Scheduling policy design for autonomic systems

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

Department of Computer Science and Engineering,
Washington University,
St. Louis, MO 63130, USA
E-mail: rlg1@cse.wustl.edu
E-mail: ttidwell@cse.wustl.edu
E-mail: cdgill@cse.wustl.edu
E-mail: wds@cse.wustl.edu
*Corresponding author

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 and efficient foundation for verifying and enforcing concurrency and timing guarantees is still needed for these approaches. This paper summarises our previous work on addressing these challenges, on Markov decision process-based scheduling policy design and on wrapping repeated structure of the scheduling state spaces involved into a more efficient model, and presents a new algorithm called expanding state policy iteration (ESPI), that allows us to compute the optimal policy for a wrapped state model.

Keywords: autonomic systems; policy iteration; scheduling; state space reduction.

Reference to this paper should be made as follows: Glaubius, R., Tidwell, T., Gill, C.D. and Smart, W.D. (2009) 'Scheduling policy design for autonomic systems', *Int. J. Autonomous and Adaptive Communications Systems*, Vol. 2, No. 3, pp.276–296.

Biographical notes: Robert Glaubius received his BS in Computer Science from the University of Nebraska – Lincoln in 2001. Currently, he is a PhD Candidate in Computer Science and Engineering in the Department of Computer Science and Engineering, Washington University in St. Louis. His research interests center around reinforcement learning: particularly, methods for automatic state abstraction and reformulation in large or continuous state spaces.

Terry Tidwell received his BS in Computer Science from the University of Tulsa and is currently pursuing a PhD in Computer Science at Washington University in St Louis. His research interests include the development and validation of cyber-physical systems and system verification via model checking.

Christopher D. Gill is an Associate Professor of Computer Science and Engineering at Washington University in St. Louis. His research interests include formal modeling, verification, implementation, and empirical evaluation of policies and mechanisms for enforcing timing, concurrency, fault-tolerance, and security properties in distributed, mobile, embedded and real-time systems. He has more than 50 refereed and invited technical publications and has an extensive record of service in review panels, standards bodies, workshops, and conferences for distributed real-time and embedded computing. He is a Member of the ACM, the IEEE and the IEEE Computer Society.

William D. Smart is an Assistant Professor of Computer Science and Engineering at Washington University in St. Louis. His research interests span the fields of mobile robotics, machine learning and brain–machine interfaces. He is the Co-chair of the IEEE Robotics and Automation Society Committee on Competitions and Challenges, and is the Founding Chair of the ICRA Robot Challenge.

1 Introduction

An autonomic computing system must respond adaptively to varying operating conditions, automatically and without external intervention. The adaptive behaviours 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 behaviours of the system.

The design of such autonomic systems, therefore, must strive to meet these timing constraints to the extent possible, which in turn requires adaptive but precise scheduling of system resources to ensure their allocation is feasible for the system's constraints. How to ensure reliable execution of autonomic computing systems is thus an important and challenging research problem.

Existing approaches to ensuring the verifiably feasible use of system resources online often employ some kind of reference monitor (Irvine, 1999), 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 and network security, separation kernels (ARINC Incorporated Annapolis, 1997; Vanfleet et al., 2005) for partitioning resource use, and user level sandboxes (Garfinkel, 2003; Garfinkel et al., 2004; Goldberg et al., 1996; Provos, 2003) that intercept system calls made by application programs, illustrate the applicability of resource monitors to managing the execution of system activities.

1.1 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, these approaches have important limitations. For sandbox approaches, the crucial limitation is in how precisely the desired execution semantics can be enforced, while for separation kernel approaches 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 (Liu, 2000) offer what amounts to a kind of (admittedly by-passable) 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 (Aswathanarayana et al., 2005; Goyal, 1996; Regehr and Stankovic, 2001; Regehr et al., 2003) 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 (Migliaccio et al., 2005; Tidwell et al., 2006). 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 are not currently available.

1.2 Solution approach

To overcome the limitations of existing approaches for ensuring the feasible use of system resources, we have developed new techniques (Glaubius et al., 2008; Tidwell et al., 2008) that are flexible in the policies they can enforce, and within which particular resource monitors can be customised according to their intended use. In this paper, we review those contributions and present a new algorithm for computing optimal scheduling policies.

In Section 3, we describe our method for scheduling policy design (introduced in Tidwell et al. (2008)) that can be tailored to specified workloads, which is based on formalising the problem as a Markov decision process (MDP). 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 further consideration of other relevant techniques to future work.

Section 4 describes our technique for wrapping similar regions of the scheduling state space that allows a scheduling policy to be encoded as an MDP with a greatly reduced number of states (Glaubius et al., 2008). It also eliminates the need for setting an artificial time horizon and thus avoids the edge effects associated with our original technique (Tidwell et al., 2008). Because of the compact representation of the MDP, it is easier to create policies derived from more densely sampled states, allowing better approximations of policies with continuous time semantics.

In Section 5, we present the main contribution of this paper, expanding state policy iteration (ESPI), a novel approach that allows us to compute the optimal policy for a wrapped state model, which builds on and amplifies the previous contributions of our research. In Section 6, we discuss how ESPI also elicits the necessary states for verification of the resulting policy. Finally, in Section 7, we summarise the contributions of our research, and describe planned future work.

2 Related work

2.1 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 such as safe and efficient multi-threading (Garfinkel, 2003). Hybrid interposition architectures (Garfinkel et al., 2004), therefore, move part of the sandbox into the kernel, but still rely on the kernel's native scheduling policies and mechanisms, which do not offer sufficient control over system components such as interrupts (Tidwell et al., 2006), 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 (ARINC Incorporated Annapolis, 1997; Vanfleet et al., 2005). For example, the MILS kernel (Vanfleet et al., 2005) 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 (Martin et al., 2000) 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 (Jones, 1997), illustrate that identifying all dependences up front in real-world systems is a daunting task.

2.2 Scheduling techniques

Many thread scheduling policies have been designed and analysed to ensure guaranteed feasibility of resource use in closed real-time systems (Liu, 2000). Most of those approaches assume that the number of tasks accessing system resources, and their invocation rates and execution times, are all well characterised. Real-time systems approaches that allow even basic extensions such as asynchronous task arrival must depend on special services, e.g. admission control (Zhang et al., 2007), to maintain resource feasibility at run-time.

