A Top Layer Design Approach for Adaptive Real-Time Software

J. JEHUDA[*] **and D. M. BERRY**[†]

[*] *Technion - Israel Institute of Technology, Department of Electrical Engineering, Technion City 32000, Haifa, Israel.* jehuda@techunix.technion.ac.il
[†] *Technion - Israel Institute of Technology, Department of Computer Science, Technion City 32000, Haifa, Israel.* dberry@cs.technion.ac.il

Abstract. The flexible job-oriented program model and distributed *top-layer* architecture described in this paper represent a novel top-layer approach to real-time software design and implementation, which can achieve portable, adaptable, fault-tolerant, and predictable high-value performance for a significant class of large-scale real-time systems with dynamic requirements, resources and constraints. Following a brief introduction to real-time software control issues and current *layer-by-layer* trends and limitations, we present an adaptive top-layer alternative to program modeling and control which can efficiently guarantee dynamic hard and soft requirements on a distributed platform, while providing *best-effort* system values for arbitrary system state sequences. Practical and scalable control algorithms have been devised, analyzed, and tested, which strongly suggest that such an approach is viable even for dynamic large-scale and complex systems where conventional layer-by-layer alternatives fail. We also show how the proposed top-layer architecture might efficiently accommodate known levels of non-deterministic behavior within the platform and the environment. A musical ATLAS testbed is being developed to illustrate the feasibility of this approach.

Keywords. Complex real-time software, dynamic systems, best-effort adaptation, hybrid scheduling, distributed processing.

1. Introduction

A real-time system can be viewed as a collection of interacting components at various hardware and software layers, e.g. semiconductor components, the hardware architecture, operating system, programming languages, and compilers. The *top-layer* comprises the application program, while all underlying layers are collectively referred to as the run-time *platform*[1]. A conventional *layer-by-layer* approach assumes that *predictable* real-time systems can be obtained only by ensuring that *all* system components, within *all* system layers, behave in an *a priori* known predictable manner (Halang and Stoyenko, 1991). Component interactions and *worst-case* characteristics can then be analyzed to ascertain that all critical program tasks will be schedulable under all foreseeable circumstances. Unfortunately, current layer-by-layer methodologies cannot efficiently accommodate real-time systems which are *complex*, i.e. systems which must simultaneously cater to several dimensions of real-time systems, such as multiple processors, tight and loose time constraints, hard and soft deadlines, periodic and aperiodic tasks, static and dynamic subsystems. In addition, these methods cannot support real-time systems which operate in uncertain environments, or real-time platforms which contain non-deterministic elements with worst-case characteristics which are too high. Many are concerned that challenges posed by such systems are not being met (Stankovic, 1988). The primary objectives of this paper are to describe a job-oriented program model and a software control architecture which facilitate a platform-independent *top-layer* approach, which might be more appropriate for such systems.

A top-layer approach differs from a conventional *layer-by-layer* approach, in that it attempts to achieve predictable real-time behavior by focusing only on the program layer. Such an approach was first proposed by Stankovic and Ramamritham (1990), who point out that even layer-by-layer methodologies are not fool-proof, always requiring error handlers to deal with unexpected input, inevitable hardware failures, and probable software bugs. Even in a top-layer approach there are always system components, e.g. error handlers, which may still require layer-by-layer techniques to maximize guarantees. They suggest, however, that strict layer-by-layer requirements can be relaxed for *most* system components, as long as we can guarantee that error handlers or other alternative

[1]Platform layers do not include environment-dictated system elements, e.g. sensors and actuators, which are viewed as part of the application *environment*.

actions will always bring the system to a safe state. To date, no one has demonstrated how a top-layer approach might be realized. Moreover, it has never been shown that such an approach can indeed accommodate complex multi-processor systems, uncertain environments, and non-deterministic platforms.

The top-layer approach, herein described, essentially adopts a *job-oriented* RtTS scheme (Jehuda et al, 1995) originally devised for time-sharing systems with dynamic sets of real-time jobs. The RtTS architecture uses a divide-and-conquer strategy, most appropriate for *complex* systems. The strategy requires that each job be internally responsible for the scheduling of its own tasks, using arbitrary real-time scheduling policies. Arbitrary sets of such jobs can then reliably *time-share* any given processor, subject to a very simple joint schedulability test. Job sets, job requirements, and processing resources, are assumed *dynamic*, so the RtTS control architecture is designed to simultaneously carry out best-effort job adaptations and dynamic job load balancing in a manner which *automatically* maximize the system value for arbitrary system circumstances. Employing efficient control algorithms, these decisions are made in fractions of a second, making the strategy appropriate for uncertain environments. When using platforms with non-deterministic elements, a flexible control framework and novel feedback mechanisms enable avoiding worst-case assumptions and then quickly adapting to avoid impending faults.

