by

$$u(\tau) = \sum_{i \in \tau} \frac{C_i}{T_i}. \tag{1}$$

2.6 Real-Time Applications

We define a *real-time* application as one comprised of arbitrary mixtures of perpetual periodic and sporadic tasks. In a *hard* real-time application, all tasks are *hard*, while in a *soft* real-time application, all tasks are *soft*. The real-time systems we wish to address will usually contain a mixture of both hard *and* soft tasks. A major challenge for the successful implementation of such real-time systems is finding a cost-effective platform and a practical algorithm for *scheduling* these tasks in a manner that *always* guarantees the hard deadlines while ensuring that the soft tasks contribute an *acceptable* system gain.

Practical real-time scheduling algorithms have been studied extensively in recent decades, e.g. [SC88], and it is generally well-known whether a given scheduling policy can reliably schedule a given task set on a given run-time platform. The easiest case to handle is when all tasks are periodic, preemptive, independent, and with fixed computation times per task. *Preemptive* tasks are such that their execution can be interrupted at any point of execution to execute a higher priority task. *Independent* task are those which have no precedence constraints and lack shared resources which require synchronized access, so that no task can be *blocked* by another task and therefore each task can be scheduled without considering how other tasks are scheduled.

2.7 Optimal Hard Scheduling

In a hard real-time context, a given scheduling policy is considered *optimal* if it always finds a feasible schedule if such a schedule exists. It is well known that when using conventional priority-driven scheduling, then EDF is optimal for the above systems when task priorities can be dynamically altered in accordance with the scheduling policy [Der74]. RM scheduling, on the other hand, is optimal for the above systems when the scheduling policy must determine a fixed-priority for each task [LL73]. As already mentioned, both policies have very simple utilization-based schedulability criteria. Both policies are also *practical* in the sense that online scheduling overheads are generally proportional to the number of tasks. RM scheduling overheads are lower, thanks to the fixed priorities, while EDF-scheduled processor utilizations are generally higher. No wonder that both policies are quite popular in hard real-time systems.

2.8 Best-Effort Soft Scheduling

In a soft real-time context, a given scheduling policy might be considered *optimal* if it maximizes the *expected* system gain. When dealing with independent sporadic tasks and fixed task priorities, allocating higher priorities to tasks with the higher value densities is known to be optimal [Whi92], the task value *density* being equal to the task value divided by the average task execution time. Due to their sporadic nature, task release times may vary from one run to another, so there may very well be a different allocation of priorities which would provide a higher system gain for specific runs. Nevertheless, given that these task release times are random, the above density-based priorities should maximize the mean system gain over a significant enough number of runs. Scheduling policies which maximize the *expected* gain are generally referred to as *best-effort* policies.

2.9 Speculative Soft Scheduling

When dealing with non-independent soft tasks, e.g. tasks which require synchronized access to shared resources, density-based priorities might not maximize the expected gain due to the blockage of higher density tasks by lower density tasks. A less greedy policy, on the other hand, might achieve

a higher gain by only scheduling task instances when all of their required resources are available, thereby reducing blockage. Heuristic policies which have been shown to often be effective, but which lack proved best-effort qualities, will henceforth be referred to as *speculative* policies.

2.10 Complex Real-Time Applications

Practical scheduling and schedulability criteria get more complicated when dealing with multi-processor platforms, with dynamic task sets, and with more complicated task set characteristics, such as those having both hard and soft deadlines, those having both periodic and sporadic tasks, those having varying computational needs, those having tasks with non-preemptive code segments, and those having inter-task dependencies such as those caused by precedence-constraints, communicating tasks, and shared resources. First of all, such systems require a *load-balancing* scheme for allocating tasks or task instances to each of the available processors without overloading any of these processors. Once allocated, appropriate scheduling policies must schedule the tasks on each of the processors while taking into consideration the unknown release times of the sporadic tasks as well as non-preemptive code segments and inter-task dependencies. Moreover, dynamic task sets require effective and practical load-balancing and scheduling algorithms which can be carried out online with reasonably low overheads.

The targeted problem domain for this paper includes applications with all of the above forms of complexity, but with the following restrictions, all for simplicity's sake:

1. We assume that tasks are computationally intensive and that the platform processor capacities are the only limiting factors when carrying out task load-balancing. Memory and I/O resources are therefore currently assumed to be infinite, even though extensions to multi-dimensional resource limitations appear straight-forward.

2. We assume that the dynamic task sets and task computational requirements are *state*-dependent, in the sense that the application has an *a priori* known set of states, and each state has a known fixed set of tasks with known worst-case computational requirements per task. This would enable us to resolve most load-balancing and scheduling is-

sues offline for all possible states if the number of states were reasonably small.

3. We generally assume that system states vary only *occasionally*, i.e. at significantly low rates, so that load-balancing and task scheduling for each new state can be carried out knowing that the new state will persist long enough to be considered perpetual. We later show, however, when and how this assumption can be relaxed.

4. We assume that for each possible system state, there exists an *a priori* known uniprocessor scheduling solution, i.e. a scheduling policy and computational capacity such that the policy can be shown to adequately schedule the required task set for that state when these tasks share a single processor with the given capacity. To accommodate all possible system states, all we have to do is provide a single processor with the worst-case computational capacity. A single processor with the worst-case capacity will usually not be available on the targeted multi-processor run-time platform, but this constraint guarantees, at least, that the task characteristics for each system state are such that the schedulability of any given task *subset* on any given processor can be determined offline, as well.