Hierarchical scheduling techniques (Aswathanarayana et al., 2005; Goyal, 1996; Regehr and Stankovic, 2001; Regehr et al., 2003) 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 preemption and require reasoning about continuous time may encounter problems with decidability (Kesten et al., 1999), so that special techniques that exploit knowledge about the structure of the specific scheduling problem (Tidwell et al., 2007) 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 (Held and Karp, 1962). A related technique, reinforcement learning (RL) (Sutton and Barto, 1998) (often called approximate dynamic programming), has been identified as a learning technology that holds great promise for the autonomic computing community (Tesauro, 2007). It has been applied successfully to several domains, including computer cluster management (Tesauro et al., 2007), network configuration repair (Littman et al., 2004) and job scheduling (Whiteson and Stone, 2004). 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 goal of our research is to develop new scheduling techniques that are flexible in the policies they can enforce, particularly in the context of systems with dynamic execution requirements. In this section, we present the foundations of our approach (Tidwell et al., 2008) upon which the new research contributions presented in this paper (Sections 5 and 6) are built.

3.1 System model

In previous work (Tidwell et al., 2008), we proposed an abstract system model in which:

1 multiple threads of execution require mutually exclusive use of a single common resource (i.e. a CPU)

2 whenever, a thread is granted the resource, it occupies the resource for a finite and bounded subsequent duration

3 the duration for which a thread occupies the resource may vary from run-to-run but overall obeys a known, independent and bounded distribution

4 a scheduler repeatedly chooses which thread to run according to a given scheduling policy, dispatches that thread and waits until the end of the duration during which the thread occupies the resource.

This system model establishes a foundation for the kinds of scheduling enforcement problems that can arise in open soft real-time systems built atop commonly used operating systems such as Linux or VxWorks. For example, within the Linux kernel, hard and soft interrupts may be threaded and placed under scheduler control (Tidwell et al., 2007), with different resulting durations of resource occupation for the different kinds of interrupts. As future work, we plan to extend our approach to address issues such as preemption among *inter-dependent* intervals of execution, which this system model does not support. We represent the state of the system in terms of the resource utilisation of each thread of execution.

3.2 Scheduling decision model

As in Tidwell et al. (2008), our scheduling decision model consists of sequentially deciding to dispatch one of n threads, whenever the CPU becomes available.

The scheduler's objective is to maintain the relative resource utilisation for each thread near some target utilisation, encoded in a vector u. Threads release the CPU after some time; the execution time of thread *a* is non-deterministic with known distribution Pa. We make the simplifying assumption that run-times for subsequent executions of the same thread are independent, so that historical thread executions do not colour future thread executions.

We represent this scheduling decision model as a MDP. The literature on solving MDPs is quite mature; in this section, we will state a number results that will be useful in this work and refer the interested reader to that literature for a full coverage (Bertsekas and Tsitsiklis, 1996; Puterman, 1994; Rust, 1996).

An MDP is a four-tuple (*X, A, R, P*), where *X* is a set of process states and *A* is a set of available actions. The transition function $P(y|x, a)$ describes the probability of entering state *y* after taking action *a* from state *x*. The real-valued expected reward function $R(x, a)$ indicates the immediate cost or benefit of executing action *a* in state *x*.

A policy π recommends actions in each state of an MDP. Our objective is to discover a policy that maximises the value obtained within the MDP. We focus on the discounted reward setting; given a discount factor $\gamma \in [0, 1)$, the value of a policy π at a state *x* is the discounted cumulative reward obtained by following policy π, defined as:

$$V^{\pi}(x) = E\left\{\sum_{t=0}^{\infty} \gamma^t R\left(x_t, \pi(x_t)\right) \middle| x_0 = x\right\} \tag{1}$$

where x_{t+1} is distributed according to $P(\cdot \mid x_t, \pi\ (x_t))$. This places greater emphasis on rewards that can be reached in fewer steps. In the model described in this paper, the undiscounted case (i.e. $\gamma = 1$), results in $V^{\pi} = -\infty$ for any policy π, thus some discounting strategy is necessary in order to distinguish between policies. The optimal policy, π^*, maximises Equation (1). Its value function V^{π^*}, or more compactly V^*, satisfies the system of Bellman equations

$$V^*(x) = \max_{a \in A}\left\{R(x,a) + \gamma \sum_{y \in X} P(y \mid x, a) V^*(x)\right\}, \tag{2}$$

We write this compactly using the Bellman backup operators Γ_a and Γ: for any state *x* and real-valued function *V* over *X*,

$$\left(\Gamma_a V\right)(x) = R(x,a) + \gamma \sum_{y \in X} P(y \mid x, a) V(y) \tag{3}$$

$$(\Gamma V)(x) = \max_{a \in A}\{(\Gamma_a V)\}(x) \tag{4}$$

In this notation, we can write Equation (2) as

$$V^* = \Gamma V^* \tag{5}$$

When *X* is finite, we can compute the optimal value function V^* using one of a number of classical approaches, such as *value iteration* or *policy iteration*. We will discuss value iteration here, as it provides concepts that we will use to establish results for the thread scheduling problem. We will discuss the policy iteration algorithm in Section 5, where

we propose an extension to the thread scheduling problem. Both of these algorithms were developed for problems in which X and A are finite.

Value iteration computes the value of behaving optimally for t steps, with t increasing. By convention, we define $V_0 = 0$ and compute V_t recursively according to

$$V_t = \Gamma V_{t-1}$$

The algorithm converges asymptotically on V^* in the supremum norm when the magnitude of rewards are bounded. In practice, we use V_t for some suitably large t as an approximation to V^*. Given a value iterate V_t we can define a greedy policy π_t as

$$\pi_t(x) = \arg\max_{a \in A} \left\{ \left(\Gamma_a V_t \right)(x) \right\} \tag{6}$$

Since V^* satisfies the fixed point condition $V^* = \Gamma V^*$, the greedy policy with respect to V^* is the optimal policy. The closer V_t is to V^*, the closer the corresponding policy is to the optimal policy (Singh and Yee, 1994). When the set of policies is finite, the greedy policies corresponding to the value iterates generated by value iteration converge in finite time.

3.3 Utilisation state model

In our previous work (Tidwell et al., 2008), the scheduling decision state space consists solely of utilisation states $x \in \mathbb{Z}_+^n$, where $\mathbb{Z}_+$ is the non-negative integers. Component x_a of x gives the cumulative historical CPU utilisation of thread a, so that the sum of components $\sum_{a=1}^{n} x_a$ is the total CPU usage over the entire system history. Since CPU usage is strictly non-negative, we can write this more compactly as $\| x \|_1$, where $\| \bullet \|_1$ is the Manhattan distance.

