

Scheduling for Reliable Execution in Autonomic Systems^{*}

Terry Tidwell, Robert Glaubius, Christopher Gill, and William D. Smart

Department of Computer Science and Engineering
Washington University, St. Louis, MO, USA
{ttidwell, rlg1, cdgill, wds}@cse.wustl.edu

Abstract. Scheduling the execution of multiple concurrent tasks on shared resources such as CPUs and network links is essential to ensuring the reliable operation of many autonomic systems. Well known techniques such as rate-monotonic scheduling can offer rigorous timing and preemption guarantees, but only under assumptions (i.e., a fixed set of tasks with well-known execution times and invocation rates) that do not hold in many autonomic systems. New hierarchical scheduling techniques are better suited to enforce the more flexible execution constraints and enforcement mechanisms that are required for autonomic systems, but a rigorous foundation for verifying and enforcing concurrency and timing guarantees is still needed for these approaches. The primary contributions of this paper are: (1) a scheduling policy design technique that can use different decision models across a wide range of systems models, and an example of how a specific (Markov Decision Process) decision model can be applied to a basic multi-threaded system model; (2) novel model checking techniques that can evaluate the behavior of the system model when it is placed under the control of the resulting scheduling policy; and (3) an evaluation of those scheduling policy design and model checking techniques for a simple but representative example of the kinds of execution scenarios that can arise in autonomic systems.

1 Introduction

An autonomic computing system must respond adaptively to varying operating conditions, automatically and without external intervention. The adaptive behaviors that allow such a system to continue to perform under dynamic conditions in turn place varying demands on shared system resources, and the capacities of the system's resources constrain the possible behaviors of the system. Furthermore, to verify that an autonomic computing system can manage its resources both feasibly and adaptively at run-time, checkable models of the interactions among (1) the system's resource management policies and mechanisms,

^{*} This research was supported in part by NSF grant CNS-0716764 (Cybertrust) titled "CT-ISG: Collaborative Research: Non-bypassable Kernel Services for Execution Security" and NSF grant CCF-0448562 (CAREER), titled "Time and Event Based System Software Construction".

(2) the system’s resources, and (3) the adaptive demands that system activities place on the resources, must be developed. How to ensure reliable execution of autonomic computing system activities is thus an important and challenging research problem.

Existing approaches to ensuring the verifiably feasible use of system resources on-line often employ some kind of *reference monitor* [1], which mediates all requests for access to system resources according to specified policies. Although reference monitors have been considered most extensively in the contexts of data security and network security, separation kernels [2, 3] for partitioning resource use, and user level sandboxes [4–7] that intercept system calls made by application programs, illustrate the applicability of resource monitors to managing the execution of system activities.

Limitations of Existing Approaches: As we discuss in further detail in Section 2, while the user level sandbox and separation kernel approaches offer important features for ensuring feasible use of resources by system activities, each of the approaches has important limitations. For the sandbox approach the crucial limitation is in how precisely (especially with reference to timing) the desired execution semantics can be enforced, while for the separation kernel approach the limitation is the burden placed on system developers to encode the nuances of complex system dependences according to strict resource separation semantics.

Real-time schedulers [8] offer what amounts to a kind of (admittedly bypassable) resource monitor by ensuring resource feasibility of a set of system tasks. Although they can offer strong guarantees under non-adversarial conditions, such classical scheduling approaches only apply under very constrained assumptions that do not pertain in many autonomic computing contexts. Hierarchical schedulers [9–12] offer greater flexibility in enforcing less constrained scheduling policies precisely, and our previous work has shown that integrating hierarchical thread-level scheduling mechanisms within a kernel-level resource monitor is a useful step towards non-bypassable control over the execution of system activities [13, 14]. However, rigorous analysis of these more advanced scheduling approaches remains a largely open problem, so that for the most part analytical guarantees of resource feasibility under those policies currently are not available. Furthermore, it is difficult to apply standard verification techniques such as model checking without exploiting knowledge about the specific scheduling policy, which we have also investigated in our prior work [15].

