

Formal Methods for Dynamic Power Management: A Tutorial

Gethin Norman¹ David Parker¹ Marta Kwiatkowska¹
Sandy Irani² Sandeep Shukla³ Rajesh Gupta⁴

¹ University of Birmingham, Birmingham, UK

² University of California, Irvine, USA

³ Virginia Tech, Blacksburg, USA

⁴ University of California, San Diego, USA

Abstract. Dynamic power management or DPM refers to the runtime strategies used to minimize power consumption and to find the right performance/power tradeoff for a system and/or its components. We survey the two primary formal approaches to DPM: online algorithms and stochastic optimization. The first approach is based on the ideas of on-line algorithms and competitive analysis. We show how competitive analysis of the problem helps us devise near optimal strategies for power management with theoretically proven bounds on competitive ratios. We also prove these bounds as tight bounds by formally establishing relevant lower bounds. In the stochastic modeling approach DPM problem is modeled as the problem of finding optimal stochastic strategies, with probabilistic guarantees on performance. We touch upon some earlier methods for deriving stochastic DPM strategies from the literature, followed by an introduction to probabilistic model checking as a formal framework in which such strategies can be derived quite naturally. We present an introduction to the probabilistic model checking tool PRISM, developed at the University of Birmingham, and show how it is used to derive and analyze stochastic DPM strategies.

1 Introduction

1.1 Dynamic Power Management

The nature of computing has been changing over the last few years from server, workstation and desktop based computing to embedded, ubiquitous and pervasive computing. Handheld devices, wireless sensors and biomedical devices are gaining more and more prominence in all arenas of human life. However, as we move from wired to wireless, power savings in these computing devices become more crucial. As a result, much research has been done in the area of low-power design, power management and the balance between computation and communication power. Each kind of approach to power savings has its own limitations. For example, circuit-level or architecture-level mechanisms cannot take advantage of application characteristics. As a result, system-level power management, which is characterized by operating system controlled power saving measures, based on

the observation of application characteristics, has gained significant attention in the last few years. There are two distinct flavors of system-level power management: Dynamic Voltage/Frequency scaling (DVS/DFS) and Dynamic Power Management (DPM). In this tutorial, we will focus on the latter.

Dynamic Power Management (DPM) is a way to save energy in devices that can be turned on and off under operating system control. This approach is quite distinct from DVS or DFS. DPM has gained considerable attention over the last few years, a trend evidenced in the research literature [17, 42, 9, 8, 40, 38, 11, 39], as well as concerted industry efforts such as Microsoft's OnNow [27] and ACPI [18]. Due to the importance of the minimization of power consumption in today's embedded systems, a lot of work has been initiated in both the component manufacturing industry and the systems design industry.

A survey of most of the techniques developed for DPM before 2000 can be found in [6]. In this extensive review, the solution approaches to DPM have been classified into *predictive schemes* and *stochastic optimum control* schemes. Predictive schemes attempt to predict a device's usage behavior in the future, usually based on the past history of usage patterns, and decide to change power states of the device accordingly. Stochastic approaches make probabilistic assumptions (based on observations) about usage patterns and exploit the nature of the probability distribution to formulate an optimization problem, the solution to which drives the DPM strategy.

It has been noted that predictive schemes are mostly based on devices with two power saving states, whereas there are many instances of devices in the embedded world which have multiple states. Examples of such devices may be found in [6, 41]. In order to provide DPM strategies for multi-state systems, the stochastic optimum control approach has been proposed in the literature [6, 41, 7, 11, 35, 37]. However, such stochastic approaches also have their drawbacks, including the fact that they make many assumptions about the probabilistic nature of the inputs and may be more computationally expensive to implement.

In the literature, one can find many strategies proposed and evaluated for DPM, such as predictive strategies [17, 42], stochastic modeling based strategies [7, 35], session clustering and prediction strategies [26], on-line strategies [38], and adaptive learning based strategies [12]. In [25], one can find a quantitative comparison between various existing management strategies. Any strategy for an individual idle period can be described by a threshold on the idle time before transitioning from the active to the sleep state. For multiple state systems, there is a sequence of thresholds each of which indicates when to transition to the next lower power consumption state. Previous work on prediction based dynamic power management can be categorized into two groups: adaptive and non-adaptive. Non-adaptive strategies set the thresholds for the algorithm once and for all and do not alter them based on observed input patterns. Adaptive strategies, on the other hand, use the history of idle periods to guide the decisions of the algorithm for future idle periods. There have been a number of adaptive strategies proposed in the literature [17, 21, 7, 11].

Many adaptive dynamic power management strategies [17, 42, 38, 21, 11, 26] use a sequence of past idle period lengths to predict the length of the next idle period. These strategies typically describe their prediction for the next idle period with a single value. Given this prediction, they transition to the power state that is optimal for this specific idle period length. In the case that their prediction is wrong, they transition to the lowest power state if the idle period extends beyond a fixed threshold value. For the sake of comparison with other approaches, we shall call these predictive DPM schemes *single-value prediction schemes (SVP)*. Of particular interest is the work of Chung, Benini and De Micheli [12] that addresses multiple idle state systems using a prediction scheme based on adaptive learning trees. Their method has an impressively high hit ratio in its prediction.

1.2 Formal Methods for DPM

Most of the previous work on dynamic power management has been based on ad-hoc techniques, such as the use of regression equations over the previous few idle periods to predict the next idle period, interpolation techniques or learning based techniques. The stochastic DPM literature tends to be more formal in the sense that assumptions are made as to the characteristics of the probability distribution of idle periods, device response times etc. These are then used to formulate optimization problems. Much of the stochastic DPM strategy literature uses Markov models, based on assumptions about how and when requests can arrive (whether at certain time points or at any time). For example, discrete- and continuous-time Markov chains have been used. However, even though such work can be categorized as based on formal foundations, formal methods tend to be based more on formal modeling techniques, complexity analysis and usage of a formal framework. As a result, in this tutorial we do not discuss this previous work in detail. In the absence of formal methods, most of the previous DPM work has been evaluated with simulation techniques which are time consuming and often not completely reliable. With formal models and formal techniques such as competitive analysis or model checking, one can give precise bounds on the performance of the algorithms or strategies. In the case of probabilistic model checking, one can avoid simulation completely and find out probabilistic guarantees on the behavior and performance of the strategies.

1.3 Organization

The remainder of this tutorial paper is organized as follows. Section 2 describes the basic concepts of on-line algorithms and competitive analysis in the context of DPM. It introduces a system model which abstracts real ACPI compliant devices, gives a deterministic on-line algorithm and a probabilistic learning based approach for DPM, as well as the corresponding competitive analysis. Section 3 introduces the stochastic approach to DPM, with references to the existing literature. Section 4 describes the basics of probabilistic model checking and the PRISM tool. Section 5 shows in detail how to derive probabilistic DPM strategies

using probabilistic model checking and PRISM. Finally, Section 6 concludes the tutorial.

2 On-line Algorithms, Competitive Analysis and DPM

In this approach, dynamic power management is considered to be an inherently *online* problem, in that an algorithm governing power management must make decisions about the expenditure of resources before all the input to the system is available [1]. In particular, for DPM strategies, the input is the length of an upcoming idle period and the decision to be made is whether to transition to a lower power dissipation state while the system is idle. For instance, the intervals between data transfer requests to a radio subsystem in a packet radio network interface are not known in advance. It may not be an effective power reduction strategy to shutdown power to such a device as soon as it is detected to be idle. If the idle period is too small, the power-up cost could surpass the energy saved by shutting down the subsystem. On the other hand, it is important to power down for long idle periods when the device is not in use. Analytical solutions to such online problems are often best characterized in terms of a *competitive ratio* [32] that compares the cost of an online algorithm to the optimal offline solution which knows the input in advance.

Earlier work on competitive analysis of dynamic power management strategies presents bounds on the quality of DPM solutions [38, 21]. Competitive analysis has proven to be a powerful tool in providing a guarantee on the performance of an algorithm for any input. There are two chief limitations of this earlier work:

- These previous works are based on two-state devices, real systems often have multiple idle states with transition energy costs that must be taken into account.
- The competitive ratio is very similar to the lower bounds in algorithmic complexity theory, which means that the ratio applies to a worst case scenario and hence may not reflect the average behaviour of the strategy.

These two limitations are addressed in the competitive analysis approach which follows. We present analytical bounds on the performance of strategies for systems with multiple idle states. We also provide an experimental foundation that makes use of competitive ratio as an experimental measure of the performance of DPM strategies and show that, in practice, the competitive ratio could be well below the theoretically established worst case bound.