For the sake of brevity, real-time applications belonging to the above multi-processor problem domain, will henceforth be referred to simply as *complex* real-time applications. The paradigms serving current commercial and research real-time platforms cannot offer practical solutions for such applications. Real-time extensions to commercial operating systems, e.g. Real-time UNIX [FGG+91] and Real-time MACH [TNR90], generally cater to best-effort *average* performance and are therefore inappropriate for *hard* real-time. Scheduling in the distributed MARS platform [DRSK89] is static, and therefore inappropriate for *dynamic* task sets. Spring [SR89] strives for a mode flexible combination of off-line and on-line scheduling techniques, but it, too, must rely on static load-balancing and scheduling when dealing with mandatory hard tasks. The object-oriented MARUTI system [LTCA89] focuses on fault-tolerance and on-line FCFS (first-come-first-serve) guarantees using a non-preemptive, calendar-based, scheduling scheme, so it cannot accommodate dynamic task *characteristics* or best-effort adaptation to dynamic run-time circumstances. The distributed ARTS kernel [TM89] uses on-

line, RM-based, preemptive priority-driven scheduling, which can more readily cater to dynamic task sets and dynamic, value-oriented, load-balancing, but task priorities are static and therefore inappropriate for dynamic task characteristics. RM schedulability gets much more complicated when dealing with complex task characteristics, so ARTS can be impractical for dynamic task sets when they have complex task characteristics. The object-oriented CHAOS kernel [GS89] focuses primarily on efficient tailoring of the real-time kernel to arbitrary platforms, thereby enhancing the portability of its applications, but it, too, uses various forms of FCFS guarantee-oriented scheduling which cannot readily accommodate complex task characteristics. Thus, when dealing with hard real-time tasks, all of the existing platforms use static scheduling or online-FCFS guarantee scheduling, neither of which can support *best-effort adaptation* to dynamic run-time circumstances.

The inability of these platforms to cater to our complex problem domain is not surprising. It is well known [MD78] that *optimal* hard real-time scheduling on a multi-processor platform is NP-complete even for the much simpler case of purely periodic and independent tasks. When dealing with *complex* real-time systems with *hard* deadlines, even non-optimal policies can become impractical for online use since these cannot provide *a priori* guarantees that *all* hard deadlines will always be met. A *portable* solution must therefore somehow devise *offline* solutions to cater to all possible system states and all members of a family of platforms. But, as we soon point out, the growth of possible platform configurations and system states is usually exponential, so that such an approach would be intractable, as well. Thus, the development of a *top-layer* design approach for such systems is a formidable challenge.

3 Motivation

The quest for less complicated, more flexible, and less expensive real-time software is the primary motivation for seeking a platform-independent, *top-layer* design approach for complex hard real-time systems. Another primary motivation for portability is that it paves the way to two additional features of our top-layer approach: the ability to support concurrent real-time applications and best-effort adaptation in a hard real-time context.

In many real-time applications, customizing the application to run on a new platform and getting it to run concurrently with additional applica-

tions can be a relatively simple matter. This, however, is rare with *complex* real-time applications, and even more rare when these applications have *hard* tasks. Before we show how these goals are obtained, we must first understand why traditional bottom-up approaches are inherently limited in these respects.

3.1 Traditional Bottom-Up Approaches

As previously mentioned, choosing a *cost-effective* run-time platform is a major challenge for complex real-time applications. A very powerful platform, i.e. one which would result in a platform utilization which is much lower than what can be obtained by an optimal scheduler, might indeed minimize the task coding effort and simplify scheduling, but it also results in higher *production* costs. A less powerful platform might require more optimized code and a more sophisticated scheduling scheme, thereby incurring higher *development* costs. Traditional real-time design approaches work *bottom-up* from the platform hardware to the top-layer application, working from known times for hardware instructions to guaranteed worst-case upper bounds on the executions of system and library routines invoked by the application tasks, to guaranteed worst-case predictions on the timing behavior of the tasks themselves. As already mentioned, when dealing with a complex real-time application, finding an adequate load-balancing and real-time scheduling solution for a given platform can be a very computationally intensive process, and it must also consider every possible system state.

A bottom-up approach essentially forces us to commit to a particular platform very early in the system design when precise task requirements and characteristics, e.g. worst-case execution times, can still be quite vague. Thus, the run-time platform is essentially chosen before we can adequately evaluate its cost-effectiveness. Alter the nature, number, and capacity of the available platform processors and we must once again use computationally intensive algorithms to devise new load-balancing and scheduling solutions.

3.2 Portability and Real-Time Software Costs

The inherent platform-dependence of bottom-up real-time software has very serious implications on real-time software development and maintenance costs.

Major difficulties arise when poor initial estimates or evolving system requirements result in finding the targeted specifications unrealistic or the chosen platform to be inappropriate. For example, such difficulties are common place in cutting-edge projects in which little can be learned from past project experiences, or in which available processing resources are often limited by volume, weight, and power consumption. Such difficulties generally have a severe impact on project scheduling and budget, and this impact is often magnified by greater task scheduling complexities because new scheduling solutions may require radical code changes. Note that once the chosen platform has been found inadequate, porting a complex application even to a platform with an *equal* number of *faster* processors might require new scheduling solutions. The new schedulers may be required either because the original cyclic executive solution assumes minimal execution times for certain activities and these assumptions no longer hold, or because the original scheduling solution is no longer possible because the improvements in the original worst-case execution time estimates are not evenly distributed among the tasks. Porting a complex real-time application to a more cost-effective platform, e.g. to a smaller number of more powerful processors, will require a re-distribution of tasks among processors as well as new scheduling solutions for each processor.

Platform-independent software is a valuable asset even when the targeted platform is indeed appropriate. Application debugging and testing are formidable tasks for many real-time systems, and portable applications can be developed and tested on platforms with the most effective tools available and then ported to the production platforms, which are often bereft of such tools. Portability facilitates software development, eases software maintenance, encourages software reuse, and increases application longevity by enabling easy porting to new platforms when initial platforms become obsolete. Thus any design method that produces software that is more portable is worth its weight in gold.

3.3 Concurrent Real-Time Applications

Another primary motivation is the growing need for platform-independent real-time applications which can run concurrently and reliably along-side additional platform-independent real-time applications. As platform-independent Java-coded multimedia applets grow more prevalent, we require a design

method that will enable arbitrary sets of applets to reliably share a given run-time platform if the resources of that platform are adequate. In a conventional approach to real-time scheduling, each set of applets and each platform poses a different scheduling problem which would have to be resolved online to determine whether platform resources are adequate and if yes, what the scheduling solution should be. As already stated, for complex real-time applications, the online resolution of such problems would be computationally intractable. A major top-layer design objective is to enable us to resolve each application's scheduling needs independently and offline, and to do so in a manner which will support an online time-sharing mechanism which will consider the real-time deadlines of each application and enable us to quickly evaluate whether available platform resources are adequate.