Scheduling actions consist of the decision to run thread $a \in A = \{1, 2, \ldots, n\}$. We assume that the discrete run-time distributions for each thread a, P_a, are known and are defined over $\{1,2,\ldots, \beta\}$ We use these to define the state transition probabilities

$$P(y \mid x, a) = \begin{cases} P_a(\theta) & y = x + \theta \Delta_a \\ 0 & \text{otherwise,} \end{cases} \tag{7}$$

where $\Delta_a = (\delta_{a1}, \delta_{a2}, \ldots, \delta_{an})$ and δ_{ab} is the Kronecker delta. This means that the system can only move between states that differ only in terms of the utilisation of thread a; the probability of this transition equals the probability $P_a(\theta)$ that thread a runs for $\theta = y_a - x_a$ time quanta.

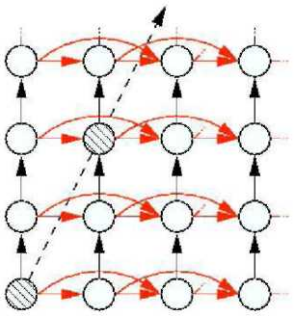
We formulate rewards in this model based on the deviation from target utilisation $u \in \mathbb{R}_+^n$. We aim to provide thread a with a proportion u_a of the CPU time, with the requirement that the per-thread targets u_a satisfy $0 < u_a < 1$ and sum to one. The reward function $R(x, a)$ depends on a penalty function r,

$$R(x,a) = \sum_{\theta=1}^{\beta} P_a(\theta) r\left(x + t\Delta_a\right) \tag{8}$$

$$r(x) = -\| x - \| x \|_1 u \|_1 \tag{9}$$

the penalty function is the distance between x and the target utilisation at time $\|x\|_1, \|x\|_1 u$. Figure 1 shows this model for an example with two threads.

Figure 1 Utilisation state space graph for a system with two threads (see online version for colours)



Note: Each vertex is a utilisation state; edges represent transitions with non-zero probability. Thread one (arrows pointing to the right) is shown in red, thread two (arrows pointing upward) is shown in black. The target utilisation ray is shown by the dashed arrow. States marked with a hashed pattern achieve target utilisation.

In practice, we cannot work directly in the utilisation state space due to its infinite size. In our previous work, we addressed this by truncating the utilisation state space by adding a termination time; we only considered states with cumulative utilisation less than the termination time T. Since the resulting state space is finite, we can apply standard techniques directly. However, the policies obtained from this model suffered from artefacts near the termination time, and so provided poor approximations to optimal policies over the full space.

A further problem with this approach is that the state space is large in both the number of threads n and the termination time. The number of states in this truncated model is $\sum_{i=0}^{T} \binom{n+i-1}{n-1}$, which grows exponentially in T and n. In Section 3, we describe a new approach that takes advantage of the similarity of the transition function to preserve optimality while eliminating the dependence on an artificially introduced termination time, by 'wrapping' the state space in order to line up states with similar futures.

4 Wrapped state model

To ensure a tractable representation of the utilisation state space described in Section 3, our original work (Tidwell et al., 2008) introduced a fixed time horizon into the model, which enforces a bound on the number of possible utilisation states. This assumption is quite restrictive, since we often do not know in advance the system's life span, or the intervals of system execution over which scheduling decisions must be evaluated.

In Glaubius et al. (2008), we extended our previous work to eliminate the dependence on a time horizon by taking advantage of recurrence in the system model. As we discuss in this section, that allows us instead to use a discounting scheme that can be thought of as a prior probability that a system will continue to execute from time step to time step and to derive scheduling policies that are optimal to arbitrary planning horizons.

The intuition behind our approach to exploiting recurrence is to notice that the marked states in Figure 1 are highly similar: they have the same reward and the relative distribution of future states is the same. We would expect the optimal policy to prefer the same action in these states. The key idea of our approach is to detect similar states and collapse them together to obtain a smaller, more tractable model.

We say that a system is *periodic* with period k if and only if k is a positive integer satisfying ku is in $\mathbb{Z}_+^n$, which means that the utilisation ray (the set of real-valued points obtained by applying positive real scale factors to u) passes through utilisation states at regular intervals. For example, in Figure 1, $u = (1/3, 2/3)$, so ku is integer-valued for any $k > 0$ that is a multiple of three. We are particularly interested in the minimum period, which in this case is $k = 3$. Lemma 1 establishes the existence of periodic systems and provides a method for determining the minimum period for a broad class of utilisation-based systems.

Lemma 1: Suppose $u = (u_1, u_2, \ldots, u_n)$ such that for each a, $0 < u_a < 1$ is rational, and let p_a and q_a be the least positive integer terms satisfying $u_a = p_a/q_a$, that is, $GCD(p_a, q_a) = 1$. Then the system is periodic with minimum period $k = LCM(q_1, q_2, \ldots q_n)$.

Proof: Suppose that l is a period. Then for any $a \in A$, $lu_a = l(p_a/q_a) = z_a \in \mathbb{Z}_+$. Therefore, $lq_a = z_a/p_a$, so z_a/p_a is a positive integer. $l = q_a(z_a/p_a)$, so l is a common multiple of $q_1, q_2, \ldots, q_n$. It follows that $k = \text{LCM}(q_1, q_2, \ldots, q_n)$ is the minimum period of the system.

We use the period of the utilisation state model to wrap the state space. One can visualise our wrapping method in two dimensions as first drawing the utilisation states on a special sheet of paper with the state (0, 0) in the bottom left corner as in Figure 1 and extending to infinity away from that corner. The wrapping procedure consists of rolling up the sheet to form a tube so that the states $\lambda k\text{u}, \lambda \in \mathbb{Z}_+$, rest atop one another. We treat utilisation states that rest atop one another as equivalent in this wrapped model.

The key idea behind the wrapped state model is that the distance from the initial utilisation (0, 0, …, 0) is not all that important when choosing how to act. Historical utilisation states matter only to the extent they determine the distribution over future utilisation states, and that distribution only matters to the extent that it predicts the deviation from the utilisation ray. By wrapping up the state space, we are enumerating unique possible deviations from the target utilisation while pruning out states that encode redundant deviation information. This results in a significant reduction in the number of MDP states; further, we can do this without loss of optimality in the scheduling policy for the original utilisation state model.