All of the previous work on competitive analysis for dynamic power management has concentrated on two-state systems [21, 22, 38]. We say that an algorithm is c -competitive if, for any input, the cost of the online algorithm is bounded by c times the cost of the optimal offline algorithm for that input. The *competitive ratio* of an algorithm is the infimum over all c such that the algorithm is c -competitive. It has been proven that 2 is the best competitive ratio achievable by any deterministic online algorithm [32]. We extend this analysis

to show a 2-competitive algorithm for the multi-state case. This result is tight in the sense that there is no constant $c < 2$ such that there is a deterministic c -competitive algorithm which works for all multiple power down state systems. However, it may be possible to have a competitive ratio less than c for a specific system, depending on the parameters of the system (e.g. number of states, power dissipation rates, start-up costs, etc.) Note that this algorithm is non-adaptive since it does not use any information about the arrival sequence of jobs to the device.

As discussed above, competitive analysis often gives overly pessimistic bounds for the behaviour of algorithms. This is inherently the result of the fact that competitive analysis is a worst-case analysis. Competitive analysis still has great value in situations where it is impractical to obtain and process information for predicting future inputs. However, in many applications there is structure in the input sequence that can be utilized to fine tune online strategies and improve their performance. Indeed, earlier work [7, 35] has relied on modeling the distribution governing inter-arrival times as an exponential distribution. In practice, such stochastic modeling seems to hold well for specific kinds of applications. However, these assumptions have led to complications in other settings due to such phenomena as the non-stationary nature of the arrival process, clustering, and the lack of independence between subsequent events. These problems have been addressed to some extent in [26, 12].

In [19], we introduced an approach that models the upcoming input sequence by a probability distribution that is *learned* based on historical data. One of the strengths of this method is that we make no assumptions about the form of this distribution. Once the distribution is learned, we can automatically generate a probability-based DPM strategy that minimizes the expected power dissipation given that the input is generated according to that distribution. We compare the expected power dissipation of our online algorithm to that of the optimal offline algorithm to get a *probabilistic competitive ratio*. This method has been used in the context of two-state systems [21, 22]. We generalize this work for multi-state systems.

2.1 The System Model

We focus on strategies for a single peripheral device whose power is managed by the operating system. The device can be in one of the n power states denoted by $\{s_1, \dots, s_n\}$. The power consumption for state i is denoted by α_i . The states are ordered so that $\alpha_i > \alpha_j$ as long as $i < j$. Thus, state s_1 is the *ready* state which is the highest power consumption state.

We are also given, from the manufacturer's specification, the transition power p_{ij} , and transition times t_{ij} , to move from state s_i to s_j . Usually, the power needed and time spent to go from a higher state to a lower state is negligible, whereas the power and time needed to transition to a higher state from a lower state is high. Thus, we simplify the model by considering only the time and power necessary to power up the system. Furthermore, all of the algorithms considered in this tutorial have the property that they only transition to the ready state

when powering up and never transition to an intermediate higher powered state. As a result, we only need the time and total energy consumed in transitioning up from each state i to the ready state. The total energy used in transitioning from state i to the ready state is denoted by β_i .

We note that, in cases where the time and energy used in transitioning to lower power consumption states is non-negligible, they can be easily incorporated by folding them into the corresponding power-up parameters. This can be done as long as the time and energy used in transitioning down is additive. That is, we require that for $i < j < k$, the cost to go from i to j and then from j to k is the same as the cost of going from i directly down to k .

The input to the DPM is a sequence of requests for service that arrive through time. With each request, we are told the time of its arrival and the length of time it will take to satisfy the request. If the device is busy when a new request arrives, it enters a queue and is served on a first-come-first-serve basis. In this case, there is no idle period and the device remains active through the time that the request is finished. This means that the number of idle periods is generally less than the number of requests serviced. Whenever a request terminates and there are no outstanding requests waiting in the system, an idle period begins. In these situations, the DPM is invoked to determine to which power consumption states the device should transition and at what times.

If the device is not busy when a new request arrives, it will immediately transition to the ready state to serve the new request if it is not already there. In the case where the device is not already in the ready state, the request can not be serviced immediately, but will have to incur some latency in waiting for the transition to complete. This delay will cause future idle periods to be shorter. In fact, if a request is delayed, some idle periods may disappear. Thus, we have an interesting situation where the behaviour of the algorithm effects future inputs (idle period lengths) given to the algorithm.

Another interesting phenomenon is that delaying servicing a request will tend to result in lower power usage. Consider the extreme case where the power manager remains in the deepest sleep state while it waits for all the requests to arrive and then processes them all consecutively. This extreme case is not allowed to happen in our model since we require that the strategy transition to the ready state as soon as any request appears. However, it illustrates the natural trade-off which occurs between power consumption and latency. See [38] for a more extensive discussion of this trade-off. Our experimental results explore this trade-off for the set of algorithms studied.

In the next two sections we present our analysis for the deterministic and probability-based algorithms. In this analysis, we do not take into account the delay incurred in returning to the ready state. This is because the analysis focuses on an individual idle period. These sections only address the problem of minimizing total energy expenditure given that the length of the upcoming period is governed by a fixed probability distribution (probability-based case) or is arbitrary (deterministic case). The empirical evaluations do examine the effect of start-up latency as multiple requests for service arrive through time.

2.2 The Deterministic Algorithm

To get the optimal cost, plot each line $c = \alpha_i t + \beta_i$. This is the cost of spending the entire interval in state i as a function of t , the length of the interval. Take the lower envelope of all of these lines. Call this function $LE(t)$. The optimal cost for an interval of length t is $LE(t) = \min_i \{\alpha_i t + \beta_i\}$. The online algorithm called the Lower Envelope Algorithm (LEA) will also follow the function LE . It will remain in the state which realizes the minimum in LE and will transition at the discontinuities of the curve. That is, LEA will remain in state j as long as $\alpha_j t + \beta_j = \min_i \{\alpha_i t + \beta_i\}$, for the current time t .

For $j = k$ to 1, let t_j be the solution to the equation $\alpha_j t + \beta_j = \alpha_{j-1} t + \beta_{j-1}$. t_j is the time that LEA will transition from state j to state $j - 1$. We will assume here that we have thrown out all the lines which do not appear on the lower envelope at some point. This is equivalent to the assumption that $t_k < t_{k-1} < \dots < t_2 < t_1$. (See Figure 1).

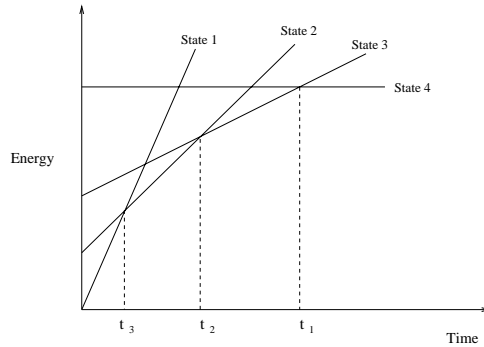


Fig. 1. Energy consumption for each state of a four state system. Each state is represented by a line which indicates the energy used if an algorithm stays in that state as a function of the length of the idle period. For each state, the slope is the power dissipation rate and the y intercept is the energy required to power up from that state.

Theorem 1. *The Lower Envelope algorithm is 2-competitive.*

The proof of Theorem 1, can be found in [19]. As emphasized earlier, this algorithm, does not take into account input patterns. Thus, the worst case scenario, obtained via Theorem 1, shows that the energy cost resulting from the on-line decisions can be no worse than 2 times the energy cost of the optimal offline strategy which knows the input sequence in advance. We show later that, depending on request arrival patterns, this worst case bound may not really happen, and the empirical ratio of the online to offline costs may be much lower.

However, as shown in [35, 12, 7], input sequences are often interrelated, and hence modeling of the input pattern and exploiting that knowledge in the design of the algorithm can help bridge the gap between the performance of online

strategy and that of the optimal offline strategy. In the next subsection, we discuss our probability-based algorithm and show that if the request inter-arrival time probability distribution is known before hand, the worst case competitive ratio can be improved by 21%, with respect to the deterministic case. Moreover, simulation based experimental evaluation [19] shows that this worst case bound is pathological. In fact, we can bring the energy cost of the online algorithm within 27% of the optimal offline one.

2.3 The Probability-Based Algorithm

Optimizing Power Based on a Probability Distribution. In this section, we assume that the length of the idle interval is generated by a fixed, known distribution whose density function is π . We first discuss systems with two states and then give our generalization to the multi-state case. let β be the start-up energy of the sleep state and α the power dissipation of the active state. Suppose that the online algorithm uses τ as the threshold at which time it will transition from the active state to the sleep state if the system is still idle. In this case, the expected energy cost for the algorithm for a single idle period will be

$$\int_0^\tau \pi(t)(\alpha t)dt + \int_\tau^\infty \pi(t)[\alpha\tau + \beta]dt.$$

The best online algorithm will select a value for τ which minimizes this expression. The offline algorithm which knows the actual length of an upcoming idle period will have an expected cost of

$$\int_0^{\beta/\alpha} \pi(t)(\alpha t)dt + \int_{\beta/\alpha}^\infty \pi(t)\beta dt.$$

It is known for the two state case, that the online algorithm can pick its threshold τ so that the ratio of its expected cost to the expect cost of the optimal algorithm is at most $e/(e-1)$ [21, 22]. That is, for any π , and any α and β ,

$$\frac{\min_\tau \left\{ \int_0^\tau \pi(t)(\alpha t)dt + \int_\tau^\infty \pi(t)[\alpha\tau + \beta]dt \right\}}{\int_0^{\beta/\alpha} \pi(t)(\alpha t)dt + \int_{\beta/\alpha}^\infty \pi(t)\beta dt} \leq \frac{e}{e-1}.$$

This is optimal in that for any α and β , there is a distribution π such that this ratio is at least $e/(e-1)$.