Solution Approach and System Model: To overcome the limitations of existing approaches for ensuring the verifiably feasible use of system resources, we are developing new techniques (1) that are flexible in the policies they can enforce, and (2) within which particular resource monitors can be customized

according to their intended use. In this paper, we consider only a very basic and abstract system model in which:

- multiple threads of execution require mutually exclusive use of a single common resource (i.e., a CPU) in order to run;
- whenever a thread is granted the resource, it occupies the resource for a finite and bounded subsequent duration;
- the duration for which a thread occupies the resource may vary from run to run of the same thread but overall obeys a known independent and bounded distribution over any reasonably large sample of runs of that thread;
- a scheduler initially chooses which thread to run according to a given scheduling policy, dispatches that thread, waits until the end of the duration during which the thread occupies the resource, and then repeats that sequence perpetually.

This basic system model serves to illustrate simple but representative examples of the kinds of scheduling enforcement problems that can arise in autonomous systems built atop commonly used operating systems such as Linux or VxWorks. For example in Linux every dispatch of an application thread occupies the CPU for at least a jiffy and the scheduler only preempts threads at jiffy boundaries. Within the Linux kernel, our previous work has considered how hard and soft interrupts also may be threaded and placed under scheduler control [15], with different resulting durations of resource occupation for the different kinds of interrupts.

In Section 3 we present a method for scheduling policy design that can be tailored to specified workloads, which is based on a Markov Decision Process (MDP) approach. The MDP approach is an illustrative example of a more general class of scheduling policy design approaches that could be used in our solution approach, though we defer consideration of other relevant techniques, such as reinforcement learning, to future work. In Section 4 we present a novel model checking approach that makes use of finite execution histories. This approach can be used for exhaustive exploration of possible system traces, to verify properties such as the feasible use of resources under a scheduling policy designed according to the approach in Section 3. In Section 5 we evaluate the application of our approach to a sample system configuration, based on threads being scheduled to maximize adherence to a target utilization for each thread. The results of this evaluation show that these techniques can be practically applied. Finally, in Section 6 we summarize the contributions of this paper, and describe planned future work.

2 Related Work

Reference Monitor Approaches: User-level sandboxes have been used to intercept system call requests and may record, deny, reorder, replace, or dispatch any request. This approach offers significant flexibility because all system calls can be subjected to arbitrary handling by the sandbox. However, sandboxes that do this entirely within user space have difficulty supporting standard features like safe and efficient multi-threading [5]. Hybrid interposition architectures [6] therefore move part of the sandbox into the kernel. However, this approach still relies on the kernel's native scheduling policies and mechanisms, which do not offer sufficient control over system components such as interrupts [14], and thus leave system activities vulnerable to accidental or adversarial interference through interaction channels (such as resources shared among threads) that do not pass explicitly through the system call interface.

Separation kernels can provide more stringent enforcement of system policies, but unfortunately existing approaches do so inflexibly, by segregating resources into discrete partitions, and strictly controlling communication and other interactions among different partitions [2, 3]. For example, the MILS kernel [3] partitions memory and CPU resources into separate virtual machines on which processes then execute, controlling not only access to resources, but also communication between processes running in different partitions. Through such strict separation, these approaches allow formal specification and verification [16] of resource isolation between the partitions.

The main limitation of existing separation kernel approaches is that application developers must assign processes to resource partitions correctly, so that independent system activities are isolated, but system activities that have inherent dependences can still interact appropriately. This obligation places a significant burden on system designers, and examples of non-adversarial interference between activities of complex autonomous systems, such as the Mars Pathfinder priority inversion problem [17], illustrate that identifying all dependences up front in real-world systems is a daunting task.

Scheduling Policy Design: Many thread scheduling policies have been designed and analyzed to ensure guaranteed feasibility of resource use in closed real-time systems [8]. Most of those approaches assume that the number of tasks accessing system resources, and their invocation rates and execution times, are all well characterized. Real-time systems approaches that allow even such basic extensions such as asynchronous task arrival must depend on special services (e.g., admission control [18]) to maintain resource feasibility at run-time.

Hierarchical scheduling techniques [9–12] offer greater flexibility in their ability to enforce scheduling policies adaptively at run-time, according to multi-

faceted scheduling decision functions that are arranged hierarchically into a single system scheduling policy. However, there has been little prior work on verification of what guarantees can be made by such hierarchical scheduling policies. Furthermore, verification of scheduling policies that induce thread pre-emption and require reasoning about continuous time may encounter problems with decidability [19], so that special techniques that exploit knowledge about the structure of the specific scheduling problem [15, 20] may be needed before the techniques we are developing can be applied to systems with more nuanced execution semantics than the basic system model described in Section 1 (e.g., systems in which an actuator or sensor could be triggered arbitrarily on a continuous time line).