Whereas each RtTS job originally refers to an independent application, the *job* abstraction is applied here to loosely-coupled subsystems within a single application. Each job may support several alternate modes of operation to choose from, and may also recognize various internal states which affect the job's current processing requirements and contribution to the overall system value. Scheduling policies employed by each job may also be mode and state-dependent, enabling application of arbitrary deterministic or speculative scheduling policies to each subsystem, wherever and whenever necessary. Additional control elements are introduced into the RtTS architecture to enable porting an application from one platform to another without source-level modification. The result is a *platform-independent* real-time application which has been completely designed and implemented with very few assumptions regarding the platform layers. A musical ATLAS testbed (Jehuda and Berry, 1994) is being developed to illustrate the feasibility of this approach.

This paper is organized as follows. We begin with a description of the job-oriented program model and the real-time software control issues in Section 2. Here we use a remote-controlled craft example to demonstrate how it can be applied to complex systems targeted for uncertain environments, possibly running on a non-deterministic platform. Current *layer-by-layer* trends and limitations are briefly discussed in Section 3. The RtTS software control architecture is then described in Section 4. To facilitate the development of platform-independent programs, Section 5 introduces additional elements into the architecture, whereupon Section 6 suggests how the subsequent top-layer architecture can accommodate known degrees of platform non-determinism. Conclusions appear in Section 7.

2. The RtTS Program Model

The top-layer approach is appropriate only for real-time systems which can apply the *job*-oriented multiprocessor program model, described below. In particular, it is useful only for programs which can be decomposed into a set, J, of *loosely-coupled* subsystems, called *jobs*[2]. The active set of jobs, J, is assumed to be dynamic, running on a set of processing nodes, C. We also require that at any given moment, each job $j \in J$ have a set of selectable modes modes of operation, M_j. A typical example of such a system might be the remote-controlled civilian or military craft depicted in Figure 1. The *video* job is responsible for video acquisition and compression, which can be carried out at various frame rates and resolutions, using any of several compression algorithms. The *navigation* job periodically determines the craft's current position and motion at appropriate rates. The *communication* job transfers video and navigational information to the remote control system via radio, while catering to various interference levels by using appropriate redundancy and error correction techniques. The *guidance* job controls locomotion actuators to comply with remote control directives at various speeds. Rates, resolutions, levels, speeds, and techniques are all dictated by any number of selectable modes per job.

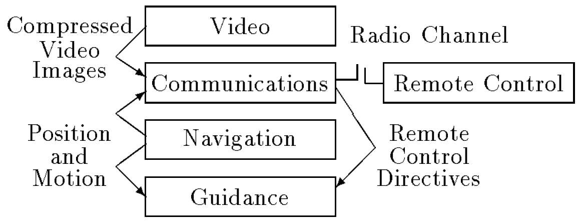

Fig. 1. A Remote-Controlled Craft

Besides requiring different processing capacities, each mode also clearly provides a different quality of service, referred to as the *job reward*, $r_j(m), m \in M_j$. Thus, for example, a video mode with higher frame rates ' and resolutions will have a higher reward than other video modes. The total *system value* is assumed to be a function of the current job rewards. For *best-effort adaptation*, job modes must be automatically selected in a manner which maximizes the system value without overloading available resources. Letting $m_j \in M_j$ represent the currently selected mode for job j, then the current system value, v, is assumed to be a function of the set of all currently selected

[2]Each job might be a pre-existing software package, or any self-contained system aggregate, naturally or artificially defined.

job rewards, $\{r_j(m_j)\}$. The workload associated with each job mode, $m \in M_j$, comprises an arbitrary task set, $\tau_j(m)$, which may also be *complex* in the sense that it might have to simultaneously cater to several dimensions of real-time systems (Stankovic and Ramamritham, 1990), e.g. state-dependent execution times and job priorities, tight and loose time constraints, hard and soft deadlines, periodic and aperiodic tasks, and various forms of inter-task dependencies. To be reliably scheduled, we require that each job mode be already equipped with an arbitrary real-time scheduling policy, $\pi_j(m)$, which can be deterministic or speculative, static (off-line scheduling) or dynamic (on-line scheduling). Common *deterministic* schedulers for *hard* real-time task sets include the *static* cyclic executive (CE) (Locke, 1992), and the *dynamic* rate monotonic (RM) and earliest deadline first (EDF) (Liu and Layland, 1973) policies. When dealing with *semi-hard* and *soft* real-time tasks, we might prefer more *speculative* best-effort scheduling policies, e.g. those of Locke (1986) and Miller (1990), which can provide higher value-oriented performance. Each $\pi_j(m)$ policy must also determine the minimal processing capacity required by it to either satisfy all $\tau_j(m)$ requirements (in a deterministic approach), or to obtain a given *expected* value (when using a speculative approach). This minimal capacity is henceforth referred to as the job *bandwidth*, $b_j(m)$, often measured by the minimal required processor clock frequency in MHz units.