3.4 Best-Effort Adaptation

The design of portable real-time applications makes way to a new challenge which was only mildly relevant for platform-dependent applications in the past. When designed to run on a given platform, an application will be generally be designed *a priori* to best exploit the available resources of that platform. When designed to run on any of several platforms, an application aught to be designed with inherent flexibility so that it can adapt to best exploit the available resources on each run-time platform. Even when designed for a specific platform, an application with state-dependent characteristics aught to best adapt to each state. To distinguish between this and best-effort scheduling, this will be referred to as *best-effort adaptation*. To this end, we introduce the notion of a set of application *modes* of operation to be chosen from, each mode known to contribute to some notion of a system gain. The current application mode must then be selected for each platform and for each state in a manner that will maximize the system gain without overloading the platform resources, e.g. without missing any hard deadlines. Different modes may employ different algorithms, different task sets, and different task requirements, e.g., different task periods. Such multiple-mode applications will henceforth be referred to as *best-effort* applications, and we will generally assume a large number of modes to best accommodate a large variety of states and platforms.

4 A Top-Layer Approach

Our goal, therefore, is to produce real-time software that can readily port from one shared-memory multi-processor platform to another without altering the program and without having to devise new scheduling solutions for each new platform. The run-time platform can therefore be finalized much later in the development cycle when all the necessary data are available.

The keys to our approach are simple yet flexible platform and aggregate job-oriented program models that should enable decomposing larger, generally highly complex, scheduling problems into several smaller, generally simpler, scheduling problems that can be resolved independently and offline, and then encapsulated in a platform-independent manner. Each of these smaller scheduling problems are referred to as a *job*, and software components developed using the approach can be treated as reusable *real-time objects* that should be readily incorporated into new programs without having to deal with scheduling details within them. Well designed jobs should facilitate near-optimal platform utilizations. Evolving system requirements could then be readily resolved with the related jobs without effecting the other jobs in the system. Schedulability is at the core of all real-time systems, so this modularity in the scheduling domain should greatly reduce the complexity of load-balancing and the making of best-effort mode selection decisions.

4.1 The Platform Model

We will generally assume that all targeted platforms belong to the same *family* in the sense that task execution times on any given processor within the family are assumed to be equal to the number of machine cycles needed to complete the task, divided by a processor characteristic called the processor *capacity*. This notion of capacity essentially reflects the processing *speed* of that processor. All platforms within a family are assumed to differ only in the number and capacity of the shared-memory processors in each platform.

When all processors share a common machine language and the execution of each machine instruction requires an equal number of machine cycles for all processors, then the processor *capacity* is simply equal to the inverse of the machine cycle time and therefore prescribed in MHz units. For such platform families we seek *binary-level portability*, whereby top-layer applica-

tions should run on an arbitrary number of shared-memory processors with arbitrary processor capacities without modification. This restricted notion of a platform family can be extended to embrace *source-level portability*, i.e. different processors and different machine languages, as well. Once the family of processors has been established, each platform is fully characterized by a set of processor capacities.

Note that the primary obstacle to portable real-time applications is the complexity of resolving scheduling in a uniform manner for platforms with different task execution times and different resource capacities. Even when assuming portability over a specific family of processors, different processor numbers and capacities can produce an exponential growth of different scheduling problems. When dealing with hard complex real-time applications, resolving scheduling and schedulability for all possible platform configurations would therefore be intractable. Thus the development of portable real-time applications is a formidable challenge even when limited to specific processor families.

4.2 The Static Top-Layer Program Model

In the *static* program model, the software application is modeled as a set of real-time subsystems, called *jobs*, each job comprising of any number of periodic and sporadic real-time tasks. As previously described in Section 2, the computational requirements of each task are typically provided in time units and for a specific processor. Our platform model assumes a specific family of processors, but the processor capacity (speed) may vary, so the task computational requirements are therefore specified, instead, in machine cycles. Real-time scheduling is independently resolved offline for each job by initially assuming that the job task set runs on a dedicated processor, by selecting an appropriate conventional policy for scheduling the job's internal tasks, and by computing the minimal processor capacity, i.e. the minimal rate of machine cycles needed, to guarantee all *hard* deadlines and the minimal *soft* value contributions required by that job. One job, for example, could be employing a cyclic executive policy to schedule its tasks, and will be known to satisfy its temporal requirements when run on a processor with a machine cycle rate of 2 MHz or more, while another job might use a rate-monotonic scheduling policy for its tasks, and its minimal machine cycle rate might be 10 MHz.

Thus, the top-layer program model can be applied to arbitrary job sets, provided that each job consists of a task set that has a known scheduling solution for an arbitrary known processor capacity. The real-time task characteristics and scheduling solutions are encapsulated within the jobs by making each job personally responsible for its internal scheduling, e.g., by including a scheduling task or thread for facilitating the chosen scheduling policy. The tasks of a job could even be completely serialized at compile time to eliminate the need for this run-time scheduling task.

4.2.1 Job Time-Sharing

The load-balancing granularity in our model is the program *job*, i.e., it is understood that all the tasks in a given job will always run together on a single processor, always share the same memory, and otherwise compete with other jobs for resources offered by the shared platform. Once a satisfactory scheduling algorithm is determined offline and implemented for each job, then the normally dynamic, on-line scheduling of all the tasks in the system, i.e., of all the tasks in all the jobs, can be reduced to that of the dynamic, on-line scheduling of only the jobs. Once a job is scheduled, its internal offline-determined scheduling algorithm determines which of its tasks will run within its allocated running slots. An appropriate real-time job *time-sharing* scheme can then enable several jobs to share a given platform if it can guarantee that each job will be dispatched in a timely manner and allocated the required rate of machine cycles such that the job's internal scheduling policy will always reliably schedule the job's tasks. An *overloaded* processor is one which has been allocated a job set for which such guarantees cannot be provided by the job time-sharing mechanism. The jobs running on a given platform may belong to a single application or may belong to several concurrent applications, so the same scheme can be used to time-share any number of real-time applications.

