

On Temperature-Aware Scheduling for Single-Processor Systems

Deepak Rajan and Philip S. Yu

IBM T. J. Watson Research Center, Hawthorne, NY 10532, USA
{drajan,psyu}@us.ibm.com

Abstract. Power-aware operating systems/processor controllers ensure that the system temperature does not exceed a threshold by utilizing system-throttling, where the clock speed is scaled to an equilibrium load. We denote this as the Constant policy, and compare against Zig-Zag policies that alternate between phases of cooling and heating. In this paper, we characterize and calculate the best possible Zig-Zag policy, and argue that simple system-throttling rules are often optimal.

In reality, however, the system design often forces us to implement Zig-Zag policies. In particular, we consider the case where the processor can operate only at a few discrete states; thus it is required to alternate between cooling and heating phases. In such a setting, we develop an algorithm that outperforms all other Zig-Zag policies, and present computational experiments emphasizing the performance of our algorithm.

1 Introduction

Energy and temperature management of processor systems is an increasingly important problem as their power consumption rises drastically with every new generation, while the rate of technological improvements in cooling systems has not been keeping pace [1]. Naturally, this has resulted in a large body of work attempting to incorporate energy and temperature considerations into processor scheduling levels. This is now implementable at the operating system/program level since most modern day processors have interfaces that allow the user to control its speed in real-time using a mechanism called dynamic voltage scaling (DVS) [2]. For a detailed investigation into DVS and other mechanisms for implementing dynamic thermal management, see [3,4].

Processors usually ensure that the system temperature does not exceed a maximum amount by system-throttling. In [5], we showed that such simple system-throttling policies (represented by the class of Constant policies) are optimal under certain simplifying assumptions. In this paper, we are interested in developing the optimal Zig-Zag policy, for two main reasons. Firstly, such an analysis allows us to determine the exact conditions when a carefully constructed Zig-Zag policy can outperform the best Constant policy. Secondly, in most real systems, it is not possible to implement simple system-throttling using a Constant policy; one is forced to Zig-Zag because of the constraints of the system. In particular, this is true of current implementations of DVS; the processor can operate only

in a small discrete set of speeds [4]. Under such settings, an optimal Zig-Zag policy can provide significant benefits over a naive implementation. Note that all operating policies that do not maintain a constant processor speed belong to the class of Zig-Zag policies.

We begin in Section 2 by introducing the thermodynamics models. The main contributions of this paper are presented in Section 3, where we carefully deconstruct many simplifying assumptions in an attempt to analyze the effectiveness of system-throttling, and in Section 4, where we characterize the optimal Zig-Zag policy and compare it against Constant. In Section 5, we develop an efficient speed-scaling algorithm that implements our results. To illustrate our approach, we consider a practical setting where the Constant policy can not be implemented, and present the results of computational experiments in Section 6. Finally, in Section 7, we summarize the contributions of this work.

1.1 Related Work

Many other researches have looked at power management of processors. This list is by no means exhaustive, but illustrates some settings where speed-scaling has proved to be effective. In many scenarios, the tasks have different processor, memory and I/O requirements. Thus, it is possible to mix and match tasks to reduce net system power utilization [6,7]. In multi-processor systems, if it is possible to move jobs among the different processors, then a scheduling algorithm can outperform system-throttling by moving jobs between hot and cold processors [8,9]. In some settings, the net energy available is limited; studied both in a theoretical setting [10,11,12,13], and in a practical setting [14,15,16]. In other studies, the authors present a variety of scheduling and workload management strategies to develop temperature-aware computing centers [17,18,19].

2 Problem Setting

We first present the heat model for estimating the temperature of the system. We formalize system-throttling using the Constant policy, which keeps the workload constant such that the temperature threshold is not violated. We formalize all other scheduling policies using the notion of a Zig-Zag policy, since it must have alternate periods of cooling and heating. We consider a single processor system, which at speed ℓ can complete w units of work in time w/ℓ .

We assume that the system is cooled using Newton's law; $dT = -\rho T$, where dT is the instantaneous rate of change of temperature, and ρ is a positive constant [20]. To model the heat gain due to the processor, we assume that $dT = \beta P$, where P is the power dissipated and β is a positive constant [13,21]. We model the power dissipated by the processor as $P = \ell^\alpha$, where α is strictly larger than 1, and ℓ is the speed of the processor [10,22]. Combining these effects,

$$dT = \beta\ell^\alpha - \rho T. \quad (1)$$

This is an ordinary differential equation that can be easily solved [23] for constant speed. Let us define $\tau(\ell) = \beta\ell^\alpha/\rho$. We mention that the system cools/heats

up exponentially until it reaches a stable temperature. This temperature is unique for speed ℓ and is given by $\tau(\ell)$. Similarly, we also define the speed that stabilizes the system at temperature T as $\ell(T)$. Let ℓ_0 be the speed at which the system operates at the maximum system design temperature T_{max} ; then, $\ell_0 = \ell(T_{max})$. One way to ensure that the system never exceeds T_{max} is:

1. If the temperature hits T_{max} , the system enforces throttling (speed = ℓ_0).
2. If the temperature is below T_{max} , the system increases the speed to 1.