Once mode m_j has been selected for job j, all job tasks in each $\tau_j(m_j)$ must share the same processor, but entire job task sets may *migrate* from one processor in C to another. Job *load-balancing* therefore comprises of finding a a partitioning of J into $|C|$ disjoint job subsets, $\{J_c\}$, $J = \bigcup_c J_c$, such that all jobs $j \in J_c$ are schedulable on processor c. Letting β_c denote the processing speed or *capacity* or processor c, and letting $b_j(m_j)$ denote the currently selected bandwidths for each job, it can be shown (Jehuda and Koren, 1995) that a job set, J_c, is *jointly* schedulable on processor c by an appropriate real-time time-sharing scheme, if

$$\sum_{j \in J_c} b_j(m_j) \le \beta_c. \qquad (1)$$

Thus ten jobs can reliably time-share a processor with a 10 MHz capacity if each job requires a 1 MHz bandwidth.

To efficiently accommodate the dynamic needs of the system, we require a variety of job modes with a wide range of bandwidths. Mode workloads should also be migratable, i.e. node-independent, whenever possible.

Alternate job modes can be used to encapsulate a wide variety of adaptation forms. Common forms of adaptation include *approximation* techniques which adjust time constraints to supportable levels (Gopinath and Schwan, 1989), *imprecision* techniques which use sieve functions to produce intermediate results when time runs outs (Liu et al, 1991), and *polymorphic* techniques which provide alternate methods for accomplishing specific tasks (Kenny and Lin, 1991), or use different scheduling policies (Ramamritham and Stankovic, 1991). Modes which require hard guarantees will use deterministic layer-by-layer methodologies to determine worst-case job bandwidths for $\tau_j(m)$ using $\pi_j(m)$. Other job modes might use smaller bandwidths to provide less deterministic accuracy or softer speculative guarantees.

Also to be considered are internal job *states* which may affect current job rewards and bandwidths. In the inertial navigation subsystem, for example, the rate of navigational positioning must be higher when the craft travels faster, thereby requiring a greater bandwidth. In a similar fashion, dynamic rewards may be used to reflect varying job priorities under various circumstances, e.g. navigation might be more important than video once a final destination has been determined. The set of active jobs, J, may also vary, e.g. the remote-controlled craft may also have radar capabilities which are not always active. Runtime *platform* characteristics may also vary with time, as when processing nodes are upgraded, when they fail, or when certain processing resources are diverted to activities outside the application. Mode selection and load balancing decisions must therefore be reconsidered with each significant transition in the set of active jobs, in the internal job states, in the external environment, and in platform processing capacities.

3. Previous Work

The primary contribution of the job-oriented top-layer architecture is in its ability to address several formidable issues, *simultaneously*: complex task sets, hard multi-processor schedulability, best-effort adaptability, uncertain environments, fault-tolerance, non-deterministic platforms, source-level portability, and scalability. Several subsets of these issues have been treated in the past. Surveying them all would be beyond the scope of this paper, so we mention here only a few and refer to (Jehuda and Berry, 1995) for more details. Jo (1994) and Ramamritham et al (1990), respectively, have devised centralized and distributed approaches to dynamic load-balancing. A variety of effective adaptation techniques have already been employed in the experimental Spring (Ramamritham and Stankovic, 1991), RESAS (Bihari and Schwan, 1988), GEM (Schwan et al, 1987), and Concord (Liu et al, 1991) projects. Unfortunately, none of these solutions can adequately support the integrated needs of our program model.

We suggest that the primary difficulty in all of these examples is that they attempt to tackle real-time scheduling, dynamic load-balancing, and best-effort adaptation at the *individual* task level, using *uniform* scheduling policies. Doing so at the task level is simply too complex. When assuming a uniform scheduling policy, even a small number of hard tasks forces us to making inefficient worst-case assumptions everywhere, all the time.

Rather than attempt to online generate new scheduling solutions for dynamic sets of individual tasks, we propose a *job*-oriented approach in which each job must already be equipped with an arbitrary deterministic or speculative scheduling policy which maximizes processor utilization for the tasks in that job. The software control architecture can then use a *divide and conquer* strategy to simultaneously facilitate real-time time-sharing, dynamic load balancing, and best-effort adaptations for arbitrary sets of self-contained jobs. As far as we know, the architecture is unique in its generalized support for hybrid scheduling solutions[3], and in its ability to best-effort adapt to arbitrary circumstances while guaranteeing predictable real-time performance for *all* active tasks. Current heuristics for best-effort adaptations can already respond within milliseconds while typically providing average 95%-98% performance levels. Job-level adaptations also enable efficient accommodation of varying sets of critical tasks as well as tasks with varying criticalities. When load-shedding is necessary, it can be done with a global perspective of requirements and rewards, rather than resorting to arbitrary FCFS arbitration.