4.2.2 The Job Bandwidth

To accommodate the above time-sharing scheme, the minimal rate of machine cycles required for each job must be appropriately augmented to accommodate the worst-case job time-sharing overheads. This augmented rate of machine cycles is referred to as the job *bandwidth*, a term used to denote a

periodic allocation of machine cycles as in the bandwidth-preservation techniques of [LSS87, SSL89]. We then require that the job time-sharing scheme on each processor must reliably guarantee scheduling for an arbitrary job set if the sum of job bandwidths does not exceed the given processor capacity. Different time-sharing schemes can be used for different classes of complex real-time applications, with EDF job time-sharing being quite effective when dealing with computationally intensive tasks and relatively relaxed task frequencies.

Another interesting job time-sharing feature is that *minimal* bandwidth allocations are guaranteed for *all* jobs in the system so that jobs with low priority tasks never starve when running together with jobs containing higher priority soft tasks that would otherwise fully load the processor. The proposed EDF job time-sharing scheme may appear trivial, but we know of no previous attempt to thus resolve in this way complex scheduling problems of this nature.

4.2.3 Black-Box Jobs

We also require that the job time-sharing mechanism need know only the job bandwidths and the current earliest deadline for each job. This enables top-layer jobs to be treated as *black-box* real-time software objects in the sense that they can be reliably load-balanced and time-shared without having to divulge their internal task characteristics and scheduling policies. This contrasts with conventional approaches where the real-time platform can only cater to the needs of *white-box* applications, i.e. applications which reveal all tasks and a significant set of task characteristics. Each job sharing a given processor can therefore employ a totally different scheduling policy, e.g. CE, EDF, or RM with PCP. Major job design goals should include minimizing the job bandwidths and maximizing the job *bandwidth utilizations*, i.e. the rate of machine cycles actually consumed by the job tasks divided by the job bandwidth. If these goals are achieved, time-shared, real-time scheduling can be near optimal. Job bandwidths and utilizations should therefore be carefully considered when decomposing the software application into jobs and when selecting scheduling policies for each job.

4.2.4 Job Load-Balancing

Schedulability is at the core of all real-time systems. Therefore, this black-box modularity in scheduling domain greatly reduces the complexity of load-balancing when the job is the basic load-balancing entity. As previously mentioned, arbitrary job sets may share a given processor if the sum of job bandwidths does not exceed the processor capacity. Thus the allocation of jobs to processors on any given shared-memory multi-processor platform is essentially a generalization of the classic binpacking problem, and the processor capacities should generally be great enough to contain several jobs. Binpacking is strictly NP-complete, but it has several practical algorithms, e.g. first-fit-decreasing, which we have appropriately adopted to carry out job load-balancing in our model. A given platform can therefore serve any application if the job bandwidths of that application can be partitioned over the processors of that platform without overloading any of the processors.

This static program model is portable in the sense that job load-balancing is readily carried out on any platform without having to alter the scheduling algorithms being internally used by each job. This scheme is also readily applied to systems running several concurrent real-time programs, in which a request to run a program is accepted if the subsequent aggregate job set can be load-balanced over the available processors based solely on the job bandwidths within the programs. Thus job bandwidths and appropriate algorithms for real-time job time-sharing and load-balancing provide a simple mechanism for determining whether arbitrary top-layer programs can reliably run on arbitrary platforms.

4.3 The Dynamic Top-Layer Program Model

The dynamic, top-layer program model assumes that system requirements and available resources can be represented by well-defined platform and program *states* that vary at relatively low rates. The *platform state* reflects the available platform resources and it is altered when a processor fails or when there is a *hot* (on-the-fly) reconfiguration of hardware. The *program* (or workload) state is altered when a user request or other run-time factors require changes in the current task mix and/or affect the task execution times and how quickly these tasks must respond. Such changes clearly alter the minimal processing bandwidths required by each job to satisfy its task needs,

so the current allocation of jobs to processors may no longer be efficient or valid.

To address the needs of such systems, the static top-layer program model is extended by letting each job support one or more alternate *modes* of operation, sometimes referred to as job *versions*, each mode encapsulating an arbitrary set of tasks and an appropriate internal scheduling policy for satisfying each task's deadlines when it is provided with a minimal *bandwidth* of machine execution cycles. Each job mode is therefore fully characterized by the required bandwidth along with a *reward* representing the aggregate value contributed by the internal tasks when provided with the given bandwidth. These job mode bandwidths are already determined offline along with the internal scheduling policies for each job mode, so they must be provided as tables or functions of the *a priori* known program states. Thus in a cellular base station application, (see Section 4.3.4) the bandwidth for handling a telephone session should vary as a function of the current session mode, e.g. the chosen sample rate and error-correction method, as well as the state of the session, e.g. whether connecting, speaking, or faxing. Job mode rewards, also, may be offline determined as functions of the program state, e.g. a higher session sample rate will contribute very little when connecting and in other states the contribution will depend on the spectral nature of the information being transferred. *Suspendable* jobs are also supported by the dynamic program model. Jobs that can be suspended have a zero-bandwidth, zero-reward suspend mode to indicate that possibility.

In the model, the aggregate system gain must be adequately expressed as a function of the rewards provided by the current job modes. In some cases, the system gain function might be additive, i.e. equal to the sum of current job mode rewards. Best-effort accommodation for each state is accomplished by the online selection of job modes that maximize the system gain for that state without overloading the available processors. Thus, all job mode bandwidths and rewards must be reevaluated with each altered state, whereupon they must be considered in the re-selection of job modes and the re-allocation of jobs to processors. Each job may have an arbitrary number of states, and the application state is a combination of current job states, so the number of system states can be exponential. For such cases, the offline resolution of load-balancing and scheduling at the task level would be intractable for all possible systems states even when dealing with relatively simple task sets.