Let us now consider the multi-state case. As in the previous section, let t_j be the solution to the equation $\alpha_j t + \beta_j = \alpha_{j-1} t + \beta_{j-1}$. t_j is the time that LEA will transition from state j to state $j-1$. We will assume here that we have thrown out all the lines which do not appear on the lower envelope at some point. This is equivalent to the assumption that $t_k < t_{k-1} < \dots < t_2 < t_1$. For ease of notation, we will define t_{k+1} to be 0 and t_0 to be ∞ . The cost (expected energy consumption) of the optimal offline algorithm is:

$$\sum_{i=0}^k \int_{t_{i+1}}^{t_i} \pi(t)[\alpha_i t + \beta_i] dt.$$

Now to determine the online algorithm, we must determine k thresholds, where the threshold τ_i is the time at which the online algorithm will transition from state i to state $i - 1$. In the spirit of the deterministic online algorithm for the multi-state case, we will let τ_i be the same as the threshold which would be chosen if i and $i - 1$ were the only two states. We call this algorithm the Probability-based Lower Envelope Algorithm (PLEA). The proof of the following theorem appears in [19].

Theorem 2. *For any distribution, the expected cost of the Probability-based Lower Envelope Algorithm is within a factor of $e/(e - 1)$ of the expected cost for the optimal offline algorithm.*

Learning the Probability Distribution. The algorithm which uses PLEA in conjunction with our scheme described below which learns the probability distribution is called Online Probability-Based Algorithm (OPBA). It works as follows: a window size w is chosen in advance and is used throughout the execution of the algorithm. The algorithm keeps track of the last w idle period lengths and summarizes this information in a histogram. Periodically, the histogram is used to generate a new power management strategy.

The set of all possible idle period lengths $(0, \infty)$ is partitioned into n intervals, where n is the number of bins in the histogram. Let r_i be the left endpoint of the i^{th} interval. The i^{th} bin has a counter which indicates the number of idle periods among that last w idle periods whose length fell in the range $[r_i, r_{i+1})$. The bins are numbered from 0 through $n - 1$. $r_0 = 0$ and $r_n = \infty$.

The last w idle periods are held in a queue. When a new idle period is completed, the idle period at the head of the queue is deleted from the queue. If this idle period falls in bin i , then the counter for bin i is decremented. The new idle period is added to the tail of the queue. If this idle period length falls into bin j , the counter for bin j is incremented. Thus, the histogram always includes data for the last w idle periods. In [19] simulation results include a study experimenting with values for w . Those results show that the algorithm is very robust under a variety of different ranges for the window size, w .

The counter for bin i is denoted by c_i . The threshold is selected among n possibilities: r_0, \dots, r_{n-1} (the lower end of each range). We estimate the distribution π by the distribution which generates an idle period of length r_i with probability c_i/w for each $i \in \{0, \dots, n-1\}$. The sum of the counters is the window length

w . Thus, the threshold is taken to be

$$\arg \min_{r_t} \sum_{j=1}^{t-1} \left(\frac{c_j}{w} \right) r_j (\alpha_i - \alpha_{i-1}) dt \\ + \sum_{j=t}^n \left(\frac{c_j}{w} \right) [r_i (\alpha_i - \alpha_{i-1}) + (\beta_{i-1} - \beta_i)] dt.$$

A similar approach was taken for a two state system in the context of determining virtual circuit holding time policies in IP-over-ATM Networks [22].

Naturally, we would also like to implement this algorithm as efficiently as possible. We have implemented the algorithm for finding the $m - 1$ thresholds in time $O(mn)$, where m is the number of states and n is the number of bins in the histogram. Two important factors which determine the cost (in time expenditure) of implementing our method is the frequency with which the thresholds are updated and the number of bins in the histogram. The frequency with which the thresholds are updated is the subject of one set of experiments which we perform.

Naturally, we would also like to minimize the number of bins used in the histogram. This must be balanced with the fact that the finer grained the histogram, the more accurate our choice of thresholds will be. One important fact that arose in implementing this strategy is that a finer grained binning is more important in some ranges than in others. A way of addressing this problem that was key to the success of the algorithm is to use the thresholds of the optimal strategy (the t_1, \dots, t_{m-1} from above) to guide the choice of bins and their ranges. We choose a constant c number of bins per state. In our case, we chose $c = 5$. The range from t_i to t_{i+1} is divided into c equal sized bins. Figure 2 shows a sample histogram from our experiments.

3 Stochastic Approaches to DPM

In the remainder of this tutorial, we discuss the stochastic version of the DPM problem. The problem basically requires one to devise a strategy (policy) which is *probabilistic*, in the sense that the actions to be taken by the strategy have probabilities attached to them. Unlike deterministic strategies, where a particular state of the system will lead the strategy to take a deterministic action, here, the strategy can choose between multiple actions with pre-designated probabilities.

In recent years, several approaches for designing stochastic DPM strategies have been proposed [30, 7, 6, 11, 35, 37, 36, 41]. These methodologies are based on a stochastic model of the DPM problem, which incorporates the probabilistic characteristics of request arrivals to the device, the device response time distribution, the power consumption by the device in various states and the distribution of energy consumption in changing states. From this stochastic model, an exact optimisation problem is formulated, the solution to which is the required

Bin	Range:		Range Size	Count
	Low End	High End		
1	0	11.2	11.2	35
2	11.2	22.4	11.2	2
3	22.4	33.6	11.2	4
4	33.6	44.8	11.2	7
5	44.8	56	11.2	4
6	56	478.8	422.8	5
7	478.8	901.6	422.8	3
8	901.6	1324.4	422.8	2
9	1324.4	1747.2	422.8	0
10	1747.2	2170	422.8	4
11	2170	4911	2741	7
12	4911	7652	2741	9
13	7652	10393	2741	2
14	10393	13134	2741	5
15	13134	15875	2741	4
16	15875	19050	3175	2
17	19050	22225	3175	3
18	22225	25400	3175	1
19	25400	28575	3175	0
20	28575	∞	∞	1

Fig. 2. A snapshot of the histogram used in OPBA. Note that the offline thresholds are 56, 2179 and 15875 milliseconds. The number of bins per state is 5.

optimal stochastic DPM policy. The strategy devised must ensure that power savings are not achieved at an undue cost in performance. For example, a new request should be always served in a reasonable time. The constructed policy optimizes the *average* energy usage while minimizing *average* delay. The policies are usually validated by simulation to check for the soundness of the modeling assumptions, and the effectiveness of the strategies in practice [35, 30].

The stochastic models which have been used in the literature are discrete-time Markov chains [30, 7], continuous-time Markov Chains [35, 37, 36] or their variants [41]. The approaches vary in the model of time: In the continuous-time case, mode switching commands can be issued at any time, and events can happen at any time. In the discrete-time case, all events and actions occur at certain discrete time points. The continuous-time assumption makes the formulation of the problem easier. In practice, such stochastic modeling seems to work well for specific kinds of applications. Generally, the stochastic matrices for these models are created manually. In [36], stochastic Petri nets are used, which allows automatic generation of the stochastic matrices and formulation of the optimisation problems.

In the following sections, we introduce probabilistic model checking, a formal method for the analysis of probabilistic models, and show how it can be applied to the stochastic DPM approaches described above. We will show that it is a very general technique which can be used to design stochastic DPM strategies in a straightforward manner. Moreover, these techniques based on model checking can prove properties about the designed strategy and can estimate various parameters based on the amount of reliability needed.

4 Probabilistic Model Checking

Model checking is a well established and successful technique for the automatic verification of finite state systems. In recent years, a significant amount of work has gone into *probabilistic model checking*, which allows for verification of systems that exhibit probabilistic behaviour. These include randomised algorithms, which use probabilistic choices or electronic coin flipping, and unreliable or unpredictable processes, such as fault-tolerant systems or communication networks.

To perform *probabilistic model checking* one first constructs a probabilistic model of the system under study. As in the non-probabilistic case, this is usually some kind of labelled transition system which defines the set of all possible states that the system can be in and the transitions which can occur between these states. However, in this case, one must also augment the model with information about the likelihood that each transition will take place.