Dynamic programming has long been used for large-scale scheduling problems, such as those encountered in large machine shops [21]. A related technique, Reinforcement Learning (RL) [22] (often called Approximate Dynamic Programming), has been identified as a learning technology that holds great promise for the autonomic computing community [23]. It has been successfully applied to several domains, including computer cluster management [24] and network configuration repair [25], and job scheduling [26]. However, RL algorithms are typically iterative and, in practice provide an approximation to the optimal solution. This approximation improves over time, as the algorithm sees more training data but, for realistic problems, convergence to the optimal is often slow.

3 Scheduling Policy Design

The scheduling decision model consists of sequentially deciding to dispatch one of n threads whenever the CPU is available. Threads may release the CPU after a non-deterministic duration, that as we noted in Section 1 falls within a known and bounded distribution. A dispatched thread always executes for at least one time quantum. The scheduler’s objective is to maintain the relative resource utilization for each thread near some target utilization vector \mathbf{u} .

We represent this scheduling decision model as a Markov Decision Process (MDP) [27]. In general, an MDP is a four-tuple (X, A, R, T) . X is the set of process states, and A is the set of available actions. The transition function T describes the dynamics of the system as a conditional probability measure $P(y|x, a)$ of transitioning from state x to y on action a . The real-valued reward function $R(x, a, y)$ describes the immediate cost or benefit for transitioning from state x to y on action a . In the discounted reward setting, future rewards are weighted by a factor of $\gamma \in (0, 1]$, which weights rewards by temporal proximity.

A policy π recommends an action in each state. An optimal policy, π^* , maximizes the expected sum of discounted rewards observed as the system executes. Finding π^* reduces to computing the optimal state-action value function Q^* . $Q^*(x, a)$ is exactly the sum of discounted rewards obtainable by taking action a from state x , then executing the optimal policy thereafter. $Q^*(x, a)$ is the solution to the system of Bellman equations

$$Q^*(x, a) = \sum_{y \in X} P(y|x, a) [R(x, a, y) + \gamma V^*(y)], \quad (1)$$

where $V^*(x) = \max_{a \in A} \{Q^*(x, a)\}$ is the optimal state value function. Given Q^* , $\pi^*(x) = \operatorname{argmax}_{a \in A} \{Q^*(x, a)\}$.

In the scheduling MDP, each action corresponds to the choice to dispatch a particular available thread. The MDP's states are identified by the time quanta utilized by each thread. These are integer-valued vectors $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{N}^n$ for a system with n threads. In order to bound the number of states, we introduce a termination time τ , so that the state set $X = \{\mathbf{x} \in \mathbb{N}^n : \|\mathbf{x}\| \leq \tau\}$ where $\|\cdot\|$ denotes the 1-norm. We treat the boundary states \mathbf{x} such that $\|\mathbf{x}\| = \tau$ as *absorbing states*, so that further actions do not change the state of the system. The parameter τ defines the extent to which the scheduler looks into the potential future evolutions of the system's execution state when making a decision.

The MDP's transition function is defined in terms of the run-time distribution for each thread. Let $\Delta_i = (\delta_{i1}, \dots, \delta_{in})$ be the change in state after thread i executes for a single time quantum; δ_{ij} is the Kronecker delta ($\delta_{ij} = 1$ when $i = j$, otherwise $\delta_{ij} = 0$). The transition probability of the system moving from state \mathbf{x} to state \mathbf{y} after dispatching thread i can be non-zero only when \mathbf{y} and \mathbf{x} differ only in element i , i.e., only when $\mathbf{y} = \mathbf{x} + t\Delta_i$ for some positive integer t . If \mathbf{y} is non-absorbing, the transition probability is exactly $P_i(t)$, the probability that thread i executes for t time quanta. If \mathbf{y} is absorbing, then $P(\mathbf{y}|\mathbf{x}, i)$ is the cumulative probability of executing for t or more time steps, $\sum_{s=t}^{\infty} P_i(s) = 1 - \sum_{s=1}^{t-1} P_i(s)$. To summarize,

$$P(\mathbf{y}|\mathbf{x}, i) = \begin{cases} P_i(t) & \exists t > 0, \mathbf{y} = \mathbf{x} + t\Delta_i \text{ and } \|\mathbf{y}\| < \tau \\ 1 - \sum_{s=1}^{t-1} P_i(s) & \exists t > 0, \mathbf{y} = \mathbf{x} + t\Delta_i \text{ and } \|\mathbf{y}\| = \tau \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