4. RtTS Software Control

The *job*-oriented program model of Section 2 enables encapsulating internal complexities and adaptation alternatives in a manner which can be efficiently managed by the RtTS software control architecture. Real-time time-sharing, dynamic job load balancing, and high-value mode selection can then be carried out for an arbitrary set of jobs, J, on an arbitrary set of processors, C, in an integrated manner. An *integrated* approach is essential to critical real-time systems because decisions in each of realms are influenced by each other, e.g. independent mode selection may produce a set of critical tasks which cannot be load-balanced, and a load-balancing decision may generate job subsets per processor which cannot be reliably scheduled.

Control Layer	Level	Function	Typical Rates
Policy-Making	4	when? method? scope? depth?	mins
Automated Meta-Control	3	mode selection & load balancing	secs
Real-Time Time-Sharing	2	job time-sharing	msecs
	1	job scheduling	

Fig. 2. Time-Sharing Architecture

4.1 Real-Time Time-Sharing

For each job $j \in J$, each job mode $m \in M_j$, is internally responsible for the scheduling of its own task set, $\tau_j(m)$, using an arbitrary real-time scheduling policy, $\pi_j(m)$, found to be adequate when running on a processor with a minimal processing bandwidth, $b_j(m)$. Only one job mode in M_j can be active at any given moment, so we use $m_j \in M_j$ to denote the current job j mode. As depicted in Figure 2, the internal $\pi_j(m_j)$ job schedulers constitute the bottom control layer in the architecture. Each bandwidth can be viewed as a *virtual* processor with a processing speed equal to $b_j(m_j)$, so any J_c job set can reliably time-share a processor c with a capacity of β_c, subject to Equation (1). The job time-sharing control layer carries out this time-sharing for each job set J_c on each processor $c \in C$ using an EDF interleaving policy. Each $\pi_j(m_j)$ job scheduler keeps the time-sharing layer up-to-date regarding the earliest pending deadline, d_j, within its $\tau_j(m_j)$ task set. The time-sharing layer on each processor c always dispatches the job $j \in J_c$ with the closest d_j value. Whenever Job $j \in J_c$ is dispatched, the job scheduling of the $\tau_j(m_j)$ task set is carried out in accordance with the internal $\pi_j(m_j)$ scheduling policy. The outcome is that all $\pi_j(m_j)$ schedules are essentially *interleaved* without being altered or violated. The $b_j(m)$ bandwidths are computed in a manner which also accommodates worst-case time-sharing overheads[4]. Together, the bottom two layers facilitate a real-time time-sharing scheme which enables each job to use an arbitrary and independent real-time scheduling policy to best meet its complex requirements.

It follows from Equation (1) that this time-sharing scheme essentially maintains[5] the processor utilization provided by each $\pi_j(m_j)$ scheduling policy within its own bandwidth. Thus, a primary design objective in a time-sharing architecture is to decompose the system into high utilization jobs, i.e. jobs with task sets which have high utilization scheduling solutions for them. Another design objective is to choose a scheduling policy for each mode which will minimize the bandwidth (maximize the utilization) for that mode. Thus a static cyclic executive scheduler will usually be applied to a set of strictly periodic tasks, for which this method can successfully avoid resource contentions, untimely preemptions, and unnecessary context switching overheads. Dynamic schedulers will usually be applied to task sets which cannot be efficiently accommodated by static scheduling, e.g., tasks with irregular arrival times, and tasks which are not readily broken up into cyclic time frames. More advanced systems might eventually support *automatic scheduling*, whereby the $\pi_j(m)$ scheduling policy for each job mode is determined automatically to minimize the bandwidth. Once this is accomplished, we are guaranteed that the time-shared processor utilization will be equally high.

[3] Young and Shu (1991) have proposed a *hybrid* scheduling scheme for dealing efficiently with complex task sets, but only CE and RM are reconciled, and it is done in a manner which cannot facilitate dynamic load balancing.

[4] Worst-case time-sharing overheads are shown to be of $O(N)$ complexity where N denotes the maximum number of jobs in J_c, so that b_j bandwidths can be computed for arbitrary J_c job sets and for any combination of π_j scheduling policies.

[5] Utilization losses due to time-sharing overheads are typically low and they are usually compensated for by reduced average losses due to capacity fragmentation (Jehuda et al, 1995).

4.2 Automated Meta-Control

Above the bottom two layers we find the *meta-control* layer which must seek the most profitable job mode selection and job load-balancing for a given system state. The primary objective of the automated meta-controller is to select job modes $m_j \in M_j$ which will produce the *optimal* system value, v^*, i.e. the highest obtainable system value which is still schedulable. Let J_c represent a job subset allocated to processor $c \in C$, and let β_c denote the processor c capacity. When using a time-sharing scheme, a given selection of job modes is *schedulable* if and only if there exists a partitioning of the job set J over the set of available processors C such that

$$\forall c \in C : \sum_{j \in J_c} b_j(m_j) \leq \beta_c. \qquad (2)$$

Finding such a schedulable partitioning is the job of the load-balancer, whereupon fully reliable time-sharing is guaranteed for all jobs in J.