One interesting note is that we do not need to consider the thread run-time distributions when determining the state wrapping; it depends only on the period and target utilisation vector. This is true under our assumption that the thread run-time distributions are independent of the utilisation state. This allows us to collapse equivalent wrapped states without losing information about the distribution of future states.

We will show that the value function is periodic with period k, in the sense that the optimal value function $V^*(x) = V^*(x + \lambda k\, u)$ for any positive integer λ. This relies on the fact that the penalty function is periodic:

$$\begin{aligned} r(x+\lambda ku) &= -\| x+\lambda ku - \| x+\lambda ku \|_1\, u \|_1 \\ &= -\| x+\lambda ku - \| x \|_1\, u - \lambda ku \|_1 \\ &= -\| x - \| x \|_1\, u \|_1 \\ &= r(x) \end{aligned}$$

This holds because u, x, k and λ are non-negative and $\|u\|_1 = 1$. As a consequence, the expected reward function is also periodic:

$$\begin{aligned} R(x+\lambda ku, a) &= \sum_{\theta=1}^{\beta} P_a(\theta) r\left(x+\theta\Delta_a + \lambda ku\right) \\ &= \sum_{\theta=1}^{\beta} P_a(\theta) r\left(x+\theta\Delta_a\right) \\ &= R(x,a) \end{aligned}$$

With these results, we can show that the optimal value function is also periodic.

Theorem 1: For all states x and non-negative integers λ,

$$V^*(x) = V^*(x+\lambda ku) \tag{10}$$

Proof: We will inductively show that, for all x, $V^*(x) = V^*(x + \lambda ku)$ by showing that $V_t(x) = V_t(x + \lambda ku)$. Initially, $V_0 = 0$, so it is trivially true that $V_0(x) = V_0(x + \lambda ku) = 0$. Suppose $V_t(x) = V_t(x + \lambda ku)$. Then for any $a \in A$,

$$\begin{aligned} \left(\Gamma_a V_t\right)(x) &= R(x,a) + \gamma \sum_{\theta=1}^{\beta} P_a(\theta) V\left(x+\theta\Delta_a\right) \\ &= R(x+\lambda ku, a) + \gamma \sum_{\theta=1}^{\beta} P_a(\theta) V_t\left(x+t\Delta_a + \lambda ku\right) \\ &= \left(\Gamma_a V_t\right)(x+\lambda ku) \end{aligned}$$

Therefore,

$$\begin{aligned} V_{t+1}(x) &= \max_{a\in A}\left\{\left(\Gamma_a V_t\right)(x)\right\} \\ &= \max_{a\in A}\left\{\left(\Gamma_a V_t\right)(x+\lambda ku)\right\} \\ &= V_{t+1}(x+\lambda ku) \end{aligned}$$

Since, the per-action Bellman operators Γ_a agree on periodic states x and $x + \lambda ku$, the greedy policy agrees at these states as well. Since the optimal policy is greedy with respect to the optimal value function, and the optimal value function is periodic, the optimal policy is periodic.

The implication of this result is that if a set of states can be parameterised in terms of some canonical state x and displacements λku, then we only need to estimate and store a

value for one member of that set. These states are equivalent with respect to the value function, so it is sufficient to construct a policy that makes decisions based solely on which equivalence class of states the current utilisation state belongs to. We compute the value for the 'smallest' member of equivalence classes of states $\{x + \lambda ku : \lambda \in \mathbb{Z}_+\}$ given x in $\mathbb{Z}_+^n$, which we can think of as wrapping up the state space in the direction u. To describe this wrapping procedure formally, we define the wrapping function,

$$w(x) = x - \lambda_x ku, \quad \lambda_x = \max\{\lambda \in \mathbb{Z}_+ : \forall a \in A, x_a - ku_a \geq 0\} \tag{11}$$

$w(x)$ denotes a vector modulo operation with modulus ku. Using Theorem 1, we know that every state x with $w(x) = y$ has the same value, so we only need to compute the value at these states. In Figure 1, the marked states are equivalent to each other and to all other utilisation states along the utilisation ray.

The transition function between wrapped states is defined by analogy to Equation (7):

$$P(w(y) \mid w(x), a) = \begin{cases} P_a(\theta) & w(y) = w(x + \theta\Delta_a) \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

One concern could be that θ may not be unique: that $w(y)$ equals both $w(x + \theta_1\Delta_a)$ and $w(x + \theta_2\Delta_a)$, with distinct values θ_1 and θ_2. Suppose that this is the case. Then

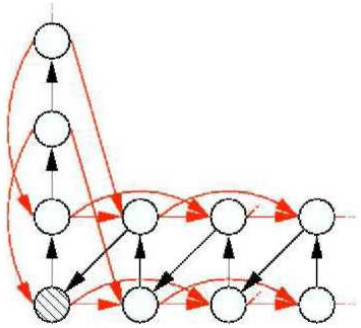
$$\begin{aligned} & w(x + \theta_1\Delta_a) = w(x + \theta_2\Delta_a) \\ & \equiv x + \theta_1\Delta_a - \lambda_1 ku = x + \theta_2\Delta_a - \lambda_2 ku \\ & \equiv (\theta_1 - \theta_2)\Delta_a = (\lambda_1 - \lambda_2) ku \end{aligned}$$

If $\lambda_1 = \lambda_2$, then $\theta_1 = \theta_2$. If $\lambda_1 - \lambda_2 \neq 0$, $u = \Delta_a(\theta_1 - \theta_2)/[k(\lambda_1 - \lambda_2)]$ a degenerate target utilisation in which only thread a should be utilised. We are not concerned with such cases because the optimal scheduling policy is then trivial. Therefore, the transition model in Equation (12) is well-defined. Figure 2 illustrates this wrapping method on the example from Figure 1. This process basically takes the original collection of utilisation states and deletes all of the utilisation states that are componentwise no less than ku, then reattaches transitions to deleted states to their equivalents among the wrapped states.

The wrapped state MDP consists of the set of states $X = \{w(x) : x \in \mathbb{Z}_+^n\}$, the same set of actions A as in the utilisation state model, and reward function R and transition function P restricted to the wrapped states as described above. A useful equivalent definition for the state space is that $X = \{x \in \mathbb{Z}_+^n : x = w(x)\}$. Since for any state $x = w(x) + \lambda ku$, for some $\lambda \in \mathbb{Z}_+$, Theorem 1 applies, $V^*(w(x)) = V^*(x)$. It follows that $\pi^*(w(x))$ on wrapped state $w(x)$ is the optimal action at each equivalent state in the utilisation state model.