We define the reward function $R(\mathbf{x}, i, \mathbf{y})$ in terms of a per-state cost function C . The cost of a state \mathbf{x} , $C(\mathbf{x})$, is the squared Euclidean distance between \mathbf{x} and the target utilization at time $\|\mathbf{x}\|$, $\|\mathbf{x}\|\mathbf{u}$:

$$C(\mathbf{x}) = - \sum_{i=1}^n (x_i - u_i \|\mathbf{x}\|)^2. \quad (3)$$

Since actions only change one component of the state vector, we define $R(\mathbf{x}, i, \mathbf{y})$ only when $\mathbf{y} = \mathbf{x} + t\Delta_i$ for some t . In order to encourage the scheduling policy to maintain target utilizations while threads execute as well as when decisions are made, we define the reward as the discounted sum of the costs of states from \mathbf{x} to \mathbf{y} , excluding \mathbf{y} .

$$R(\mathbf{x}, i, \mathbf{y}) = R(\mathbf{x}, i, \mathbf{x} + t\Delta_i) = \sum_{s=0}^{t-1} \gamma^s C(\mathbf{x} + s\Delta_i) \quad (4)$$

Figure 1 depicts the *utilization state space* and its transition function for a problem with two threads and a termination time of three quanta. Each thread in this example has a deterministic run-time of one quantum.

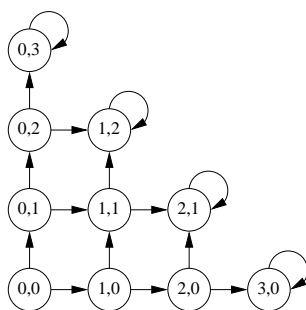


Fig. 1. Transition graph for a scheduling MDP with $\tau = 3$ and two threads. Each thread has a deterministic single quantum run time. Right arrows indicate the change in state as thread 1 runs, up arrows show the state transition when thread 2 runs.

Excluding absorbing states, the scheduling MDP transition graph is acyclic. The future expected rewards of states depend only on states with greater cumulative utilization. This enables us to solve for the value function directly by working backwards from the absorbing states, as long as n and τ are sufficiently small.

We first compute the future expected reward of each absorbing state \mathbf{x} . The future expected rewards of these states are the costs of remaining in them forever,

$$V^*(\mathbf{x}) = - \sum_{t=0}^{\infty} \gamma^t C(\mathbf{x}) = -C(\mathbf{x})/(1 - \gamma). \quad (5)$$

Next we iterate over non-absorbing states, working backwards from states with high to low utilization. Let $T = \tau - \|\mathbf{x}\|$ be the number of remaining quanta before termination from one such state. The future expected reward of dispatching

thread i in state \mathbf{x} is

$$Q^*(\mathbf{x}, i) = \sum_{s=1}^T P(\mathbf{x} + s\Delta_i | \mathbf{x}, i) [R(\mathbf{x}, i, \mathbf{x} + s\Delta_i) + \gamma V^*(\mathbf{x} + s\Delta_i)]. \quad (6)$$

Computing the value function in this case takes $\mathcal{O}(n\tau|X|)$ time. The τ term arises because the future expected reward of a state is the weighted average over possible future states. Computing V^* in the recursion requires maximizing over all n actions from each potential future state. If we know that a thread can only occupy the resource for at most k time steps, then we can replace the τ term with k by restricting the summation in Equation 6 to only the possible run-times. In this paper we consider only problems where n and τ are small enough to allow exact computation of the value function as detailed above. The number of states grows quickly, as $|X| = \sum_{t=0}^{\tau} \binom{n+t-1}{n-1}$, so eventually we would need to approximate the value function as n and τ increase.

4 Verification

Model checking has been applied to the offline verification of a wide range of systems. Model checking verifies systems by first exhaustively enumerating all reachable states and the transitions among them. Specifically, given a transition and a predecessor state, the next state represents the possible values the system variables can take on. To differentiate these states from the *utilization states* in the MDP described in Section 3, we call these states *verification states*. The verification states capture the *possible* evolutions of the system’s utilization state.

Safety properties are specified as temporal logic expressions evaluated over the verification state space. A system is verified if, during exhaustive enumeration, no verification state is found where that expression is false. In this paper we do not detail how to evaluate particular temporal logic expressions, but rather describe the strategies for the exhaustive enumeration of the verification state space induced by a particular scheduling policy.