4.3.1 Online Meta-Control Decisions

The process of choosing between available software adaptations, e.g. alternate job modes, is referred to as software *meta-control* because it essentially determines which tasks must be scheduled, what their time constraints are, and possibly which scheduling policy should subsequently be used. In the dynamic top-layer program model, online software meta-control decisions require best-effort job mode selection and job load-balancing, which is shown to be equivalent to a composite binpacking and multiple-choice knapsacking problem, which is NP-hard. Online meta-control decisions must be carried out with each program or platform state transition. Therefore, minimal and reasonable decision-making overhead is crucial. To minimize the response time for such changes, we have developed two families of practical approximation algorithms, QDP and G^2, for making near-optimal best-effort meta-control decisions with reasonable time complexity, typically providing 95% performance in fractions of a second for substantial numbers of jobs, modes, and processors.

In our top-layer program model, task characteristics, constraints, and scheduling policies are encapsulated within each job mode. Thus each job must also implement an online mechanism for providing the required bandwidth for each available mode as a function of the current system state. We assume that extensive offline schedulability analysis for each job mode can produce efficient online mechanisms for providing these state-dependent job mode bandwidths, and a similar online mechanism must be provided for the state-dependent job mode rewards as well. The system must therefore respond to each state transition by querying each job for its current mode alternatives, mode bandwidths, and mode rewards so that it can make a best-effort meta-control decision.

4.3.2 Meta-Control Decision Epochs

As already described in Section 2, we assume in our model that the rate of altered system and platform states is reasonably low so that task sets remain stable enough for conventional scheduling policies to be applied to them over a significant time interval. Furthermore, we assume that software configuration adaptations, i.e. transitions from one job mode to another, can either be carried out instantly or be done quickly enough that these adapta-

tion overheads can be accommodated. As we soon show by example, many applications can be designed to meet these limitations, and we also believe that these limitations will eventually be relaxed by appropriate extensions to our current meta-control algorithms. We also assume that meta-controller decision-making processes can be accommodated without disrupting essential real-time processes. One way of avoiding disruption is to dictate that all meta-controller decisions take effect at discrete points in time called *decision epochs*, and to ensure that task deadlines are adequately synchronized with these epochs so that new tasks can be introduced and old tasks can be flushed simultaneously.

4.3.3 A Virtual Reality Example

Consider, for example, a virtual reality program required to maximize the overall realism produced by simulating and displaying realistic behavior of a dynamic set of visible entities. Each entity can be treated as a job with several modes, each mode using a different simulation update rate. The realism contributed by each entity's mode could be a factor of the entity's state, e.g. the nature of its motion as well as observed size and visibility, so the reward attributed to each entity must consider how the update rate for that entity will contribute to the overall realism in its current state. Higher update rates might even imply different simulation models requiring a different set of tasks and constraints, so that the task set serving each entity mode may be different. Once the entity modes have been determined, it is essential that all hard deadlines associated with these tasks be met. To facilitate seamless meta-controller adaptations, we might decide that adaptations may only occur at epochs that are at times that are multiples of 100 milliseconds. This decision implies that all other update rate alternatives should be multiples of 10 Hertz as well. This would enable the system to instantaneously switch job modes, i.e., update rates and simulation models, at any given adaptation epoch, since none of the tasks belonging to the replaced simulation models should be executing at those points in time. Another interesting possibility might be to carry out the adaptations in stages, whereby all transitions to lower bandwidth modes are carried out first, asynchronously, at their first possible opportunities, and all other transitions can then follow in a carefully crafted sequence which never overloads the system. The existence of such a sequence would have to be ascertained as part of the meta-controller

decision-making process.

4.3.4 A Cellular Base Station Example

There are, however, many examples in which meta-controller decision epochs and task timing constraints need not be synchronized. In such a case, the need for predetermined decision epochs and instantaneous transitions are not necessary. In a cellular phone base station, for example, a rising number of subscribers or a faulty processor would cause the base station to gradually curtail a variety of voice enhancement and noise-cancellation processes so that a uniform highest overall service level is maintained for the current number of subscribers and processors. A base station has several seconds to decide whether or not it can respond to a new subscriber, and how it can best adapt to the new work load. This provides plenty of time for gradually phasing out old tasks and phasing in new tasks, without risking any missed hard deadlines in tasks serving current subscribers. This example illustrates how even significant meta-control adaptation overheads might be handled when the system is allowed an adequate response time to changes in its run-time requirements.

5 The ATLAS Experiment

Once the algorithms have been devised, it is necessary to formally derive their computational complexity to determine whether they have a chance of behaving as hoped. However, this formal modeling is not enough. Theoretical estimates are only orders of magnitude and it is not known what the multiplicative constants are. It is necessary to build and run a system designed according to the top-down principles, making use of these algorithms, in order to verify the effectiveness of the approach. To this end we have devised a scalable, best-effort real-time system with a realistic mix of hard and soft deadlines, with very frequent meta-control decision epochs, capable of porting unaltered to any of several accessible platforms, while meeting all hard deadlines and maximizing the system gain for each platform.

The system developed is a music synthesizing system called ATLAS, an acronym for *a top-layer audio synthesizer*. All digital synthesizers produce digital samples that must be output via appropriate digital-to-analog inter-