The wrapped state model does not immediately cure all the problems of the original unbounded utilisation state model. The set of states is still infinite, so we are no more able to compute the value function exactly over X than we were able to over all of $\mathbb{Z}_+^n$. However, because of the wrapping of the state space, there is only a finite number of good states – for example, there is only one state, 0, that corresponds to perfect utilisation. In the next section, we use this observation to derive an algorithm, ESPI, that is able to compute good policies and exactly evaluate their value by looking at only finitely many states.

Figure 2 The wrapped state model corresponding to the utilisation state model in Figure 1 (see online version for colours)



Note: Vertices represent equivalence classes and edges represent transitions for thread one (red) and thread two (black).

5 Policy iteration

To compute the optimal policy for a wrapped state model (described in Section 4), we have extended our previous work with a novel approach based on *policy iteration* (Puterman, 1994), which we present in this section. Policy iteration (PI) is a dynamic programming approach to solving an MDP. In addition to the backup operators Γ and Γ_a (Equations (3) and (4)), we define the policy backup operator Γ_π for policy π in terms of the per-action operator $\Gamma_{\pi(x)}$,

$$(\Gamma_\pi V)(x) = (\Gamma_{\pi(x)} V)(x),$$

which backs up the value of following $\pi(x)$ at each state x.

Whereas, value iteration begins with an initial value function and tunes it asymptotically towards the optimal value function, policy iteration begins with an initial policy π_0 that is tuned towards optimal. In finite state and action MDPs, policy iteration is guaranteed to converge in finite time. PI consists of repeating two steps, *policy evaluation* and *policy improvement,* until the policies generated cease to change.

Policy evaluation: given the current policy π_t, its value function $V^{\pi_t}(x)$ is the long-term expected reward for following π_t starting at the state x. It satisfies the fixed-point condition

$$V^{\pi t} = \Gamma_{\pi_t} V^{\pi t} \tag{13}$$

If X is finite, we can solve for V^{π_t} as a finite system of linear equations. Let R^{π_t} be the n-vector of rewards $R^{\pi_t}(x) = R(x, \pi_t(x))$, and $P^{\pi_t}(y \mid x)$ be the $n \times n$ matrix of transition probabilities $P^{\pi_t}(y \mid x) = P(y|x, \pi_t(x))$. Then, we can formulate the fixed point condition as

$$V^{\pi_t} = R^{\pi_t} + \gamma P^{\pi_t} V^{\pi_t} \tag{14}$$

which we can then solve for V^{π_t} using standard algebraic techniques.

Policy improvement: in the policy iteration step, the algorithm checks whether the current policy could yield greater value by switching its action at each state. This corresponds to evaluating the value of executing action a at the current state, then following π_t afterwards:

$$\pi_{t+1}(x) = \arg\max_{a \in A} \left\{ \left(\Gamma_a V^{\pi_t} \right)(x) \right\}$$

This is guaranteed to yield a policy that is closer to optimal than the previous policy π_t.

The policy iteration algorithm as described above is only feasible when state space is finite. The wrapped state model of the thread scheduling problem defines an MDP that is discrete, but with infinitely many states. We now describe our ESPI algorithm, which allows us to compute the optimal policy for a wrapped state model.

5.1 Expanding state policy iteration

In the wrapped state model, the cost of entering a state x is the 1-norm distance from the point on the target utilisation ray with the same cumulative utilisation, $\|x - \tau u\|_1$, where $\tau = \|x\|_1$. This implies that if we were to repeatedly execute the same thread, we would naturally see progressively worse utilisation states. This is to be expected, since this one thread would be increasingly over utilised while the rest would starve.

This indicates that any good policy will avoid over- or underutilising any thread beyond some tolerance. Some notation: state x lies in the *horizon* $\tau = \|x\|$. The *target share* for thread a in horizon τ is τu_a. If $x_a > \|x\| u_a$, then a is over utilised in x; if $x_a < \|x\| u_a$, it is underutilised.

ESPI takes advantage of this intuition about the behaviour of reasonable policies by explicitly representing only those states that are necessary in order to perform policy evaluation and improvement given a collection of initial states $X_I \subseteq X$; for instance, $X_I = \{0\}$ in the thread scheduling problem. In addition to the policy evaluation and improvement steps, ESPI also includes a state expansion step.

ESPI begins with the initial policy π_0 and initial states X_I. Then, at each iteration $t = 0, 1, \ldots$, the following steps occur. *State expansion*: this consists of three steps

1 construct a state set X_t from X_I by recursively adding states that are reachable from X_t under the policy π_t

2 build a second state set, Y_t, that includes X_t and all of the states that are reachable from X_t by applying any single action

3 Close Y_t by recursively adding states that are reachable from Y_t under π_t.

Policy evaluation: evaluate V^{π_t} over all of the states in Y_t. *Policy improvement*: perform policy improvement over the states in X_t. If the policy does not change, halt.

The ESPI algorithm is described in pseudocode in Algorithm 1. The policy evaluation and improvement steps are almost identical to their counterparts in standard PI. Our contribution, the state expansion step, is significantly more involved. We will discuss each of these steps in detail.

Algorithm 1 ESPI (X_I, π_0)

for $t = 0,1,2,\ldots$ do

$X_t := C_{\pi_t} X_I$

$Y_t := C_{\pi_t}(\bigcup_{a\in A} C_a^1 X_t)$

$V_t := \left(I - \gamma P_{Y_t}^{\pi_t}\right)^{-1} R_{Y_t}^{\pi_t}$

$$\pi_{t+1}(x) := \begin{cases} \arg\max_{a\in A}\{(\Gamma_{a,Y_t} V_t)(x)\} & x \in X_t \\ \pi_0(x) & x \notin X_t \end{cases}$$

if $\pi_{t+1} = \pi_t$, return

end for

5.2 State expansion

State expansion consists of constructing an *evaluation set* X_t and an *improvement envelope* Y_t. The evaluation set X_t is constructed from the initial states X_j by taking its closure under the current policy π_t, $X_t = C_{\pi_t} X_I$. We define this closure operator recursively for states Y and policy π as follows:

$$C_\pi^0 Y = Y$$
$$C_\pi^k Y = C_\pi^{k-1} Y \cup \{y \in X : \exists x \in C^{k-1} Y, P^\pi(y \mid x) > 0\}$$
$$C_\pi Y = C_\pi^\infty Y$$

where $P^\pi(y|x) = P(y|x, \pi(x))$. In this formulation, $C_\pi^k Y$ is the set of states that can be reached in k or fewer steps from Y and $C_\pi Y$ is the set of states that can be reached from Y in any number of states. This can be computed using graph search methods.