Verification State Representation: When timing constraints must be verified, timed automata are commonly used for modeling systems [28]. Two limitations inherent to timed automata prevent us from using this typical approach for verification of scheduling policies produced by the approach presented in Section 3.

The first limitation is the state representation used by timed automata. Timed automata use continuous clock variables to abstract the passage of time. Verification states are represented as a set of constraints of the form: $c - d < x$, where c and d are clock variables and x is an integer value. Given an arbitrary policy generated by our adaptive approach, there is no guarantee that a particular

verification state can be represented using only constraints of this form. In particular, the dividing line between decision regions will most likely parallel the utilization vector. Only in the special case of two threads given equal utilization targets will it be possible to represent the decision boundary as a constraint of this form.

The second limitation precluding the use of of timed automata is the way in which verification states are propagated. Each verification state only captures the relative offset of the individual system clocks, abstracting away total elapsed system time. Timing properties of the system are encoded as guards that govern what conditions must be satisfied for state transitions to occur. These guards are expressed as inequalities between a clock and an integer. Because these inequalities are specified in the model, there is a threshold over which differences between clocks need not be tracked, therefore ensuring the number of possible verification states is finite. This property guarantees state propagation will terminate.

However, in a system that must track utilization, as in the method presented in Section 3, there is no such guarantee. With variables representing a thread’s utilization there is no bound on the size of the representation needed to track the changes made by later actions. Total system time and the time spent running any single thread both can grow without bound, and with them the number of bits needed to track utilization by a thread *accurately*.

Therefore, new methods to represent verification state and to perform verification state propagation are needed. We first consider how our scheduling policies will be implemented. One reasonable implementation is to deal with only a manageable finite history when evaluating the next action as the scheduler has finite memory. Based on this observation, the verification state becomes simply a history that encodes the last n actions.

Verification State Propagation: We then must deal with the problem of how to propagate verification states. Given the minimum and maximum execution times for each thread dispatch and a history of the last n actions, we can determine what subsets of utilization states are reachable. A simple decision procedure is then available for verification state propagation: iterate over the current set of utilization states and add the action given by the policy to the set of possible actions. As the results of our evaluation presented in Section 5 demonstrate, significant optimizations to this simple decision procedure may be available for certain scheduling policies.

This representation guarantees coverage of all possible verification states, and also guarantees that verification state space propagation terminates. This follows because the verification state space is finite (there are $\sum_{i=0}^n t^i$ possible histories where t is the number of threads and n is the maximum history length).

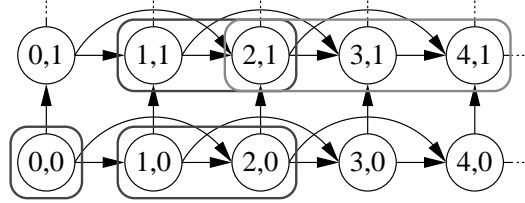


Fig. 2. Example state space exploration.

Figure 2 shows the beginning of state space exploration for a simple two thread system. In this example thread one has a deterministic run time of 1 time unit, while thread two runs for either 1 or 2 time units. The variability of thread two’s utilization induces (1) multiple possible utilization states in each verification state (except for at the origin), and (2) many utilization states that belong to multiple verification states. The boxes in Figure 2 show how the verification states are overlaid on the set of utilization states from the scheduling policy.

At first the verification state (corresponding to a null history) only includes the origin. However after the first decision, which is to dispatch thread one, the verification state now includes two utilization states, labeled (1,0) and (2,0). After two more decisions the verification state includes three of the underlying utilization states. State exploration continues until no transitions to unexplored verification states can be found.

However, this method is also pessimistic, allowing series of transitions that in practice are not possible. This means that systems positively verified are truly safe, but systems where error states are reachable relative to a given query, are not necessarily unsafe. The pessimism arises because each decision induces constraints on what the possible values of the utilizations are, over the n actions for which the decision was evaluated. As such we can create an increasingly optimistic model by continuously adding another action to the history. However, this leads to more complicated decision procedures. In order to show the applicability of this technique we will use the pessimistic method for full state space enumeration. This gives a good estimate to the relative costs of the search and can provide safety guarantees because of coverage.