Properties of the system which are to be verified are then specified, typically in probabilistic extensions of temporal logic. These allow specification of properties such as: “shutdown occurs with probability at most 0.01”; or “the video frame will be delivered within 5ms with probability at least 0.97”. A probabilistic model checker applies algorithmic techniques to analyze the state space of the probabilistic model and determine whether these specifications are satisfied.

Typically, this involves computation of one or more probabilities or performance measures. The operations required are graph-based analysis and solution of linear equation systems or linear optimisation problems.

4.1 Probabilistic Models

Three of the most common models used for probabilistic model checking are DTMCs, CTMCs and MDPs. The simplest of the three is *discrete-time Markov chains* (DTMCs). A DTMC is defined by a set of states S and a probability transition matrix $\mathbf{P} : S \times S \rightarrow [0, 1]$, where $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$. This gives the probability $\mathbf{P}(s, s')$ that a transition will take place from state s to state s' .

Continuous-time Markov chains (CTMCs) extend DTMCs by allowing transitions to occur in real-time, rather than only in discrete steps. A CTMC is defined by a set of states S and a transition rate matrix $\mathbf{R} : S \times S \rightarrow \mathbb{R}_{\geq 0}$. The rate $\mathbf{R}(s, s')$ defines the delay before which a transition between states s and s' is enabled. The delay is sampled from a negative exponential distribution with parameter equal to this rate, i.e. the probability of the transition being enabled within t time units is $1 - e^{-\mathbf{R}(s, s') \cdot t}$. When $\mathbf{R}(s, s') > 0$ for two states s, s' , a *race* occurs and the transition which becomes enabled first is the one taken. Exponentially distributed delays are often suitable for modeling component lifetimes and inter-arrival times. They can also be used to approximately model more complex probability distributions.

Markov decision processes (MDPs) are also a generalization of DTMCs. MDPs can model systems which exhibit both probabilistic and nondeterministic behaviour. An MDP is defined by a set of states S and a function mapping each state $s \in S$ onto a finite, non-empty set of probability distributions over S . Intuitively, from a given state, there are several nondeterministic choices, each of which results in different probabilistic behaviour. One common use of nondeterminism is to model the interleaving that arises from concurrency. This means that MDPs are well suited to modeling the parallel composition of probabilistic processes. Nondeterminism can also be used for underspecification, where the exact probability of some transitions is unknown or unimportant.

4.2 Analysis of Probabilistic Models

Like the conventional, non-probabilistic case, probabilistic model checking usually constitutes verifying whether or not some temporal logic formula is satisfied by a model. The two most common temporal logics for this purpose are PCTL [14, 10] and CSL [3, 5], both extensions of the logic CTL. PCTL is used to specify properties for DTMCs and MDPs and CSL is used for CTMCs.

One common feature of the two logics is the probabilistic \mathcal{P} operator, which allows one to reason about the probability that executions of the system satisfy some property. For example, the formula $\mathcal{P}_{\geq 1}[\heartsuit \textit{terminate}]$ states that with probability 1, the system will eventually terminate. On the other hand, the formula $\mathcal{P}_{\geq 0.95}[\neg \textit{repair } U^{\leq 200} \textit{terminate}]$ asserts that with probability 0.95 or

greater, the system will terminate within 200 time steps and without requiring any repairs. These properties can be seen as analogues of the non-probabilistic case, where a formula would typically state that *all* executions satisfy a particular property, or that *there exists* an execution which satisfies it. CSL also provides the S operator to reason about steady-state (long-run) behaviour. The formula $S_{<0.01}[queue_size = max]$, for example, states that in the long-run, the probability that a queue is full is strictly less than 0.01.

Strictly speaking, probabilistic specifications in PCTL and CSL (such as the examples above) always contain a probability bound so that properties are either true or false for a given system. In practice, however, this can be relaxed. Model checking algorithms for PCTL and CSL typically proceed by computing the actual probability and then comparing it to the bound. Hence, in practice, we can write an expression of the form $\mathcal{P}_{=?}[\diamond terminate]$, for which the model checker will return the actual probability that the system terminates. In many cases, the most useful form of analysis is to compute such values for a range of models or properties. For example, one might determine $\mathcal{P}_{=?}[\diamond^{\leq t} terminate]$ for a range of values of t in order to gain insight into the likelihood of the system terminating as time progresses.

Further properties can be analyzed by introducing the notion of *costs* (or, conversely, *rewards*). If each state of the probabilistic model is assigned a real-valued cost, we can compute properties such as the expected cost to reach a certain states, the expected accumulated cost over some time period, or the expected cost at a particular time instant. As in the previous paragraph, such properties can also be expressed concisely and unambiguously in temporal logic [13, 4].

4.3 PRISM: A Probabilistic Model Checker

PRISM [24, 33] is a probabilistic model checker developed at the University of Birmingham. It supports analysis of the three types of probabilistic models discussed previously: discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs). It verifies properties specified in the temporal logics PCTL (for DTMCs and MDPs) and CSL (for CTMCs). Other probabilistic model checkers include ProbVerus [15] for DTMCs, $E\vdash MC^2$ [16] for CTMCs and DTMCs, and RAPTURE [20] for MDPs.

PRISM has been used to analyze a wide range of case studies, including: probabilistic algorithms for problems such as anonymity, contract signing, leader election and consensus; and performance analysis of various queueing systems, communication networks and manufacturing systems. See [33] for further details. Figure 3 shows a screenshot of the tool running.

Probabilistic models to be analyzed in PRISM are specified in the PRISM language, which is based on the Reactive Modules formalism of Alur and Henzinger [2]. The basic components of this language are *modules* and *variables*. A system is constructed as the parallel composition of a set of modules. A module contains a number of variables which express the state of the module. Its

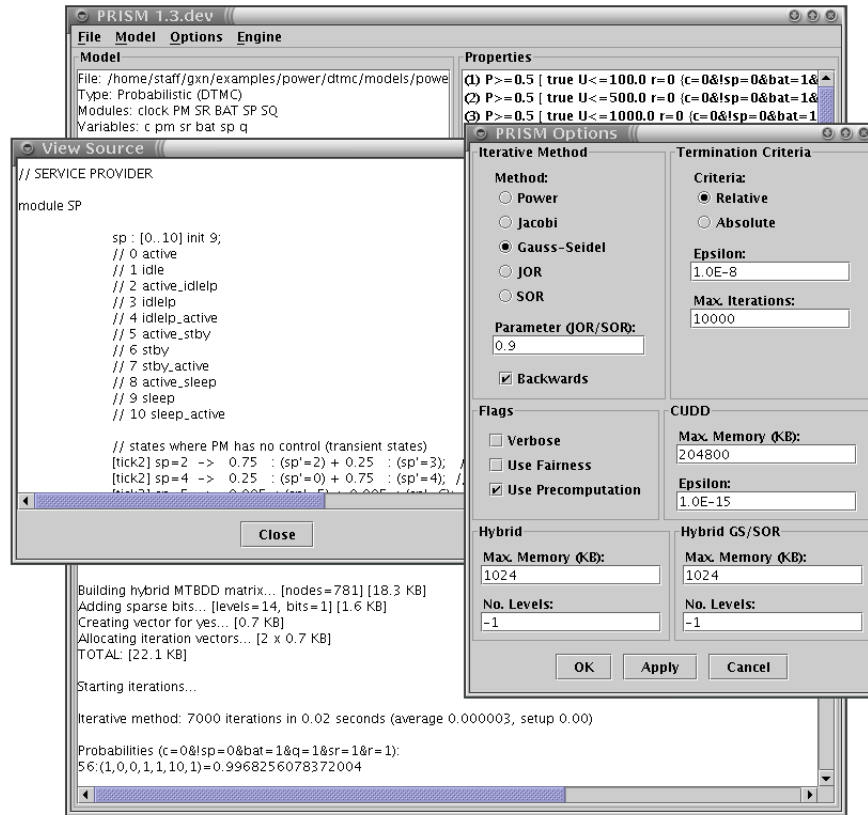


Fig. 3. Screenshot of the PRISM graphical user interface

behaviour is given by a set of guarded commands of the form:

$$[] \langle \text{guard} \rangle \rightarrow \langle \text{command} \rangle;$$

The guard is a predicate over all the variables of the system and the command describes a transition which the module can make if the guard is true. A command is specified by defining the new values of the variables of that module. This means that a module can *read* all of the variables in the system but only *write* to its own local variables. In general, the behaviour of a module is probabilistic, in which case a command takes the form:

$$\langle \text{prob} \rangle : \langle \text{command} \rangle + \dots + \langle \text{prob} \rangle : \langle \text{command} \rangle$$

where $\langle \text{prob} \rangle$ is a probability when the model is a DTMC or MDP and a non-negative, real-value (taken to be the parameter of an exponential distribution)

when it is a CTMC. In addition, the pair of square brackets at the start of a guarded command can contain a label. Actions from different modules with the same label take place synchronously. See [34, 31] for more details.