The evaluation set X_t is sufficient to evaluate V^{π_t} at any x in X_t:

$$V^{\pi_t}(x) = R^{\pi_t}(x) + \gamma \sum_{y\in X} P^{\pi_t}(y \mid x) V^{\pi_t}(y)$$

and by closure we are guaranteed that if x is in X_t and P^{π_t} (y\x) > 0, then y is also in X_t. However, X_t is not sufficient to perform policy improvement, since this requires computing

$$\left(\Gamma_a V^{\pi_t}\right)(x) = R(x,a) + \gamma \sum_{y\in X} P(y \mid x, a) V^{\pi_t}(y) \tag{15}$$

for any a, and there may be successors states that are not in the evaluation set. Therefore, it is necessary to construct the evaluation envelope Y_t.

In order to evaluate the operator in Equation (15), Y_t must contain all of the successor states of X_t under every action,

$$\bigcup_{a \in A} C_a^1 X_t$$

If x is in X_t, then for every action a, if $P(y|x, a) > 0$, then we include y in our improvement envelope. This allows us to perform the backup in Equation (15) given V^{π_t}, but is not sufficient to evaluate the policy's value function at the states that are in the improvement envelope but not in the evaluation set, since the current policy may take the system outside of the improvement envelope. In order to address this, we close the improvement envelope under the policy, so that

$$Y_t = C_{\pi_t}\left(\bigcup_{a \in A} C_a^1 X_t\right)$$

At this point, Y_t is sufficient to perform policy evaluation, which is sufficient to perform policy improvement over X_t.

Thus far, we have ignored computability of the closure operator. In general, the closure may be infinite. For example, in the thread scheduling problem the policy $\pi = a$ has infinite closure on any non-empty state set. We discuss this issue in detail in Section 6.

5.3 Policy evaluation

Policy evaluation in ESPI has only minor differences from its standard PI counterpart. There are two differences, which have to do with the policy and state representations.

In a finite MDP, a policy can be represented using a lookup table. In ESPI we cannot represent a policy for every state using a lookup table. Instead at each iteration $t > 0$, we only store the current policy π_t explicitly for the evaluation envelope X_{t-1} from the previous iteration. Everywhere else, we use an implicitly defined default policy π_0. For example, the greedy policy $\pi_0(x) = \text{argmax}_{a \in A}\{R(x, a)\}$ can be computed cheaply given the problem model.

The second difference is that we are only able to evaluate the value function for a given policy over a finite subset of the state space. In Algorithm 1, we indicate this by computing partial value functions V_t that are defined only over the improvement envelope Y_t. Since Y_t is closed under π_t, V_t and V^{π_t} agree everywhere in the improvement envelope.

In Equation (14), we define policy evaluation as a system of linear equations that can be solved to obtain V^{π_t}. The transition matrix P^{π_t} and reward vector R^{π_t} are infinite dimensional objects, so we perform policy evaluation with respect to their restrictions to the improvement envelope Y_t, the $|Y_t| \times |Y_t|$ matrix $P^{\pi_t}_{Y_t}$ and $|Y_t|$-vector R^{π_t}.

5.4 Policy improvement

As with policy evaluation, policy improvement remains fundamentally unchanged from the standard algorithm. We define the restricted per-action backup operator $\Gamma_{a,Y}$ for action a on the state set Y and function V: Y $\rightarrow \mathbb{R}$ as

$$\left(\Gamma_{a,Y}V\right)(x) = R(x,a) + \gamma \sum_{y \in Y} P(y \mid x,a)V(y)$$

Since Y_t contains all of the successors of x under action a for any state x in the evaluation set X_t, $(\Gamma_{a,Y}\, V)(x) = (\Gamma_a\, V)(x)$.

As we described above, we cannot represent the policy everywhere, but after the policy iteration step π_{t+1} is defined explicitly on every state in the evaluation set. Since π_0 is invariant to iterations, we only need to check if π_{t+1} changed for any $x \in X_t$; if it did not, then we terminate ESPI.

6 Discussion

Termination: since ESPI continues until it encounters the same policy twice, a natural consideration is whether or not the algorithm terminates. Two conditions are necessary to guarantee this:

1 the closures X_t and Y_t computed at each iteration are finite

2 ESPI requires only finitely many iteration before the policy does not change.

We will show that the closures X_t and Y_t are finite under certain assumptions about the default policy π_0 and the initial state set X_I, and provide an argument that the algorithm will terminate after finitely many iterations.

We will first show that the closures are finite. The intuition is that since $r(x)$ is the distance from the target utilisation ray, $\|x - \|x\|_1 u\|_1$, these states form a tube. Because of the way that states wrap, the tube has finite size.

Lemma 2: Let $T_\rho = \{x \in X : \|x - \|x\|_1 u\|_1 \leq \rho\}$ be a tube with radius ρ. For any $\rho < \infty$, $|T_\rho| < \infty$.

Proof: Let $X_\tau = \{x \in X : \| x \|_1 = \tau\}$ for non-negative integer τ. Since for any $x \in X_\tau$ each component x_a is between 0 and τ, $|X_\tau| \leq \tau^n$ is finite. Let $T_{\rho,\tau} = X_\tau \cap T_\rho$. We have $T_\rho = \bigcup_{\tau=0}^{\infty} T_{\rho,\tau}$, so the remainder of our proof consists of showing that for sufficiently large τ, $T_{\rho,\tau}$ is empty.

Let $x \in X_\tau$. This means that $w(x) = x$, which implies that for some thread a, $x_a < k u_a$. If we choose $\tau \geq k$, then $x_a < k u_a < \tau u_a$. Since $\| x - \tau u \|_1 = \sum_{b \in A} | x_b - \tau u_b |$, we have

$$\| x - \tau u \|_1 \geq | x_a - \tau u_a | = \tau u_a - x_a > \tau u_a - k u_a$$

Therefore, if for all a, $\tau \geq \rho/u_a + k$, then $\|x - \tau u\|_1 > \rho$. It follows that for all $\tau \geq \rho/u_a + k$, if x is in X_τ, then it can not be in T_ρ, so $T_{\rho,\tau}$ is empty. Therefore T_ρ is the finite union of finite sets, and so is finite.