5 Evaluation

We demonstrate these techniques on a small example problem with two threads and termination time $\tau = 512$. The resulting MDP has 131,882 states.

The thread run-time distributions are shown in Figure 3. These were generated by sampling Erlang distributions at the integers in the range from 1 to 16

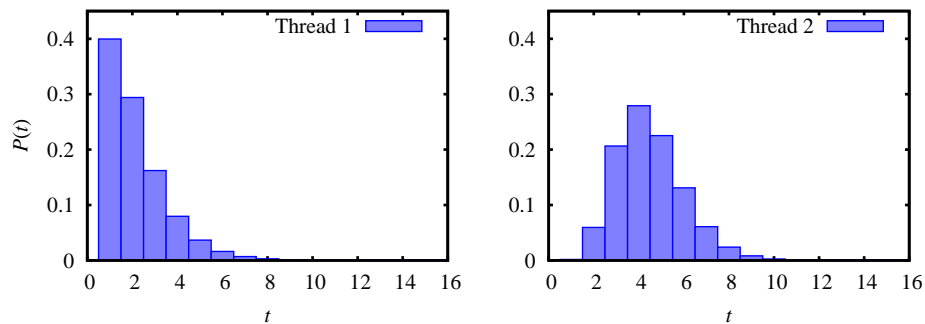


Fig. 3. Example problem run-time distributions.

inclusive, then normalizing the results to obtain discrete run-time distributions. The Erlang distribution for thread one has rate $\lambda = 1$ and shape $k = 2$ (mean 2), while the distribution for thread two has $\lambda = 2$ and shape $k = 18$ (mean 9). These distributions illustrate differences we expect to see in real systems, where user threads may be CPU-bound for long but highly variable periods of times while low-level event handlers occupy relatively short, fairly predictable intervals. The scheduling policy must therefore balance the need to maintain temporal predictability (and therefore its bias is toward thread one's smaller mean), with the enforcement of the desired utilization.

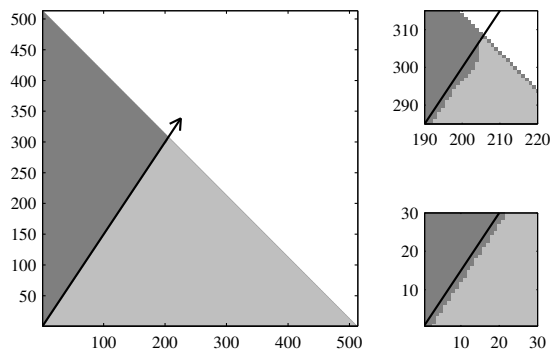


Fig. 4. Left: Optimal policy for this example problem. The ray shown in black shows the target utilization. The policy dispatches thread one (move right) in dark gray states and thread two (move upwards) in light gray states. **Bottom Right:** Close-up at the origin. The decision boundary runs roughly parallel to the target utilization ray. **Top Right:** Close-up at the terminal states near target utilization. Notice the protrusion of dark gray where the policy deviates from parallel.

The optimal policy for the example problem is shown in Figure 4. The policy recommends dispatching thread one in the dark gray regions, which advances the state along the horizontal axis. Thread two is dispatched in light gray states, advancing the state vertically. The target utilization is shown by the black ray. The decision boundary, best seen in the inset figures on the right in Figure 4, is parallel to target utilization, but translated to the right. In this interval it is better to execute thread one even though it is guaranteed to move the system away from target utilization. The alternative is to execute thread two, likely overshooting the target utilization and likely resulting in a net higher cost state because of the longer expected run-time. There is also an edge effect near the decision boundary due to the termination time. In the neighborhood of $\mathbf{x} = (210, 310)$, both actions lead to states that are quite close together because of the proximity of absorbing states. This leads to a short interval where the decision boundary lines up with the target utilization ray. Immediately prior to this interval is a protrusion where it is better to dispatch thread one. Doing so is likely to put the system into a state where dispatching thread two transitions the system directly to a good absorbing state. Due to the relatively small variance of thread one, the system can accurately aim for a good absorbing state a couple of decisions in advance.

Verification of properties in the state space induced by this scheduling policy can be significantly optimized over the simple decision procedure explained in section 4. Because of the decision boundary is mostly linear, most verification states need only determine the action suggested at the utilization states obtained by simply alternately maximizing and minimizing the share of the utilization received by each thread in the history.