The overall functionality of the PRISM tool is as follows. First, it reads and parses a model description in the PRISM language. It then constructs the corresponding DTMC, CTMC or MDP, computes the set of all reachable states, and identifies any deadlock states (i.e. reachable states with no outgoing transitions). If required, the transition matrix of the probabilistic model constructed can be exported for use in another tool. Typically, though, PRISM then parses one or more properties in PCTL or CSL and performs model checking, determining whether the model satisfies each property. A prototype version of PRISM has also been developed which supports model checking of cost and reward related properties, as described in the previous section.

5 Probabilistic Model Checking and DPM

In this section, we describe how probabilistic model checking and, in particular, PRISM can be applied to the problem of dynamic power management [28, 29]. More specifically, these techniques are applied to the stochastic DPM approaches of [35, 30]. These schemes are based on constructing a probabilistic model of the dynamic power management system from which, for a given constraint, an optimisation problem is constructed. The solution to this problem is the optimum randomised power management policy satisfying this constraint. In the following sections we describe how PRISM can be used to construct the models for this purpose.

Once an optimal power management policy has been constructed, it must be validated to ensure it performs as intended. Possible approaches are to use trace-based simulation or to actually implement the schemes in device drivers. Since, in the approach described here, the probabilistic models of the system have been explicitly constructed, we can use probabilistic model checking to validate and analyze the policies. This allows a wide range of useful information about the policy to be generated.

5.1 Modelling DPM in PRISM

Probabilistic model checking has been applied to two stochastic DPM approaches: that of Benini et al. [30, 7], based on discrete-time Markov chains, and that of Qiu et al. [35, 37, 36], based on continuous-time Markov chains. This presentation focuses on the former.

The approach is described through the example of [30, 7], an IBM TravelStar VP disk-drive [43]. The device has 5 power states, labelled *sleep*, *stby*, *idle*, *idlelp* and *active*. It is only in the state *active* that the drive can perform data read and write operations. In state *idle*, the disk is spinning while some of the electronic components of the disk drive have been switched off. The state *idlelp* (idle low power) is similar except that it has a lower power dissipation. The states *stby* and

sleep correspond to the disk being spun down. Figures 4 and 5 show actual data for these power states. Figure 4 gives the average power consumption (Watts) and the service time (ms) for each state. Figure 5 shows the average time (ms) to transition between each pair of states.

State	<i>sleep</i>	<i>stby</i>	<i>idlelp</i>	<i>idle</i>	<i>active</i>
Power (W)	0.1	0.3	0.8	1.5	2.5
Service time (ms)	0	0	0	0	1

Fig. 4. Average power consumption (W) and service times (ms) for each power state

	<i>active</i>	<i>idle</i>	<i>idlelp</i>	<i>stby</i>	<i>sleep</i>
<i>active</i>	–	1	5	220	600
<i>idle</i>	1	–	5	220	600
<i>idlelp</i>	5	–	–	220	600
<i>stby</i>	220	–	–	–	600
<i>sleep</i>	600	–	–	–	–

Fig. 5. Average transition times (ms) between power states

We now describe how the system is modeled in the PRISM language. Following the approach of [30, 7], the model constructed is a discrete-time Markov chain (DTMC). Based on the fastest possible transition performed by system, we choose a time resolution of 1ms for the model, i.e. each discrete-time step of the DTMC will correspond to 1ms.

The basic structure of the DPM model can be seen in Figure 6. The model consists of: a Service Provider (SP), which represents the device under power management control; a Service Requester (SR), which issues requests to the device; a Service Request Queue (SRQ), which stores requests that are not serviced immediately; and the Power Manager (PM), which issues commands to the SP, based on observations of the system and a stochastic DPM policy. Each component is represented by an individual PRISM module, which we now consider in turn.

Modelling the power manager (PM). The PM decides to which state the SP should move at each time step. To model this, we split each step into two parts: in the first, the PM (instantaneously) decides what the SP should do next (based on the current state); and in the second, the system makes a transition (with the SP’s move based on the choice made by the PM). To achieve this, we introduce the `CLOCK` module, given below.

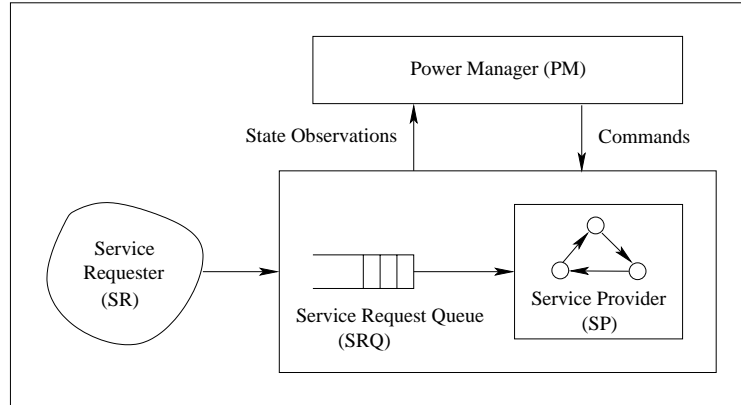


Fig. 6. The System Model

```

module CLOCK

    c : [0..1] init 0;

    [tick1] c = 0 → c' = 1;
    [tick2] c = 1 → c' = 0;

endmodule

```

Transitions of this module are labelled alternately with tick1 and tick2. The PM is then constructed to synchronize with the CLOCK on tick1, while the remaining components are constructed to synchronize with the CLOCK on tick2. A generic PM has the following form:

```

module PM

    pm : [0..4];
    // 0 - go to busy, 1 - go to idle, 2 - go to idlelp
    // 3 - go to standby and 4 - go to sleep

    [tick1] cond1 → p01 : pm' = 0 + p11 : pm' = 1 +
                    p21 : pm' = 2 + p31 : pm' = 1 + p41 : pm' = 4;
    [tick1] cond2 → p02 : pm' = 0 + p12 : pm' = 1 +
                    p22 : pm' = 2 + p32 : pm' = 1 + p42 : pm' = 4;
                    ⋮

endmodule

```

For example, if the state of the system satisfies cond_1 then the PM decides that with probability p_{0_1} the SP will move to *active*, with probability p_{1_1} the SP will move to *idle*, with p_{2_1} to *idlelp*, p_{3_1} to *standby*, and p_{4_1} to *sleep*.

Modelling the service provider (SP). In this example, the SP (the disk drive) has 5 power states. These states and the possible transitions between them are shown in Figure 7. The actual PRISM code is shown below:

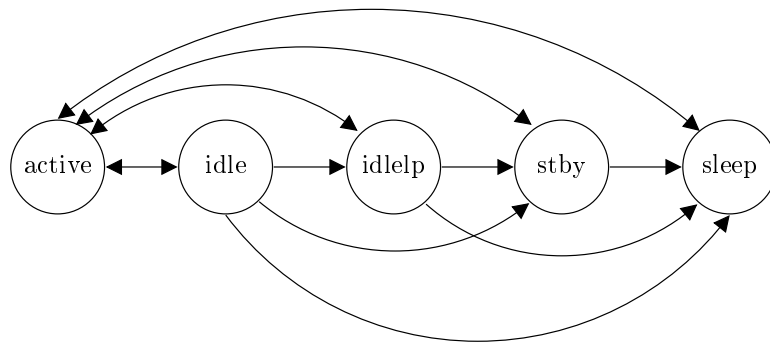


Fig. 7. SP

```

module SP

    sp : [0..10] init 9;
    // 0=active, 1=idle, 2=active_idlelp, 3=idlelp, 4=idlelp_active
    // 5=active_stby, 6=stby, 7=stby_active, 8=active_sleep
    // 9=sleep, 10=sleep_active

    // states where PM has no control (transient states)
    [tick2] sp=2 → 0.75 : (sp'=sp) + 0.25 : (sp'=3);
    [tick2] sp=4 → 0.75 : (sp'=sp) + 0.25 : (sp'=0);
    [tick2] sp=5 → 0.995 : (sp'=sp) + 0.005 : (sp'=6);
    [tick2] sp=7 → 0.995 : (sp'=sp) + 0.005 : (sp'=0);
    [tick2] sp=8 → 0.9983 : (sp'=sp) + 0.0017 : (sp'=9);
    [tick2] sp=10 → 0.9983 : (sp'=sp) + 0.0017 : (sp'=0);
    // PM: goto active
    [tick2] pm=0 ∧ (sp=0 ∨ sp=1) → sp'=0;
    [tick2] pm=0 ∧ sp=3 → sp'=4;
    [tick2] pm=0 ∧ sp=6 → sp'=7;
    [tick2] pm=0 ∧ sp=9 → sp'=10;
    // PM: goto idle
    [tick2] pm=1 ∧ (sp=0 ∨ sp=1) → sp'=1;
    [tick2] pm=1 ∧ (sp=3 ∨ sp=6 ∨ sp=9) → sp'=sp;
    // PM: goto idlelp
    [tick2] pm=2 ∧ (sp=0 ∨ sp=1) → sp'=2;
    [tick2] pm=2 ∧ (sp=3 ∨ sp=6 ∨ sp=9) → sp'=sp;
    // PM: goto stby
    [tick2] pm=3 ∧ (sp=0 ∨ sp=1 ∨ sp=3) → sp'=5;
    [tick2] pm=3 ∧ (sp=6 ∨ sp=9) → sp'=sp;
    // PM: goto sleep
    [tick2] pm=4 ∧ (sp=0 ∨ sp=1 ∨ sp=3 ∨ sp=6) → sp'=8;
    [tick2] pm=4 ∧ sp=9 → sp'=9;

endmodule

```

Recall that the SP synchronizes with the clock on tick2. Hence, all of its guarded commands are labelled with this. Note also that the behaviour of the SP depends on the PM, so the guards reference the variable pm.