Using Lemma 2, we can show that any finite set $Y \subset X$ has finite closure under any policy π in a particular class of attractive policies. The necessary attractive condition is that π can put the system in a worse state – one that is farther from target utilisation – in only finitely many states. Since this set is finite, we can draw a tube that contains both these states and Y. All possible successors of states in this tube under π lie inside an outer tube since run-times are bounded above. In this outer tube, π is guaranteed to push the system towards target utilisation, so it contains its own closure. Since Y's closure is inside this tube, Lemma 2 implies that it is finite. We now state this result formally:

Lemma 3. Let $Y \subset X$ be finite and let π be a policy. Define

$$U_\pi = \{x \in X : \exists y \in s(x, \pi(x)), \| x - \| x \|_1 u \|_1 < \| y - \| y \|_1 u \|_1\} \tag{16}$$

as the set of states in which π may result in a state that is farther from target utilisation, where $s(x, a)$ is the set of successors of x under action a. If U_π is finite, then $C_\pi Y$ is finite.

Proof: Let $\rho = \max\{\| x - \| x \|_1 u \|_1 : x \in Y \cup U_\pi\}$ be the maximum distance from target utilisation among states in Y and U_π, so that $U_\pi \cup Y \subseteq T_\rho$. Let x be a state inside this tube, and let $y = s(x, \pi(x), t)$ be a successor of x for some run-time $t \in \{1, 2, \ldots, \beta\}$. Then

$$\begin{aligned} \| y - \| y \|_1 u \|_1 &= \| s(x, \pi(x), t) - \| s(x, \pi(x), t) \|_1 u \|_1 \\ &= \| x + t\Delta_{\pi(x)} - \| x \|_1 u - tu \|_1 \\ &\leq \| x - \| x \|_1 u \|_1 + t \| \Delta_{\pi(x)} - u \|_1 \\ &\leq \rho + 2\beta \end{aligned}$$

which follows from $t \leq \beta$ and $\|\Delta_a - u\| < 2$. Since x was chosen arbitrarily, this implies that the successors of T_ρ under π are contained inside an outer tube $T_{\rho + \beta n}$. Thus, the successors of any state in $T_{\rho + \beta n}$ are also in $T_{\rho + \beta n}$: if it is in T_ρ, then we have shown above that its successors are in $T_{\rho + \beta n}$, and if it is in $T_{\rho + \beta n} - T_\rho$, then π always decreases the distance to target utilisation, and so remains $T_{\rho + \beta n}$. Therefore, $C_\pi Y \subseteq T_{\rho + \beta n}$, and so $C_\pi Y$ is finite.

Lemma 3 is sufficient to guarantee that the closures $X_t = C_{\pi_t} X_I$ and $Y_t = C_{\pi_t}(\bigcup_{a \in A} C_a^1 X_t)$ are finite. We require that the initial states X_I policy π_0 provided to ESPI satisfy the conditions of the lemma. Then, we can proceed inductively: finiteness of $X_0 = C_{\pi_0} X_I$ follows from the lemma. $\bigcup_{a \in A} C_a^1 X_0$ is finite since X_0 is finite and every state has finitely many successors, so Y_0 is finite. Suppose X_t is finite. π_{t+1} is explicitly defined only on X_t, elsewhere it is equivalent to π_0, so it satisfies the conditions of the lemma. Thus $X_{t+1} = C_{\pi_t} X_I$ is finite, and since X_{t+1} is finite Y_{t+1} is finite by the same argument that demonstrated finiteness of Y_1

The above guarantees that each iteration of ESPI terminates. This is necessary but not sufficient to guarantee termination, since if we it was able to improve the policy at each iteration, it would still run forever. We will argue informally that this can not occur. First, notice by the monotonicity lemma from, for instance, Bertsekas and Tsitsiklis (1996), we are guaranteed that π_{t+1} is no worse than π_t on any state. This result continues to hold for

ESPI because the restricted backup operators Γ_a and Γ_{π_t, Y_t} agree with the unrestricted operators Γ_a and Γ_π at every state in X_t and Y_t, respectively. This means that the policies considered are strictly increasing in quality, since if π_{t+1} and π_t have the same value on every state in X_t, we terminate the algorithm.

Monotonicity implies that ESPI always strictly improves its policy between iterations. As a consequence, if ESPI does not terminate, it must expand larger and larger parts of the state space. To see this, suppose we could draw a tube T_ρ with finite radius that contains every X_t expanded. Since T_ρ is finite and the set of threads are both finite, there are only finitely many policies that we could consider, and we would terminate. Therefore, we cannot draw a bounding tube. This means that if ESPI does not terminate, it must add progressively worse states – states with poor utilisation – to the state closures. This means that the corresponding policies have some probability of entering these poor states. Since run-times are bounded, the successors of these states are also quite poor, so there is little value to expanding into these states. Therefore, we expect that for some sufficiently large tube, the cost of any trajectory that visits states outside of this tube would be prohibitively expensive, and so any policy that does so would be worse than a policy that does not leave this tube.

It is important to note that these results are only sufficient to guarantee that ESPI provides a locally optimal policy, since at every state in the closure of the final policy there is no action that would improve it with respect to the policy's value function restricted to a finite state set, but thus far we have not been able to demonstrate that a two-step or more trajectory would fail to improve the policy. This would be the case if, for example, it was necessary to move into a high penalty state in order to reach a state with penalty near zero. It seems that the structure of the thread scheduling problem ought to prohibit this case.

Verification: because verification relies on a closure operator equivalent to that employed in the ESPI algorithm, it is implicit that we can apply verification techniques to the policies obtained by our methods.

Verification of properties over policies can be accomplished by using model checking techniques over the state space formed by the closure of the initial states over the final policy. Queries, in the form of temporal logic expressions can be evaluated over this state space provided that:

1 all reachable states are present, which is guaranteed by the closure property

2 the transition between the states is known, which can be derived in a straight-forward fashion from the state space enumeration, the wrapping function and the thread run-time distributions.

In fact, it would be possible to perform verification at each iteration of the ESPI algorithm if the computational costs are not prohibitive. For example, if it turned out the final policy did not satisfy some desired property it would be possible to 'roll back' to the latest policy that did. At each iteration in the ESPI algorithm the current policy would run through a verification step and store that policy if it does violate the desired properties. Since each policy is guaranteed to be better than the policies produced in previous iterations, we would only need to store the last such policy.

One potential alternative to storing known good previous policies would be to use violation of desired properties as a mechanism for performing policy improvement. This would allow us to restrict ourselves to exploration of a space of policies that satisfies such properties.