In the absence of idling, the system operates at speed 1 initially, followed by speed ℓ_0 . System-throttling can be represented by a policy that keeps the temperature constant (denoted as Constant); see dashed-and-dotted line in Figure 1. In general, the scheduler can decide to operate at any intermediate speed ($\ell_0 \leq \ell \leq 1$) depending on the state of the system. Furthermore, the scheduler may choose to operate at any speed $0 \leq \ell \leq \ell_0$ so as to increase the rate of cooling. We describe such alternate (but quite general) policies by the Zig-Zag policy, and characterize it as follows:

- The operating temperature range is $[T_m, T_{max}]$.
- The operating speeds are ℓ_b (cooling) and ℓ_a (heating), where $\ell_b < \ell_0 < \ell_a$.

We illustrate this policy in Figure 1 using solid lines. Note that a Constant policy maintaining temperature T_0 operates at a speed between the cooling and heating speeds of a Zig-Zag policy operating between temperatures T_m and T_0 .

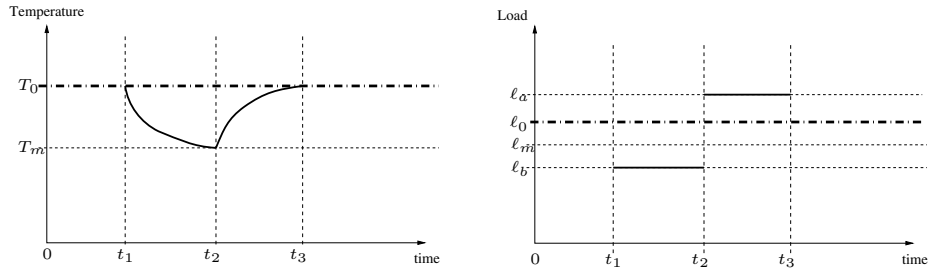


Fig. 1. Zig-Zag and Constant policies

3 Analysis of System-Throttling

In previous work [5], we showed that system-throttling is optimal for single-processor systems when the goal is to maximize the amount of work done, and this work is assumed to be a single task. We state the result here as Theorem 1. In Sections 3 and 4, we consider some of the inherent assumptions in this result. For the rest of this paper, we assume without loss of generality that the Constant policy operates at speed $\ell_0 = \ell(T_{max})$; thus maximizing the amount of work done among all possible policies that maintain a constant temperature.

Theorem 1. *The Constant policy does more work than any Zig-Zag policy.*

3.1 Multiple Tasks

All realistic scenarios involve a variety of tasks, each with its own processing requirements and importance. Let there be n tasks, each with work w_i . At speed ℓ , the time taken to complete task i is therefore w_i/ℓ . These tasks may have different levels of importance; we measure this by associating a weight γ_i with task i . Let the system begin at time 0. The jobs may arrive at different times; i.e., job i has release date r_i . Initially, we assume that all jobs arrive at time 0.

Given a schedule, we can calculate the completion time for each task in the system. We limit our attention to all objective functions that are non-decreasing functions of the completion times (“natural” objective functions). All commonly used metrics (make-span, weighted completion times, weighted flow time, maximum flow time, etc.) satisfy this property. In the absence of pre-emption, it is easy to show that any natural objective function is minimized by Constant.

We show that the Constant policy dominates any Zig-Zag policy even when pre-emption is allowed so long as we minimize a natural objective function; we state it as Theorem 2. We show that for any sequence of tasks scheduled using a Zig-Zag policy, there exists a “similar” Constant policy that dominates it. We say that two policies are similar if for all pre-empted jobs, the same fraction of its work is completed before it is pre-empted. In Figure 2, job 1 is pre-empted by job 2, and virtual job $\bar{1}$ denotes the part of job 1 completed at pre-emption.