A meta-controller *decision* consists, therefore, of a set of selected modes and a schedulable partitioning of jobs over processors. Job sets are dynamic. Job bandwidths and rewards are state-dependent. To support *fault-tolerance*, processor capacities, β_c, may also vary. Thus meta-controller decisions must be automatically made with each altered job set, each state transition, and each change in the current platform capacity. It can be shown that the meta-control problem is equivalent to a composite binpacking and zero-one multiple-choice knapsacking problem which is strictly NP-hard. Nevertheless, we have devised two very effective approximation algorithms, QDP and G^2, each of them typically requiring only fractions of a second to produce system values which are rarely suboptimal by more than a few percent. These results enable us to reevaluate our decisions every few seconds to accommodate varying job sets and states with reasonable overhead. QDP relies primarily on dynamic programming, while G^2 is gradient-greedy. G^2 performance is generally only sightly lower than QDP, and both provide significant improvements of $20\% - 40\%$ over conventional density greedy (Garey and Johnson, 1979) methods. Both algorithms also provide low time and space complexities so that the RtTS architecture is indeed *scalable* to large-scale systems. G^2 advantages include being an iterative *sieve* function, i.e. it always has a valid intermediate decision which only gets better with time. QDP also provides a simple mechanism for trading of precision for quicker response times. An appropriate analytic model has been devised for predicting *expected* QDP system values for any given system. For detailed descriptions of these algorithms, simulation results, and analytic methods, the interested reader may refer to (Jehuda and Israeli, 1994) and (Jehuda, 1995).

4.3 Policy Making

We recall that job mode bandwidths and rewards may be internally *state*-dependent, so jobs must notify the upper control layers when an internal state transition is about to occur. When optimizing system value for a given situation, the system control architecture must also consider the dynamics of migrating jobs, changing modes, and other short-term consequences, such as a possible loss of system stability due to frequent adaptations. It is the *policy-maker*'s responsibility to consider all anticipated possibilities in determining *when* automated meta-control decisions should be made, which *method* should be used, what *scope*, i.e. which jobs, processors, and mode sets should be considered, and to what *depth*, i.e. how fast and accurate each decision must be. An automobile driver, for example, will always slow down as he approaches a crossing or any other situation which might require a sudden change of speed and direction. In a similar manner, the policy maker of the remote-control craft may also take precautionary measures, e.g. allocating a larger job bandwidth than necessary when a state-transition is imminent, or preparing a set of instant local adaptations for emergencies. The policy making layer must clearly be tailored to the specific nature of each system. Nevertheless we anticipate that a well-defined set of evolving policy-making control models, e.g., Markov-decision processes, (Jehuda, 1994), will eventually handle enough applications to warrant packaging in a generic format.

5. A Top-Layer Architecture

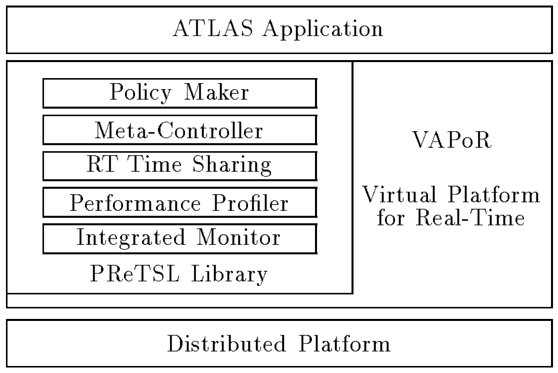

Fig. 3. The Top-Layer Architecture

Another significant design objective for any system is the ability to accommodate *long-term* adaptations, e.g. evolving run-time platforms and program specifications which change with time and experience. As illustrated by Figure 3, we accomplish this by extending the RtTS software control model, to produce a *top-layer* architecture, capable of supporting *platform-independent* real-time applications. The *performance profiler* is charged with providing the platform-dependent task attributes which are required for real-time scheduling. The integrated *monitor* primarily serves the performance profiler and it also facilitates system validation and evaluation. The VaPoR *virtual platform* supports appropriate abstractions which enable the provision of all necessary real-time services by a variety of conventional distributed platforms. All portable contingents of these components are contained in a portable real-time support library, PReTSL. Non-portable elements are isolated within the VaPoR layer which mediates between the application and the platform. Only the VaPoR layer must be reimplemented and customized to accommodate each candidate run-time platform.