7 Conclusions and future work

Our research has developed an approach to scheduling policy design for autonomic systems, in which we first derive a utilisation state space model that is particular to the thread behaviour in each distinct system. Taking advantage of the periodic similarity of utilisation states we then condense them into wrapped state models, which is a crucial advance towards efficient design and analysis of scheduling policies. Together, these are important steps towards designing verified autonomic systems with specialised scheduling policies, which we have reported in our previous work (Glaubius et al., 2008; Tidwell et al., 2008).

In this paper, we have extended that approach with ESPI, an algorithm for generating scheduling policies from the wrapped state models. We have shown that under reasonable assumptions, the algorithm terminates with a locally optimal policy. We also provide an intuition for how in future work verification could be incorporated within each iteration of the algorithm, for checking policies and using violations as a mechanism for policy iteration.

Acknowledgements

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'.

References

ARINC Incorporated Annapolis (1997) Maryland, USA: Document No. 653: Avionics Application Software Standard Inteface (Draft 15). Available at: www.arinc.com.

Aswathanarayana, T., Subramonian, V., Niehaus, D. and Gill, C. (2005) 'Design and performance of configurable endsystem scheduling mechanisms', Paper presented in the *Proceedings of the 11th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*.

Bertsekas, D.P. and Tsitsiklis, J.N. (1996) *Neuro-Dynamic Programming*, Athena Scientific.

Garfinkel, T. (2003) 'Traps and pitfalls: practical problems in system call interposition based security tools', Paper presented in the *Proceedings of the Network and Distributed Systems Security Symposium*.

Garfinkel, T., Pfaff, B. and Rosenblum, M. (2004) 'Ostia: a delegating architecture for secure system call interposition', Paper presented in the *Proceedings of the Network and Distributed Systems Security Symposium*.

Glaubius, R., Tidwell, T., Smart, W.D. and Gill, C. (2008) 'Scheduling design and verification for open soft real-time systems', *29th IEEE International Real-Time Systems Symposium (RTSS '08, to appear)*, Barcelona, Spain.

Goldberg, I., Wagner, D., Thomas, R. and Brewer, E.A. (1996) 'A secure environment for untrusted helper applications', Paper presented in the *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA.

Goyal, G.V. (1996) 'A hierarchical CPU scheduler for multimedia operating systems', *2nd Symposium on Operating Systems Design and Implementation, USENIX*.

Held, M. and Karp, R.M. (1962) 'A dynamic programming approach to sequencing problems', *Journal of the Society for Industrial and Applied Mathematics*, Vol. 10, No. 1, pp.196–210.

Irvine, C.E. (1999) *The Reference Monitor Concept as a Unifying Principle in Computer Security Education*. Available at: citeseer.ist.psu.edu/2 9 93 00.html.

Jones, M. (1997) *What Really Happened on Mars?* Available at: www.research.microsoft.com/~ mbj/Mars_Pathfinder/Mars_Pathfinder.html.

Kesten, Y., Pnueli, A., Sifakis, J. and Yovine, S. (1999) 'Decidable integration graphs', *Information and Computation*, Vol. 150, No. 2, pp.209–243.

Littman, M.L., Ravi, N., Fenson, E. and Howard, R. (2004) 'Reinforcement learning for autonomic network repair', Paper presented in the *Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004)*, pp.284–285.

Liu, J.W.S. (2000) *Real-Time Systems*. New Jersey, NJ: Prentice Hall.

Vanfleet, W.M., Luke, J.A., Beckwith, R.W., Taylor, C., Calloni, B. and Uchenick, G. (2005) *MILS: Architecture for High-Assurance Embedded Computing*, Available at: http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html (Crosstalk: *The Journal of Defense Software Engineering*, August).

Martin, W., White, P., Taylor, F.S. and Goldberg, A. (2000) 'Formal construction of the mathematically analyzed separation kernel', *ASE '00: Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, Washington, DC: IEEE Computer Society, p.133.

Migliaccio, A., Tidwell, T., Gill, C., Aswathanarayana, T. and Niehaus, D. (2005) '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.

Provos, N. (2003) 'Improving host security with system call policies', *12th USENIX Security Symposium*. Washington, DC: USENIX Association.

Puterman, M.L. (1994) *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, Wiley Interscience.

Regehr, S. and Stankovic, J.A. (2001) 'HLS: a framework for composing soft real-time schedulers', *22nd IEEE Real-time Systems Symposium*, London, UK.

Regehr, J., Reid, A., Webb, K., Parker, M. and Lepreau, J. (2003) 'Evolving real-time systems using hierarchical scheduling and concurrency analysis', *24th IEEE Real-time Systems Symposium*, Cancun, Mexico.

Rust, J. (1996) 'Numerical dynamic programming in economics', in H.M. Amman, D.A. Kendrick and J. Rust (Eds.) *Handbook of Computational Economics*, Elsevier, Vol. 1, pp.619–729.

Singh, S.P. and Yee, R.C. (1994) 'An upper bound on the loss from approximate optimal-value functions', *Machine Learning*, Vol. 16, No. 3, pp.227–233.

Sutton, R.S. and Barto, A.G. (1998) *Reinforcement Learning: An Introduction*, MIT Press.

Tesauro, G. (2007) 'Reinforcement learning in autonomic computing: a manifesto and case studies', *IEEE Internet Computing*, Vol. 11, No. 1, pp.22–30.

Tesauro, G., Jong, N.K., Das, R. and Bennani, M.N. (2007) 'On the use of hybrid reinforcement learning for autonomic resource allocation', *Cluster Comput*ing, Vol. 10, No. 3, pp.287–299.

Tidwell, T., Gill, C. and Subramonian, V. (2007) '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.

Tidwell, T., Glaubius, R., Gill, C. and Smart, W.D. (2008) 'Scheduling for reliable execution in autonomic systems', *5th International Conference on Autonomic and Trusted Computing (ATC '08)*, Oslo, Norway.

Tidwell, T., Watkins, N., Subramonian, V., Niehaus, D., Gill, C. and Migliaccio, A. (2006) '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.

Whiteson, S. and Stone, P. (2004) 'Adaptive job routing and scheduling', *Engineering Applications of Artificial Intelligence*, Vol. 17, No. 7, pp.855–869.

Zhang, Y., Lu, C., Gill, C., Lardieri, P. and Thaker, G. (2007) 'Middleware support for a periodic tasks in distributed real-time systems', *RTAS '07: Proceedings of the 13th IEEE Real Time on Embedded Technology and Applications Symposium*, Washington, DC, USA, IEEE Computer Society, pp.497–506.