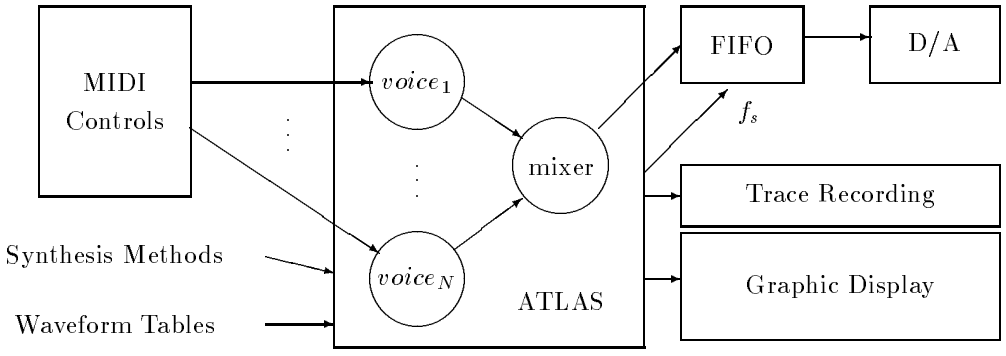

Figure 2: The ATLAS Application

faces at a predetermined sample rate. As depicted in Figure 2, the digital samples produced by ATLAS are all software computed and fed into a hardware queue (FIFO) in real-time, so the digital samples for each note must be generated and mixed at rates that sustain the flow of output samples via the hardware queue. The notes generated by ATLAS are dictated by a standard online MIDI[1] input driven by several electronic instruments, each instrument producing sequences of requests to start or stop the playing of one or more concurrent notes, e.g. when playing a chord. Each ATLAS note that has to be generated is referred to as a *voice*, with higher pitched voices requiring higher sample generation rates. Each generated note is also recorded on the the real-time display depicted in Figure 3. Thus the internal application state is a function of the number of concurrent MIDI-dictated voices, the nature of the instruments, and the voice pitches, and ATLAS is required to best adapt to each state on any of the targeted runtime platforms.

To accommodate different platforms and dynamic system states, ATLAS provides several synthesis modes (methods) of varying quality for generating the digital samples for each instrument. Each mode is characterized by the computational bandwidth requirements of the mode and the signal-to-noise ratio produced by the mode. As illustrated in Figure 4, each synthesis mode may require any number of periodic tasks. An *FM* (frequency modulation)

[1]MIDI is an acronym for the standard Musical Instrument Digital Interface Specification 1.0, which was first established by the International MIDI Association in North Hollywood, California in 1983.