5.1 The Virtual Platform

VaPoR is designed to serve as a *generic* virtual platform for real-time applications, and must therefore support a comprehensive set of operating system services typically required by such applications. As in Chaos (Schwan et al, 1990), ARTS (Tokuda and Mercer, 1989), RT Mach (Tokuda et al, 1990) and other systems (Mukherjee and Schwan, 1992), a major VaPoR premise is that real-time applications consists of *cooperating* processes which enable them to use lightweight threads to implement concurrent tasks. On a RT Mach platform, for example, job tasks would be translated by VaPoR into native Mach threads. On a non-threaded Unix platform, VaPoR might incorporate C or POSIX threads for basic timing, thread synchronization, and message-passing. VaPoR provides a uniform interface to these services to maximize portability. As in MOSIX (Barak et al, 1991), and other message-passing environments (McBryan, 1994), VaPoR must be designed to efficiently and reliably deliver messages between node-independent jobs regardless of their current placements. VaPoR abstractions avoid dictating the mechanisms to be used by these services so that the current platform architecture can be best exploited[6].

Unlike most other experimental real-time platforms, the VaPoR approach attempts to accommodate widely accepted industry standards, rather than attempt to supersede them. VaPoR adopts an Ada-like strategy whereby various services are either internally resolved within user-level run-time libraries or leased out to the platform operating system in a manner transparent to the application. It also provides all the necessary hooks to the integrated monitor and profiler so that program and platform behavior can be monitored and profiled.

5.2 Performance Profiling

The *performance profiler* is charged with providing the RtTS control layers with all the platform-dependent characteristics which they require. To compute the minimal bandwidth $b_j(m)$ for a given platform, each job Mode m in Job j requires various task characteristics, e.g. typical or *worst case execution time* (WCET), for all tasks in $\tau_j(m)$. As illustrated by the Concord project and the Flex programming language (Kenny and Lin, 1991), safe WCET estimates can be produced and perfected by combining compile-produced data with accumulated run-time experience. The Concord project also shows how calibrated WCET functions can be obtained which provide tight WCET upper bounds for each task as a function of predetermined state variables, so that job mode bandwidths can be upgraded with each state transition. The timing analysis for each job also requires typical or WCETs for all VaPoR services used by each job. All of these characteristics must be provided by the performance profiler.

5.3 The Integrated Monitor

The integrated monitoring subsystem plays a very central role in a top-layer architecture, primarily serving system validation, performance analysis, performance profiling, and job scheduling. The integrated monitor relies on a dynamic combination of *probes* and *sensors*, embedded within the VaPoR, Pretsl, and program layers for, respectively, sampling variables and for tracing execution. A job scheduler using *milestoning* (Haban and Shin, 1989), for example, relies on implanted milestone sensors, which monitor task progress to appropriately reduce the WCET assumed for that task for more efficient online scheduling. All monitored data must be time-tagged, collected, filtered, routed, processed, and recorded, while adapting to the dynamic requirements and resources of the system. Integrated monitoring schemes have already been devised for several dynamic and distributed real-time systems, e.g. (Ogle et al, 1990).

6. Accommodating Nondeterminism

The VaPoR virtual platform acts as an interface to any of several candidate run-time platforms. Unfortunately, most popular platforms have components that do not comply with layer-by-layer requirements. Most popular high-level *languages*, have *ambiguous* interpretations, e.g. as with *case* and multiple conditions, or because they have *inherently* nondeterministic statements, e.g. Occam guard commands. Languages may also suffer from *implicit* nondeterminism, such as when Ada uses a *delay* statement to implement periodic tasks. Fault handling for arithmetic exceptions and the order of subexpression evaluation are non-deterministic in *all* programming languages.

Advanced hardware technologies, such as branch prediction, pipelined execution, cache memories, and virtual memories, are all problematic because they use non-deterministic statistical models to boost performance. Much of these performance gains are lost when always assuming the worst-case scenario for each element, as required by the layer-by-layer approach. For example, when carrying out a timing analysis for a program running in a cache memory, we always assume a cache miss for every memory access, resulting in a task WCET which can be ten times worse than the typical task execution times. This results in actual processor utilizations which are extremely low and wasteful. A top-layer architecture can better exploit the performance speed-ups of-

[6]Mach 3.0 with a Unix server was probably the first to demonstrate that a portable virtual platform can outperform even a dedicated non-portable platform (Ultrix 4.0) by using appropriate platform abstractions and by optimizing each instantiation (Chen and Bershad, 1993).

fered by such non-deterministic platform elements, by containing, reducing, and diffusing these utilization losses in the following ways.

As already proposed by Stnakovic and Ramamritham (1990), worst-case layer-by-layer timing analysis might be necessary only for certain subsystems when in specific states. The job-oriented program model enables us to apply different scheduling policies to each job mode in a state-dependent manner. This enables us to *contain* WCET assumptions within specific jobs, modes, and states. Rather than *always* assume worst-case scenarios in all sub-systems all the time, we can assume the worst-case only within critical subsystems, *when* they are critical. Even when necessary, assumed WCETs can be reduced based upon actual observed behavior, as when using the milestoning technique in Section 3.