Exploration of the resulting state space for offline verification is summarized in Table 1. For comparison, two different state space exploration techniques were used. First, the unconstrained state space was explored. In unconstrained exploration (summarized in the rightmost columns) any thread can be executed from any state. This results in a finite state machine with transitions connecting every possible decision history.

The second state space exploration was informed by the scheduling policy described above. Only a subset of the states reachable in the unconstrained case is reachable in this case, since the policy may be homogeneous over the set of utilization states underlying a particular validation state. Results for the exploration of this state space are summarized in the leftmost columns. As expected, the state space exploration guided by our scheduling policy is smaller and thus faster to compute than the full state space.

History Size	Under Scheduling Policy			Unconstrained		
	States	Transitions	Time	States	Transitions	Time
1	11	12	00:00:00	13	16	00:00:00
2	22	25	00:00:00	29	39	00:00:00
3	45	53	00:00:00	61	76	00:00:00
4	89	108	00:00:00	125	156	00:00:00
5	177	218	00:00:00	253	316	00:00:00
6	353	438	00:00:00	509	636	00:00:00
7	705	878	00:00:00	1021	1276	00:00:00
8	1409	1758	00:00:00	2045	2556	00:00:00
9	2817	3518	00:00:00	4093	5116	00:00:00
10	5633	7038	00:00:00	8189	10236	00:00:00
11	11265	14078	00:00:00	16381	20476	00:00:01
12	22529	28158	00:00:02	32765	40956	00:00:03
13	45057	56318	00:00:09	65533	81916	00:00:13
14	90096	112607	00:00:38	131069	163836	00:00:52
15	180205	225241	00:04:28	262141	327676	00:05:31
16	360420	450509	00:10:20	524285	655356	00:15:04
17	720851	901047	00:38:07	1048573	1310716	00:53:41
18	1441714	1802125	02:36:04	2097149	2621436	03:29:34
19	2883441	3604283	10:47:04	—	—	—

Table 1. Summary of Verification State Space Enumeration with Different History Sizes

6 Conclusions and Future Work

We have described an approach to system verification given a rational scheduler that maximizes a weighted fairness criterion given complete knowledge of distributions of thread execution times. With this knowledge about the system, we are able to derive an optimal policy for each utilization state up to some maximum system termination time. This is a step toward designing verified autonomous systems with specialized scheduling policies.

In practice, the scheduling policies derived from our system model have produced decision surfaces that partition the utilization space into linearly separable segments (up to edge effects). We have empirical evidence suggesting that this is a persistent effect; we are currently attempting to determine formally whether or not this is always the case. If policies are linearly separable, even only in special cases, then in those cases we can apply the simpler decision procedure described in Section 5.

The MDP analysis of the scheduler is a critical component to discovering its behavior at a quantum-by-quantum level. However, this requires explicitly tabulating the possible utilization states of the system, which scales poorly with the number of threads. More importantly, the utilization state space is unbounded, which necessitates the introduction of absorbing states in the model that in-

adequately express the concerns of the original system that we are modeling. Eliminating termination time from the model is an essential next step towards broadening the applicability of this modeling approach.

It seems possible to eliminate the termination time from the MDP model by taking advantage of the self-similarity of the system dynamics from each utilization state. We can define equivalence classes over utilization states based on the displacement from zero cost states. By establishing a transition function over these equivalence classes we can capture the dynamics of the original system model without relying on some fixed termination time. As time increases the number of equivalence classes also increases, since it is possible to get farther and farther from target utilization. This can be handled by introducing absorbing states. Unlike the absorbing states described in this work, these states would likely be homogeneous with respect to the optimal policy.

One of the key limitations of the verification state spaces presented in this paper is their pessimism. In order to improve upon this, more complicated decision procedures are needed at each step of state propagation in model checking. While general and powerful decision procedures such as those proposed in [29] seem applicable, the resulting increase in the complexity of state propagation may make them intractable in practice.