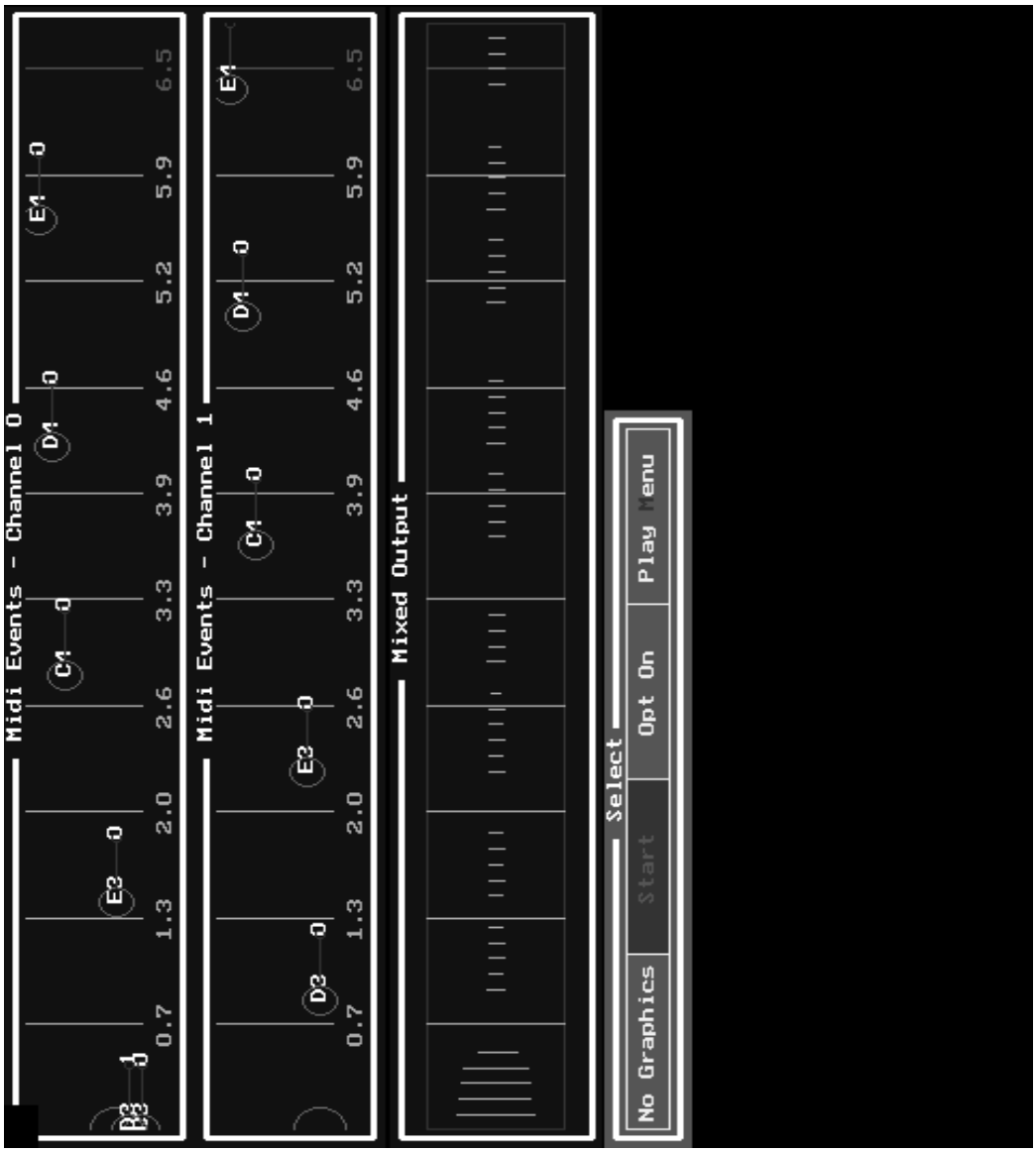

Figure 3: Real-Time ATLAS Output

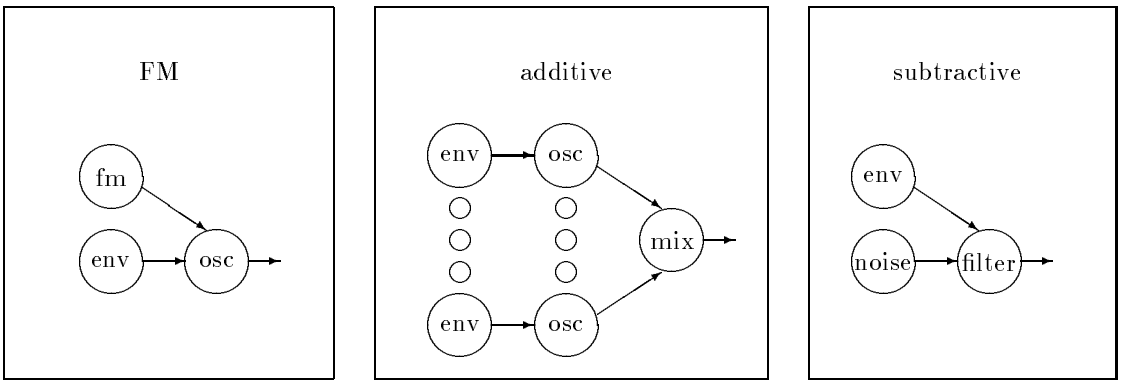

Figure 4: Typical Musical Synthesis Methods

synthesis mode, for example, requires an osc oscillator task for generating a periodic stream of digital samples which produce a voice waveform with an appropriate pitch, another fm task for generating values which frequency modulate the oscillation pitch, and a third env task for generating oscillation amplitude values to obtain an appropriate power envelope for each note. An *additive* synthesis method may require any number of oscillator and envelope tasks along with a mixer task to mix their outputs. A *subtractive* synthesis method will require a noise generation task, a digital filtering task to obtain a specific frequency spectrum, and an envelop task to modulate the amplitude. These tasks are periodic in the sense that they are periodically invoked to process and generate sample *packets* containing several samples at a time. These packets can then serve as inputs to other synthesis tasks, including the final mixer task of Figure 2 that mixes the samples of all concurrent notes before feeding its output packets into the final hardware queue.

Discontinuities in the audio output, off-pitch notes, and off-beat notes, are all very annoying even to the uneducated human ear and are therefore not tolerable. Thus, *hard* ATLAS requirements include that all task deadlines which affect voice pitch, voice synchrony, the steady flow of digital samples must always be met, and all events and decisions must be recorded so that performance and compliance can be verified. The *soft* requirements are that the generated notes be displayed, and the quality of the produced music be

best-effort, i.e. that for every system state, the system must always select the synthesis modes that maximize the overall SNR (signal-to-noise ratio) without jeopardizing the completion of any hard deadlines. For any given platform, one can always devise a system state, i.e. a large enough number of concurrent high-pitched voices with very demanding synthesis methods, which will overload that platform even when using synthesis methods with the lowest bandwidths. Lower volume voices contribute less to the SNR, so lower volume voices will be shed until the hard deadlines of all remaining voices can be guaranteed.

Each concurrent voice is treated here as a *job*, the synthesis method options available to each voice are treated as job *modes*, and the number and nature of concurrent voice jobs is clearly dynamic. As indicated by Figure 2, besides the dynamic number of voice jobs there are several additional jobs in the system responsible for MIDI processing, interactive graphic display, and the time-stamped recording of the internal trace events necessary for verifying and evaluating system performance. The top-layer architecture must facilitate the online best-effort selection of voice job modes, the dynamic load-balancing of the varying number of jobs over the available processors, and the real-time time-sharing of the job subsets sharing each of the processors, and it must do so reliably on a diverse range of platforms.

The ATLAS application actually poses several additional challenges to our top-layer approach. Somewhat contrary to our model assumptions, ATLAS state transitions are actually quite frequent, forcing us to carry out meta-controller decisions several times a second. Real-time scheduling must therefore be resolved in a manner which can accommodate synthesis mode transitions at such high rates. Furthermore, voice synthesis mode transitions cannot be carried out while a note is being generated, so that each meta-control decision must consider a different set of voice mode alternatives for each decision epoch. Finally, when dealing with *hard* real-time applications, our top-layer approach generally assumes WCETs for all tasks and it assumes that all platforms are such that task WCETs are readily derived for each platform. We believe, however, that our top-layer approach has great potential for accommodating non-deterministic platforms even when assuming sub-WCETs, thereby significantly increasing system performance for such platforms. Currently targeted ATLAS platforms include non-deterministic PC and Silicon Graphics platforms so that this potential might be explored and verified. The successful application of our top-layer approach to this

application would therefore serve also as an indicator regarding the flexibility of the approach and the hardness of assumed limitations.

Our current, non-preemptive, uniprocessor version of ATLAS has been fully designed and coded for two very different platform families, PC compatibles and Silicon Graphics, by way of a uniform platform-interfacing layer called VaPoR. Note that developing VaPoR-like layers will no longer be necessary as cross-platform virtual platforms for real-time evolve, e.g. the PERC virtual machine for real-time Java programs. Thus far, this initial ATLAS version has been fully tested for a small set of musical scores and program configurations on two very different platforms within the PC family, the author's 486/33 MHz PC, and the advisor's Pentium which is more powerful by an order of magnitude. First it was developed and tested on the author's machine and then binary-ported and run without modification on the advisor's machine as well. A standard MIDI file served to emulate online MIDI input to ensure that identical experiments could be carried out on both machines. Continuous output sample streams and precise voice pitches were maintained on both machines, but the SNR was much higher on the more powerful platform due to the automatic choice of higher sample rates and more accurate synthesis modes. When platform resources might be overloaded, only the higher volume voices were generated. No audible faults were detected on either platform, even though assumed voice generation execution times were significantly less than worst-case!