Since a time resolution of 1ms has been chosen, in order to correctly model transitions with delays longer than this time resolution *transient* states are introduced. For example, the transient state *active_idlelp* is used to model the non-unitary time transition from *active* to *idlelp*. The transition probabilities in the transient states, taken directly from the data of [30, 7], are chosen such that the mean times to move between power states are as given in Figure 5. Note that we suppose that the power dissipation in these transient states is high (2.5W).

Modelling the service requester (SR) and queue (SRQ). As mentioned above, both the SRQ and the SR will synchronize with the clock on tick2. The

SR has two states: *idle* where no requests are generated and *1req* where one request is generated per time step (1ms). The transitions between these states is based on time-stamped traces of disk access measured on real machines [7]. The module of the SR is given by:

```

module SR

    sr : [0..1] init 0;
    // 0 - idle and 1 - 1req

    [tick2] sr=0 → 0.898 : (sr'=0) + 0.102 : (sr'=1);
    [tick2] sr=1 → 0.454 : (sr'=0) + 0.546 : (sr'=1);

endmodule

```

The SRQ models queue of service requests. It responds to the arrival of requests from the SR and the service of requests by the SP. The queue size will only decrease when the SR and SP are in states *idle* and *active*, respectively. Similarly, it will only increase when the SR is in state *1req* and the SP is not *active*. The PRISM code is as follows:

```

const QMAX = 2; //maximum size of the queue

module SRQ

    q : [0..QMAX] init 0; // size of queue

    // SP is active
    [tick2] sr = 0 ∧ sp = 0 → q' = max(q - 1, 0);
    [tick2] sr = 1 ∧ sp = 0 → q' = q;
    // SP is not active
    [tick2] sr = 0 ∧ sp > 0 → q' = q;
    [tick2] sr = 1 ∧ sp > 0 → q' = min(q + 1, QMAX);

endmodule

```

Modelling a Finite Time Horizon. We suppose that there is a time horizon of one million time steps. To model this horizon, an additional module representing a battery with an expected life span of 1 million time steps is added.

```

module BATTERY

    bat : [0..1] init 1;
    // 0 - battery off and 1 - battery on

    [tick2] bat = 1  $\rightarrow$  0.999999 : (bat'=1) + 0.000001 : (bat'=0);

endmodule

```

Note that, once the battery reaches state 0, it cannot perform the action tick2 which prevents any other modules in the system from performing this action. Hence, it prevents the rest of the system from moving (i.e. the states where $\text{bat} = 0$ act as sink states).

5.2 Policy Construction

Using the PRISM language description detailed in the previous sections, the PRISM model checking tool can be used to construct a generic model of the power management system. From the transition matrix of this system, the linear optimisation problem whose solution is the optimal policy can be formulated, as described in [30, 7]. This optimisation problem is then passed to the MAPLE symbolic solver. Figure 8 shows policies constructed in this way for a range of constraints on the average size of the service request queue. The first column lists the constraint; the second column summarizes the corresponding policy.

5.3 Policy Analysis

Once a policy has been constructed, its performance can be investigated using probabilistic model checking, as described in Section 4. The generic power manager PRISM module is modified to represent a specific policy. Figure 9 shows an example of this for the constraint “queue size is less than 0.05”. This can be seen to correspond to the policy in the 6th row of the table in Figure 8. PRISM is then used to construct and analyze the DTMC for this policy.

The following are some representative results that demonstrate the utility and power of the probabilistic model checking approach. The policies analyzed are those constructed from a range of constraints on the average queue length. In Figure 10, the following properties have been computed: “average power consumption”, “average number of waiting requests” and “average number of lost requests”. Using PRISM, we associate a cost with each state and then compute the expected accumulated cost of the system until it reaches a state where the battery has run out. For example, to determine the average power consumption, the cost associated with each state is determined by the state of the SP and the data given in Figure 4.

From the table, we can see that the average power consumption of a policy decreases as the constraint on queue length used to construct it is relaxed (i.e.

constraint	optimum policy
≤ 2	remain sleeping
≤ 1.5	when the SP is active and the queue is not full: go to idle when the SR is in state 0, the SP is sleeping and the queue is full: remain sleeping with probability 0.99999953 go to active with probability 0.00000047 when the SR is in state 1 and the SP is idle: go to active
≤ 1	when the SP is active and the queue is not full: go to idle when the SR is in state 0, the SP is sleeping and the queue is full: remain sleeping with probability 0.99999850 go to active with probability 0.00000150 when the SR is in state 1 and the SP is idle: go to active
≤ 0.5	when the SP is active and the queue is not full: go to idle when the SR is in state 0, the SP is sleeping and the queue is full: remain sleeping with probability 0.99999418 go to active with probability 0.00000582 when the SR is in state 1 and the SP is idle: go to active
≤ 0.1	when the SP is active and the queue is not full: go to idle when the SR is in state 0 and the SP is idle: remain idle with probability 0.04802800 go to active with probability 0.95197200 when the SR is in state 1 and the SP is idle: go to active when the SP is sleeping: go to active
≤ 0.05	when the SP is active, the SR is in state 0 and the queue is empty: remain active with probability 0.63683933 go to idle with probability 0.36316067 when the SP is idle: go to active when the SP is sleeping: go to active
≤ 0.025	when the SP is active, the SR is in state 0 and the queue is empty: remain active with probability 0.85050171 go to idle with probability 0.14949829 when the SP is idle: go to active when the SP is sleeping: go to active
≤ 0.001	when the SP is active, the SR is in state 0 and the queue is empty: remain active with probability 0.94931282561020 go to idle with probability 0.05068717438980 when the SP is idle: go to active when the SP is sleeping: go to active

Fig. 8. Optimum policies under varying constraints on the average queue size

```

module PM

    // policy when constraint on queue size equals 0.05
    pm : [0..4];
    // 0 - go to busy, 1 - go to idle, 2 - go to idlelp
    // 3 - go to standby and 4 - go to sleep

    [tick1] sr=0 ∧ sp=0 ∧ q=0 → 0.63683933 : pm'=0 //active
                                + 0.36316067 : pm'=1; //idle
    [tick1] sp=1 → pm'=0; //active
    [tick1] sp=9 → pm'=0; //active
    [tick1] ¬(sp=9 ∨ sp=1 ∨ (sr=0 ∧ sp=0 ∧ q=0)) → pm'=pm;

endmodule

```

Fig. 9. Example input to PRISM for a derived Policy under performance constraint = 0.05

the queue size is larger). We can also validate the policy by confirming that the expected size of the queue matches the value in the constraint which was used to construct it. Finally, we see that a side-effect of this is that the average number of requests lost is also increased.

In Figure 11, we show graphical results for a range of policies. Using the same assignments of model states to costs as discussed above, we compute and plot, for a range of values of T : “expected power consumption by time T ”, “expected queue size at time T ”, and “expected number of lost customers by time T ”. The first and third properties are determined by computing expected cost cumulated up until time T ; the second by computing the instantaneous cost at time T . Again, we see that policies which consume less power have larger queue sizes and are more likely to lose requests. Here, though, we can get a much clearer view of how these properties change over time. We see, for example, that the expected queue size at time T initially increases and then decreases. This follows from the fact that the strategies wait for the queue to become full before switching the SP on.