References

1. Irvine, C.E.: The reference monitor concept as a unifying principle in computer security education. (citeseer.ist.psu.edu/299300.html)
2. ARINC Incorporated Annapolis, Maryland, USA: Document No. 653: Avionics Application Software Standard Interface (Draft 15). (1997)
3. W. Mark Vanfleet and Jahn A. Luke and R. William Beckwith and Carol Taylor and Ben Calloni and Gordon Uchenick: MILS: Architecture for High-Assurance Embedded Computing. http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html (Crosstalk: the Journal of Defense Software Engineering, August, 2005)
4. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A secure environment for untrusted helper applications. In: Proceedings of the 6th Usenix Security Symposium, San Jose, CA, USA (1996)
5. Garfinkel, T.: Traps and pitfalls: Practical problems in in system call interposition based security tools. In: Proc. Network and Distributed Systems Security Symposium. (2003)
6. Garfinkel, T., Pfaff, B., Rosenblum, M.: Ostia: A delegating architecture for secure system call interposition. In: Proc. Network and Distributed Systems Security Symposium. (2004)
7. Provos, N.: Improving host security with system call policies. In: 12th USENIX Security Symposium, Washington, DC (2003)
8. Liu, J.W.S.: Real-time Systems. Prentice Hall, New Jersey (2000)
9. Goyal, Guo, Vin: A Hierarchical CPU Scheduler for Multimedia Operating Systems. In: 2nd Symposium on Operating Systems Design and Implementation, USENIX (1996)
10. Regehr, Stankovic: HLS: A Framework for Composing Soft Real-time Schedulers. In: 22nd IEEE Real-time Systems Symposium, London, UK (2001)

11. Regehr, Reid, Webb, Parker, Lepreau: Evolving Real-time Systems Using Hierarchical Scheduling and Concurrency Analysis. In: 24th IEEE Real-time Systems Symposium, Cancun, Mexico (2003)
12. Aswathanarayana, T., Subramonian, V., Niehaus, D., Gill, C.: Design and performance of configurable endsystem scheduling mechanisms. In: Proceedings of 11th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS). (2005)
13. Migliaccio, A., Tidwell, T., Gill, C., Aswathanarayana, T., Niehaus, D.: Group scheduling in selinux to mitigate cpu-focused denial of service attacks. Technical Report WUCSE-2005-55, Department of Computer Science and Engineering, Washington University in St.Louis (2005)
14. Tidwell, T., Watkins, N., Subramonian, V., Niehaus, D., Gill, C., Migliaccio, A.: The design, modeling, and implementation of group scheduling for isolation of computations from adversarial interference. Technical Report WUCSE-2006-34, Computer Science and Engineering Department, Washington University in St.Louis (2006)
15. Tidwell, T., Gill, C., Subramonian, V.: Scheduling induced bounds and the verification of preemptive real-time systems. Technical Report WUCSE-2007-34, Computer Science and Engineering Department, Washington University in St.Louis (2007)
16. Martin, W., White, P., Taylor, F.S., Goldberg, A.: Formal construction of the mathematically analyzed separation kernel. In: ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering, Washington, DC, USA, IEEE Computer Society (2000) 133
17. Jones, M.: What really happened on Mars? www.research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html (1997)
18. Zhang, Y., Lu, C., Gill, C., Lardieri, P., Thaker, G.: Middleware support for aperiodic tasks in distributed real-time systems. In: RTAS '07: Proceedings of the 13th IEEE Real Time on Embedded Technology and Applications Symposium, Washington, DC, USA, IEEE Computer Society (2007) 497–506
19. Kesten, Y., Pnueli, A., Sifakis, J., Yovine, S.: Decidable integration graphs. *Information and Computation* **150**(2) (1999) 209–243
20. Huang, H.M., Gill, C.: Modeling timed component-based real-time systems. Technical Report WUCSE-2008-1, Computer Science and Engineering Department, Washington University in St.Louis (2008)
21. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics* **10**(1) (1962) 196–210
22. Sutton, R.S., Barto, A.G.: *Reinforcement Learning: An Introduction*. MIT Press (1998)
23. Tesauro, G.: Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing* **11**(1) (2007) 22–30
24. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: On the use of hybrid reinforcement learning for autonomic resource allocation. *Cluster Computing* **10**(3) (2007)
25. Littman, M.L., Ravi, N., Fenson, E., Howard, R.: Reinforcement learning for autonomic network repair. In: Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004). (2004) 284–285
26. Whiteson, S., Stone, P.: Adaptive job routing and scheduling. *Engineering Applications of Artificial Intelligence* **17**(7) (2004) 855–69
27. Puterman, M.L.: *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Interscience (1994)
28. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical Computer Science* **126**(2) (1994) 183–235
29. Andrei, S., Cheng, A.M.: Verifying Linear Real-Time Logic Specifications. In: 28th IEEE International Real-Time Systems Symposium, Tuscon, AZ (2007)