We suggest that worst-case assumptions can be further *reduced* by reserving a surplus capacity on each processor for absorbing transient non-deterministic overloads, and by monitoring its consumption to determine when an emergency mode transition might be necessary to avoid an impending fault. Emergency mode transitions will always be made to lower bandwidth modes prepared in advance with each meta-controller decision, so that job load-balancing can ensure that an emergency transition is available in each J_c. Higher bandwidth modes provide higher system values, but we assume that a penalty is paid for each emergency mode transition. Therefore, surplus capacities must be substantial enough to guarantee that emergency mode transitions will be infrequent enough to justify the technique. Thus, surplus capacity consumption must be monitored, and a minimal surplus must be determined.

To facilitate such a technique, the performance profiler must differentiate between job-triggered transient overloads, e.g. disk I/O, and job-independent transient overloads, e.g. clock updates and mouse input. When dealing with job-independent transient overloads the same reserved capacity can serve any number of jobs sharing an given processor, c, so inefficiency due to worst-case assumptions is thus *diffused* among all jobs in J_c.

In another variation, each job distinguishes between hard and soft job bandwidths, and job load-balancing is carried out in a manner which guarantees that each processor has sufficient soft bandwidth to absorb the transient overloads. Work on these techniques has only begun.

7. Conclusions

The authors believe that the current top-layer program model and control architecture can already be applied, as is, to a very significant class of complex real-time applications, provided that the program can indeed be decomposed into several loosely-coupled jobs, many of them node-independent, with a significant number of job modes, and with a sufficient variety of mode bandwidths. All that we require of a platform is that it have capacities capable of accommodating the minimal bandwidth modes for the most demanding job set anticipated.

As previously mentioned, the primary contribution of the job-oriented top-layer architecture is in its ability to simultaneously address several formidable issues. The RtTS time-sharing scheme enables us to accommodate dynamic and complex task sets, while providing us with a very simple criteria for hard multi-processor schedulability. The RtTS meta-controller and policy maker facilitate automatic best-effort adaptation to arbitrary circumstances, making the architecture very appropriate for uncertain environments and fault-tolerant platforms. Low QDP and G^2 time and space complexities provide the scalability necessary for handling large-scale systems as well. The top-layer architecture also offers several possible techniques for better utilizing non-deterministic platforms.

Another major contribution is the inherent source-level *portability* provided by the VaPoR virtual platform and portable PReTSL library. Platform-independent real-time software can extend longevity and enhance reusability. Moreover, portability avoids the need for prohibitive software revisions when platforms are expanded or replaced to deal with inadequate performance, growing requirements, and obsolete hardware. Once portability is achieved, any given real-time problem should ultimately disappear as computers grow faster and more powerful. Portability can also improve reliability by allowing more extensive system validation on a variety of more suitable platforms.

Skeptics may argue that a major top-layer drawback is that it cannot fully exploit the architectural aspects of the platform. This might be true today, but we suggest that this is only a temporary obstacle. Similar arguments were used to delay the use of high-level languages in real-time software. As with high-level languages, we anticipate that advanced operating systems and optimization technologies will eventually provide generic, application-independent, solutions for maximizing the utilization of platform resources without explicit directives from the application. Even today, costs incurred by reduced efficiency can be relatively insignificant when considering other aspects of system development, deployment, and maintenance. As with low-level languages, the inadequacy of layer-by-layer designs will only grow more acute as real-time systems expand in size and complexity. If viable, a top-layer approach might be the only practical alternative for many next-generation real-time systems.

8. References

Barak, A., Guday, S., and Wheeler, R. G. (1991). *The MOSIX Distributed Operating System - Load Balancing for Unix*. Number 672 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1991

Bihari, T. and Schwan, K. (1988) A comparison of four adaptation algorithms for increasing the reliability of real-time software. *Proceedings Ninth Real-Time Systems Symposium*, March, 1988, 232–243.

Chen, J. B. and Bershad, B. N. (1993) The impact of operating system structure on memory system performance. *Proceedings of the 14th ACM Symposium on Operating System Principles*, December, 1993.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability - Guide to the Theory of NP-Completeness.* Bell Telephone Laboratories, Inc..

Gopinath, P. and Schwan, K. (1989) CHAOS: Why one cannot have only an operating system for real-time applications. *ACM Operating Systems Review* **23(3)**, 106–125.

Haban, D. and Shin, K. G. (1989) Application of real-time monitoring to scheduling tasks with random execution times. *Proceedings Ninth Real-Time Systems Symposium*, December, 1989, 172–181.

Halang, W. A. and Stoyenko, A. D. (1991). *Constructing Predictable RT Systems.* Kluwer Academic Publishers.