In Figure 12 we plot the probability that a request is served by time T , given that it arrived into a certain position in the queue. Figure 13 shows the probability that N requests get lost by time T for $N = 500$ and $N = 1000$. Again this information has been computed for a range of policies and for a range of values of T . These properties are computed by adding additional state variables to the PRISM model. For those in Figure 13, for example, we add a variable which is initially zero and is increased each time a customer is lost (up to a maximum on N). We then calculate the probability of reaching any state where this variable’s value is equal to N .

policy constraint	average power consumption	average number of waiting requests	average number of lost requests
0.001	2.460629	0.001000	0.000106
0.025	2.393864	0.025000	0.000106
0.05	2.282590	0.050000	0.000106
0.1	2.060040	0.100000	0.000106
0.2	1.670410	0.200000	0.001671
0.25	1.626786	0.250000	0.006720
0.3	1.583163	0.300000	0.011770
0.4	1.495917	0.400000	0.021869
0.5	1.408671	0.500000	0.031968
0.6	1.321424	0.600000	0.042067
0.7	1.234178	0.700000	0.052166
0.8	1.146932	0.800000	0.062265
0.9	1.059686	0.900000	0.072364
1	0.972439	1.000000	0.082463
1.1	0.885193	1.100000	0.092562
1.2	0.797947	1.200000	0.102661
1.25	0.754324	1.250000	0.107710
1.3	0.710700	1.300000	0.112760
1.4	0.623454	1.400000	0.122859
1.5	0.536208	1.500000	0.132958
1.6	0.448962	1.600000	0.143057
1.7	0.361715	1.700000	0.153156
1.75	0.318092	1.750000	0.158206
1.8	0.274469	1.800000	0.163255
1.9	0.187223	1.900000	0.173354
2	0.100000	2.000000	0.183450

Fig. 10. Power versus performance

The graphs show that the probability of requests being lost within a certain time bound increases more quickly for those strategies that consume less power. These results are to be expected since, to reduce power, the strategies must force the service provider to spend more time in low power states which cannot service requests, e.g. *sleep* and *standby*.

5.4 The Continuous-Time Case

Probabilistic model checking has also been applied [28] to the stochastic optimum control approach of [35, 37, 36], which is based on CTMCs rather than DTMCs. The key differences between these two choices of model were mentioned in Section 3. From the point of view of modeling in PRISM, the two are relatively similar. The model has the same basic structure: each component (PM, SP, SRQ and SR) is a separate module and the system is constructed as the parallel composition of these modules. However, in this case we no longer require the

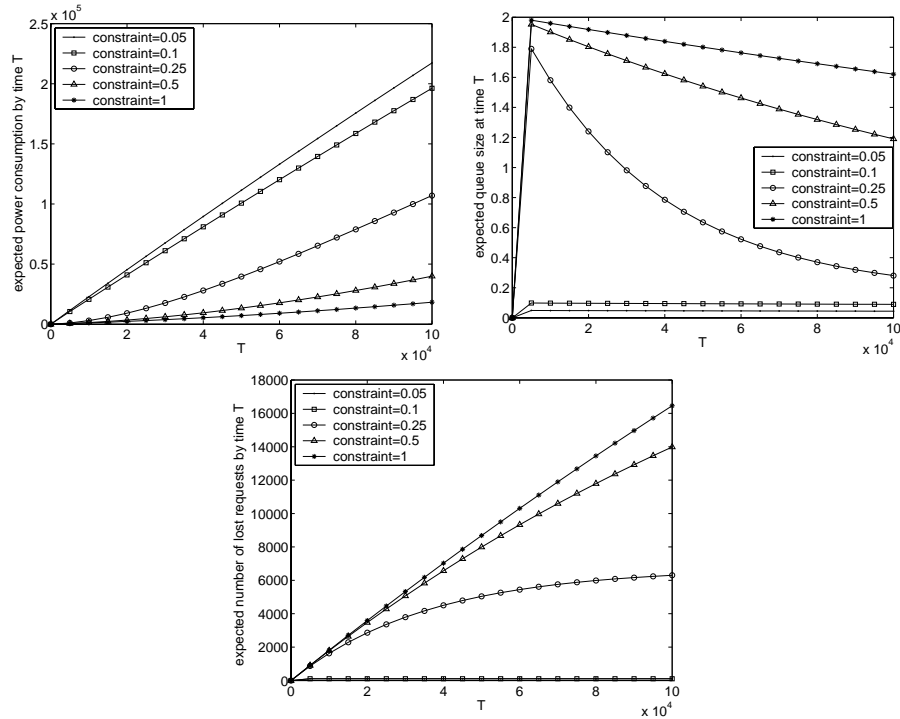


Fig. 11. Power and performance by time T (ms)

CLOCK module to control synchronization. Since the model is a CTMC, components change state according to exponentially distributed delays and the PM acts when such a state transition occurs. The construction of optimum policies from the PRISM model now follows the approach of [35, 37, 36] but is essentially the same overall process.

For analysis of policies, we can consider similar properties to the DTMC case. As in the DTMC case there is a power consumption associated with each power state of the SP (per unit of time); however, there is also a power usage associated with a change in the SP's state. The main differences are that we use the logic CSL as opposed to the logic PCTL, and that the time bound T used in the properties is now a real-value as opposed to a number of discrete steps.

In addition, in this case, using the approach of [23] we can also analyze the policies for alternative inter-arrival distributions, to give a more realistic model of the arrival of service requests. For example, Figure 14 shows the performance (average power consumption, average queue size and average number of lost requests) for optimum policies under five different inter-arrival distributions. The distributions chosen all have the same mean and it can be seen that, with the exception of the Pareto distribution, the long-run performance and costs are reasonably close to those of the exponential arrival process. For the Pareto

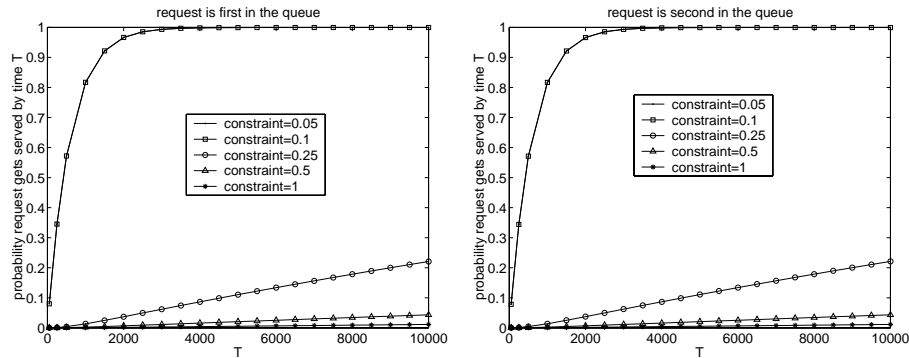


Fig. 12. Probability that a request is served by time T (ms)

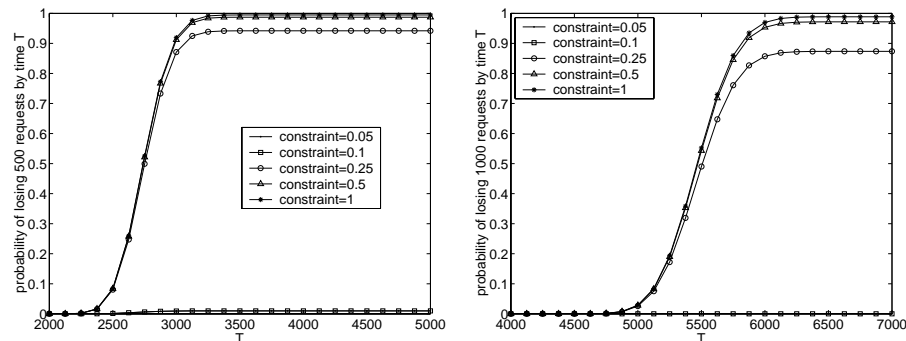


Fig. 13. Probability that N requests gets lost by time T (ms)

distribution, the average queue size is generally much smaller. This is due to the Pareto distribution's *heavy tail*: in the long run, many requests will not arrive for a very long time, in which case the service provider (SP) will serve all pending requests, leaving the queue empty.

6 Conclusions and Future Directions

We have described formal methods for dynamic power management, focusing on two frameworks: (a) on-line algorithms and competitive analysis; and (b) stochastic approaches implemented with probabilistic model checking.

In the first, we have shown our techniques for treating the DPM problem as an on-line problem and presented DPM strategies that can handle ACPI compliant devices with multiple power saving states. We have shown how to learn the probability distribution of the arrival rates of requests online and use that information in the DPM strategy. We also gave analytical bounds on the competitive ratio of these strategies.