For a quantitative analysis of performance, ATLAS supports a minimally intrusive trace mechanism whereby events, decisions, and strategically placed program checkpoints are all recorded in a cyclic double-buffered trace buffer which is periodically flushed to an appropriate file under meta-controller control. A detailed analysis of these recordings verified that no *hard* requirements were breached. The best-effort average system gain, i.e. the SNR produced by the choice of voice synthesis modes for each time frame, was indeed 95 percent the optimal for each platform, clearly matching that which was theoretically anticipated. Above all, we have successfully integrated and applied the software architecture of Section 4.3, the real-time time-sharing principles of Section 4.2.1, and the meta-control algorithms of Section 4.3.1 in a manner wich enabled the top-down developed program to port very nicely to several platforms within a predetermined family. Furthermore, the current ATLAS application has been implemented in a manner which should enable it to recompile, as is, and run equally well on any Silicon Graphics platform,

thereby illustrating that top-layer applications can readily port to additional platform families, as well. Though fully tested, so far, only on uniprocessor platforms, the extension to shared-memory multiprocessors is relatively straight-forward, and no difficulties are anticipated. Thus the current ATLAS version already provides ample evidence that a top-layer is feasible - it works!

Additional ATLAS requirements include support for a *soft* real-time interactive, menu-driven, graphic display, as well as a *hard* real-time recording of all events and decisions so that the timeliness of ATLAS activities can be validated and the system gain can be evaluated. Targeted platforms include several conventional parallel and distributed platforms which also have non-deterministic elements, so ATLAS has been designed to effectively accommodate these elements, as well. ATLAS is an excellent testbed for the study of complex real-time architectures because it is complex, platform requirements are minimal, it is readily configured and scaled to overload any conceivable platform, and hard real-time faults are readily detected by sensitive eyes and ears.

5.1 A Conventional Approach

A conventional approach to such requirements would be to determine the worst-case MIDI state to be supported by the system as well as the worst-case set of operator-induced jobs, compute the WCETs for this state and these jobs, and seek synthesis methods for this state which could be accommodated by the platform. This is also how *hardware* synthesizers are designed, but, as previously described, this would result in a very low average utilization of platform resources, possibly as low as only a few percent. Furthermore, determining the WCETs for targeted platforms is not always feasible, particularly when dealing with the ATLAS-targeted platforms which have non-deterministic components, whereupon such an approach cannot serve *hard* real-time applications. A conventional approach would also require a complete worst-case characterization for all system components, including those of the underlying platform layers as well. On a UNIX platform, for example, we would also have to model all underlying audio, mouse, keyboard, display, cache management, and file I/O activities, as platform tasks which have to be scheduled by an appropriate integrated scheduling policy along with the top-layer tasks belonging to the application. When overloaded, conventional

approaches shed loads on a FCFS basis, without being able to consider the effect of these loads on system gain.

Conventional approaches are also *inherently* non-portable. When porting from one platform to another, scheduling policies generally have to be revised because underlying platform tasks and all task characteristics will differ on the new platform, thereby altering the total set of tasks and the WCET ratios between them. This is true even when porting to a faster platform with identical machine instructions, since different operating systems, hardware devices, software drivers, and virtual memory configurations can all adversely affect scheduling. Moreover, different application task characteristics usually effect the way the program should adapt to varying circumstances, so conventional approaches cannot generally support automated best-effort adaptation in a portable manner.

5.2 ATLAS Advantages

ATLAS's architecture enables it to automatically adapt to arbitrary MIDI-specified musical requirements by selecting those modes which will maximize the synthesized SNR without sacrificing any functional and temporal requirements. ATLAS's top-layer design also enables it to automatically port from one platform to another, always best exploiting the resources available on that platform. Probably the most exciting aspect of our ATLAS experience is the success it appears to have had in more effectively accommodating the non-deterministic elements prevalent in conventional platforms. Our current version has been assuming typical, rather than elusive worst-case, execution times, with safety margins of only 20 percent. The output audio FIFO acts a shock absorber, and the current fullness of the FIFO queue acts as an indicator as to how close we are to a missed deadline. We can therefore relax our assumptions when the FIFO is full and be more careful when it gets empty. Meta-control decisions are made for each time frame, enabling us to quickly adapt when necessary. For more serious noise spikes, we also provide safety valves which can respond instantly, even before we reach our next meta-control decision epoch. The inherent flexibility and the adaptive architecture thus enable ATLAS to best-effort exploit the current platform while maintaining platform utilization levels which would probably be lower by two magnitudes if worst-case states and WCETs were assumed. When unable to synthesize *all* of the MIDI dictated voices, ATLAS always sheds

the less audible voices, rather than shed on an arbitrary FCFS basis. ATLAS is inherently portable. Therefore ATLAS performance will *automatically* improve with time as available platforms grow more powerful.

Acknowledgments

References

[Der74] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *Proc. IFIP Congress*, pages 807–813, 1974.

[DRSK89] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. *ACM Operating Systems Review*, 23(3):141–157, July 1989.

[FGG+91] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker, and M. Roberts. *Real-Time Unix Systems*. Norwell, NA:Kluwer, 1991.

[GS89] Prabha Gopinath and Karsten Schwan. CHAOS: why one cannot have only an operating system for real-time applications. *ACM Operating Systems Review*, 23(3):106–125, July 1989.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 20(1):46–61, January 1973.

[LSS87] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environments. In *Proceedings of 8th Real-Time Systems Symposium*, pages 261–270, December 1987.

[LTCA89] Shem-Tov Levi, Satish K. Tripathi, Scott D. Carson, and Ashok K. Agrawala. The MARUTI hard real-time operating system. *ACM Operating Systems Review*, 23(3):90–105, July 1989.

[MD78] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proc. of the Seventh Texas Conference on Computing Systems*, November 1978.

[Nil95] Kelvin Nilsen. Issues in the design and implementation of real-time java. Technical report, November 1995.

[SC88] John A. Stankovic and Sheng-Chang Cheng. *Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey*, pages 150–173. IEEE Computer Society Press, 1988.

[SR89] John A. Stankovic and Krithi Ramamritham. The Spring kernel: A new paradigm for real-time operating systems. *ACM Operating Systems Review*, 23(3):54–71, July 1989.

[SSL89] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1:27–60, 1989.

[TM89] Hideyuki Tokuda and Clifford W. Mercer. ARTS: A distributed real-time kernel. *ACM Operating Systems Review*, 23(3):29–53, July 1989.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In *Proceedings of USENIX 1990 Mach Workshop*, pages 73–82, October 1990.

[Whi92] D. J. White. *Markov Decision Processes*. John Wiley & Sons, 1992.