Jehuda, J. and Berry, D. M. (1994). *A top-layer architecture for ATLAS.* Anonymously available at `ftp.technion.ac.il` in `/pub/supported/ee/Computers` as `eepub946.ps`.

Jehuda, J. and Berry, D. M. (1995). *A Top Layer Design Approach for Adaptive Real-Time Software.* Anonymously available at `ftp.technion.ac.il` in `/pub/supported/ee/Computers` as `eepub966.ps`.

Jehuda, J. (1994). *Markovian thoughts on software meta-control.* Anonymously available at `ftp.technion.ac.il` in `/pub/supported/ee/Computers` as `eepub944.ps`.

Jehuda, J. (1995). *An analytic model for estimating QDP performance.* Anonymously available at `ftp.technion.ac.il` in `/pub/supported/ee/Computers` as `eepub947.ps`.

Jehuda, J. and Israeli, A. (1994). *Automated meta-control for adaptable real-time software.* Anonymously available at `ftp.technion.ac.il` in `/pub/supported/ee/Computers` as `eepub943.ps`.

Jehuda, J. and Koren, G. (1995). *Hybrid bandwidth scheduling for distributed real-time systems.* Anonymously available at `ftp.technion.ac.il` in `/pub/supported/ee/Computers` as `eepub945.ps`.

Jehuda, J., Koren, G., and Berry, D. M. (1995). *A time-sharing architecture for complex real-time systems.* Anonymously available at `ftp.technion.ac.il` in `/pub/supported/ee/Computers` as `eepub965.ps`.

Kenny, K. B. and Lin, K.-J. (1991) Building flexible real-time systems using the FLEX language. *IEEE Computer* **24(5)**, 70–78.

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

Liu, J. W., Lin, K.- J., Shih, W.- K., Yu, A. C., Chung, J.-Y., and Zhao, W. (1991) Algorithms for scheduling imprecise computations. *IEEE Computer* **24(5)**, 58–68.

Locke, C. D. (1986). *Best-Effort Decision Making for Real-Time Scheduling.* Phd Dissertation, Carnegie-Mellon University, Dept. of Computer Science, Pittsburg, Pa. 15213. May, 1986

Locke, C. (1992) Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Journal of Real-Time Systems* **4(1)**, 37–54.

McBryan, O. A. (1994) An overview of message passing environments. *Parallel Computing* **20**:417–444.

McElhone, C. (1994). *Adapting and evaluating algorithms for dynamic schedulability testing.* Technical Report, Department of Computer Science, University of York, England. February, 1994

Mok, A. K. and Dertouzos, M. L. (1978) Multiprocessor scheduling in a hard real-time environment. *Proceedings of the Seventh Texas Conference on Computing Systems*, November, 1978.

Miller, F. W. (1990) Predictive deadline multi-processing. *ACM Operating Systems Review* **24(4)**, 52–62.

Mukherjee, B. and Schwan, K. (1992). *A survey of multiprocessor operating system kernels.* Technical Report, GIT-CC-92/05, College of Computing, Georgia Institute of Technology. January, 1992

Oh, Y. (1994). *The Design and Analysis of Scheduling Algorithms for Real-Time and Fault-Tolerant Computer Systems.* Phd Dissertation, University of Virginia. May, 1994

Ogle, D., Schwan, K., and Snodgrass, R. (1990). *The dynamic monitoring of real-time distributed and parallel systems.* Technical Report, GIT-ICS-90/23, College of Computing, Georgia Institute of Technology. May, 1990

Ramamritham, K., Stankovic, J. A., and Shiah, P.-F. (1990) Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems* **1(2)**, 184–194.

Ramamritham, K. and Stankovic, J. A. (1991). *Scheduling strategies adopted in SPRING.* Technical Report, University of Massachusetts. COINS 91-45, 1991

Schwan, K., Bihari, T., Weide, B., and Taulbee, G. (1987) High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems* **5**:189–231.

Schwan, K., Gheith, A., and Zhou, H. (1990) From CHAOS-min to CHAOS-arc: A family of real-time kernels. *Proceedings of the Real-Time Systems Symposium*, December, 1990, 82–92.

Stankovic, J. A. and Ramamritham, K. (1990) What is predictability for real-time systems?. *Journal of Real-Time Systems* **2(4)**, 247–254.

Stankovic,J.A. (1988). Real-Time Computing Systems: The Next Generation. In: Stankovic, J.A. and Ramamritham, K. (Ed.), *Tutorial: Hard Real-Time Systems*, 14–37. IEEE Computer Society Press.

Tokuda, H. and Mercer, C. W. (1989) ARTS: A distributed real-time kernel. *ACM Operating Systems Review* **23(3)**, 29–53.

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

Young, M. and Shu, L.-C. (1991). *Hybrid online/offline scheduling for hard real-time systems.* Technical Report, SERC-TR-100-P, Software Engineering Research Center, Department of Computer Sciences, Purdue University. May, 1991