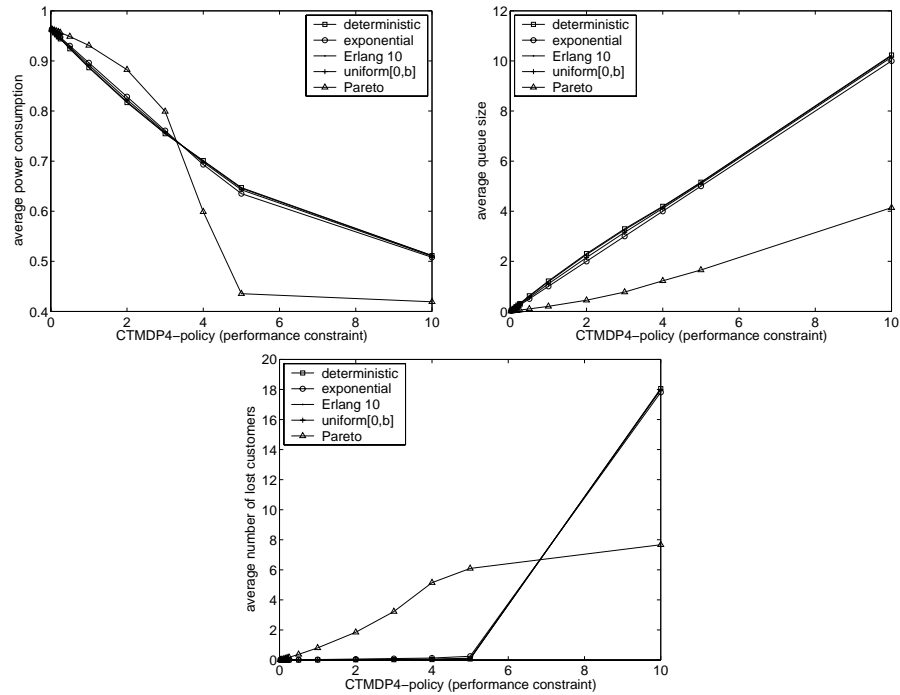


Fig. 14. Analysis of the CTMC case for a variety of inter-arrival distributions

In the second approach, we have shown that probabilistic model checking allows generation of a wide range of performance measures for the analysis of DPM policies. Statistics such as power consumption, service queue length and the number of requests lost can be computed both in the average case and for particular time instances over a given range. The fact that a constructed policy is only known to be optimal in the average case makes this information particularly interesting. Furthermore, the policies' behaviour can be examined under alternative service request inter-arrival distributions such as Erlang and Pareto.

An inherent advantage of the model checking approach is that the analysis of the state-space is exhaustive, in contrast to, say, simulation. This means that the answers computed are guaranteed to be accurate, with respect to the probabilistic model used, and that all behaviour, including corner-case scenarios, is considered. Another advantage is that probabilistic model checking presents a unified framework for automated construction, validation and analysis of DPM policies.

References

- [1] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*.

- Cambridge University Press, 1998.
- [2] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
 - [3] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Proc. Conference on Computer-Aided Verification*, July 1996.
 - [4] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. On the logical characterisation of performability properties. In *Proc. ICALP'00*, 2000.
 - [5] C. Baier, J.-P. Katoen, and H. Hermanns. Approximative symbolic model checking of continuous-time Markov chains. In *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, Eindhoven, August 1999.
 - [6] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (TVLSI) Systems*, 8(3):299–316, 2000.
 - [7] L. Benini, A. Bogliolo, G. Paleologo, and G. D. Micheli. Policy optimization for dynamic power management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):813–833, 1999.
 - [8] L. Benini, G. De Micheli, and E. Macii. Designing Low-power Circuits: Practical Recipes. *IEEE Circuits and Systems Magazine*, 1(1):6–25, Mar. 2001.
 - [9] L. Benini and G. D. Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Publications, 1998.
 - [10] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.
 - [11] E. Y. Chung, L. Benini, A. Bogliolo, and G. D. Micheli. Dynamic Power Management for Non-Stationary Service Requests. In *Proceedings of the Design Automation and Test Europe*, 1999.
 - [12] E.-Y. Chung, L. Benini, and G. D. Micheli. Dynamic Power Management Using Adaptive Learning Trees. In *Proceedings of ICCAD*, 1999.
 - [13] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.
 - [14] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6:512–535, 1994.
 - [15] V. Hartonas-Garmhausen, S. Campos, and E. Clarke. ProbVerus: Probabilistic symbolic model checking. In J.-P. Katoen, editor, *Proc. 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 96–110. Springer, 1999.
 - [16] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov chain model checker. In S. Graf and M. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 347–362, Berlin, 2000. Springer.
 - [17] C.-H. Hwang, C. Allen, and H. Wu. A Predictive System Shutdown Method For Energy Saving of Event-Driven Computation. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pages 28–32, 1996.
 - [18] Intel and Microsoft and Toshiba. Advanced Configuration and Power Interface Specification. Website, December 1996.
 - [19] S. Irani, S. Shukla, and R. Gupta. Dynamic power management of devices with multiple power saving states. *ACM Transactions on Embedded Systems*, accepted for publication, 2003.
 - [20] B. Jeannot, P. D'Argenio, and K. Larsen. Rapture: A tool for verifying markov decision processes. Tools Day, 13th International Conference on Concurrency Theory (CONCUR'02), 2002.

- [21] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Randomized competitive algorithms for non-uniform problems. In *First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 301–309, 1990.
- [22] S. Keshav, C. Lund, S. Phillips, N. Reingold, and H. Saran. An empirical evaluation of virtual circuit holding time policies in ip-over-atm networks. *IEEE Journal on Selected Areas in Communications*, 13:1371–1382, 1995.
- [23] M. Kwiatkowska, G. Norman, and A. Pacheco. Model checking expected time and expected reward formulae with random time bounds. In *Proc. 2nd Euro-Japanese Workshop on Stochastic Risk Modelling for Finance, Insurance, Production and Reliability*, 2002. To appear.
- [24] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.
- [25] Y. Lu, E. Chung, t. Simunic, L. Benini, and G. DeMicheli. Quantitative Comparison of Power Management Algorithms. In *DATE - Proceedings of the Design and Automation and Test in Europe Conference and Exhibition*, 2000.
- [26] Y. Lu and G. DeMicheli. Adaptive Hard Disk Power Management on Personal Computers. In *Proceedings of the Great Lakes Symposium on VLSI*, 1999.
- [27] Microsoft. OnNow Power Management Architecture for Applications. Website, August 1997.
- [28] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Formal analysis and validation of continuous time Markov chain based system level power management strategies. In W. Rosenstiel, editor, *Proc. 7th Annual IEEE International Workshop on High Level Design Validation and Test (HLDVT'02)*, pages 45–50. OmniPress, 2002.
- [29] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Proc. 3rd Workshop on Automated Verification of Critical Systems (AVoCS'03)*, Technical Report DSSE-TR-2003-2, University of Southampton, pages 202–215, April 2003.
- [30] G. A. Paleologo, L. Benini, A. Bogliolo, and G. D. Micheli. Policy Optimization for Dynamic Power Management. In *Proceedings of Design Automation Conference*, 1998.
- [31] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
- [32] S. Phillips and J. Westbrook. *chapter 10 of Algorithms and Theory of Computation Handbook*, chapter On-line algorithms: Competitive analysis and beyond. CRC Press, Boca Raton, 1999.
- [33] PRISM web page. <http://www.cs.bham.ac.uk/~dxp/prism/>.
- [34] PRISM manual. Available from <http://www.cs.bham.ac.uk/~dxp/prism/>.
- [35] Q. Qiu and M. Pedram. Dynamic Power Management Based on Continuous-Time Markov Decision Processes. In *Proceedings of Design Automation Conference*, pages 555–561, June 1999.
- [36] Q. Qiu and Q. Wu and M. Pedram. Dynamic power management of complex systems using generalized stochastic petri nets. In *Proceedings of Design Automation Conference*, pages 352–356, June 2000.
- [37] Q. Qiu, Q. Wu and M. Pedram. Stochastic Modeling of a Power-Managed System: Construction and Optimization. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 1999.

- [38] D. Ramanathan, S. Irani, , and R. K. Gupta. Latency Effects of System Level Power Management Algorithms. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, 2000.
- [39] S. Irani and S. Shukla and R. Gupta. Competitive analysis of dynamic power management strategies for systems with multiple power saving states. In *Proceedings of the Design Automation and Test Europe Conference*, 2002.
- [40] S. Shukla and R. Gupta. A Model Checking Approach to Evaluating System Level Power Management for Embedded Systems. In *Proceedings of IEEE Workshop on High Level Design Validation and Test (HLDVT01)*. IEEE Press, November 2001.
- [41] T. Simunic, L. Benini, and G. D. Micheli. Event Driven Power Management of Portable Systems. In *In the Proceedings of International Symposium on System Synthesis*, pages 18–23, 1999.
- [42] M. B. Srivastava, A. P. Chandrakasan, and R. W. Broderon. Predictive Shutdown and Other Architectural Techniques for Energy Efficient Programmable Computation. *IEEE Trans. on VLSI Systems*, 4(1):42–54, march 1996.
- [43] *Technical specifications of hard drive IBM Travelstar VP 2.5inch*, available at. <http://www.storage.ibm.com/storage/oem/data/travvp.htm>, 1996.