Theorem 2. *The Constant policy minimizes any natural objective function.*

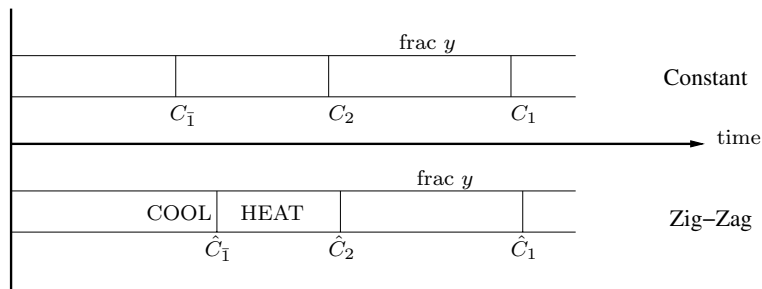


Fig. 2. Completion time: Pre-emption

3.2 Jobs with Arbitrary Release Dates

Earlier, we assumed that the Constant policy does not include any idle time; i.e., the processor would always have available work. In the absence of release dates, this is a natural assumption to make. Consider the Zig-Zag policy in Figure 2. If the release date of job 2 is the time at which it pre-empted job 1, then there exists no similar Constant policy (with fraction $1 - y$ of job 1 completed before pre-emption) that has no idle time. By scaling the speed prior to a job arrival, one may be able to respond to it better, thus minimizing a variety of natural objectives. Operating at temperature T_0 , if job 1 will be pre-empted by job 2 when it arrives at time t_2 (see Figure 1), there are three main decisions:

1. When to switch to a Zig-Zag policy? (Best choice of time t_1 ?)
2. How much to scale during cooling? (Best choice of speed ℓ_b ?)
3. How much to scale during heating? (Best choice of speed ℓ_a ?)

4 Generating Optimal Zig-Zag Policies

In this section, we address all the issues discussed above. We present the best Zig-Zag algorithm that minimizes any natural objective function. To develop the best Zig-Zag policy, one needs to understand the trade-offs involved in alternating between heating and cooling phases. We need to characterize (see Figure 1):

- the loss in work in the cooling phase (from time t_1 to time t_2) at speed ℓ_b , to cool from temperature T_0 to temperature T_m , and
- the gain in work in the heating phase (from time t_2 to time t_3) at speed ℓ_a , to heat from temperature T_m back to temperature T_0 .

We show that this trade-off is related to the function \mathcal{F} ; formalizing the result as Proposition 1. To illustrate, we plot the function $\mathcal{F}(x, 0.75, 0.5, 3)$ in Figure 3.

Proposition 1. *Consider a Zig-Zag policy that cools at speed ℓ_b until temperature T_m is reached and heats at speed ℓ_a until temperature T_0 is reached. Compared to the Constant policy, this Zig-Zag policy loses work in cooling equal to $\mathcal{F}(\ell_b, \ell_0, \ell_m, \alpha)$ and gains work in heating equal to $\mathcal{F}(\ell_a, \ell_0, \ell_m, \alpha)$; where*

$$\mathcal{F}(x, p, q, \alpha) = (x - p) \log \frac{x^\alpha - q^\alpha}{x^\alpha - p^\alpha} \tag{2}$$

Lemma 1 characterizes key properties of \mathcal{F} as a function of x (p, q, α kept constant). This result is a straightforward application of calculus, and will be used to derive the best Zig-Zag policy.

Lemma 1. *Given p, q, α such that $p > q$ and $\alpha > 1$, then for $x > 0$*

1. $\mathcal{F}(x)$ has a maximizer \bar{x} for $x > p$, and a minimizer \hat{x} for $x < q$.
2. Furthermore, $\mathcal{F}(x_1) > \mathcal{F}(x_2)$, for all $x_1 < q$ and $x_2 > p$.
3. For $x > p$, \bar{x} is the only local maximum of function \mathcal{F} , denoted by $\overline{\mathcal{F}}(p, q, \alpha)$
4. For $x < q$, \hat{x} is the only local minimum of function \mathcal{F} , denoted by $\underline{\mathcal{F}}(p, q, \alpha)$

From Proposition 1 and Lemma 1.2, we can derive an alternate proof for Theorem 1. From Lemma 1.3 and 1.4, we see that the slope of \mathcal{F} changes monotonically. As a consequence of Lemma 2, it is computationally easy to implement scheduling decisions that optimize \mathcal{F} .

Lemma 2. *Given $\alpha > 1 > p > q$, an ϵ -approximate maximizer (minimizer) of \mathcal{F} for $x > p$ ($0 < x < q$) can be calculated in $\log(\epsilon)$ time using bisection search.*

In [5], we showed that if the current system temperature is $T < T_{max}$, then the optimal speed (for increasing the temperature to T_{max}) can be calculated exactly.

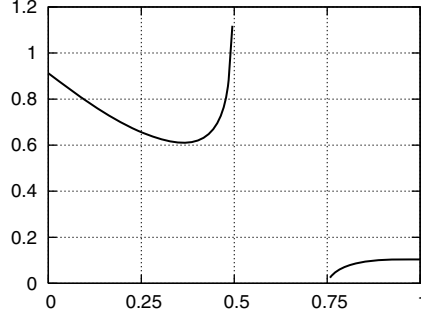


Fig. 3. Function $\mathcal{F}(x, 0.75, 0.5, 3)$

We proved that the amount of work done is maximized for a particular choice of $\ell = \bar{\ell}$, and showed how to calculate this value. Here, we derive a more general result in Theorem 3 that characterizes the optimal speed (which maximizes work done) for both cooling and heating phases of a Zig-Zag policy.

Theorem 3. *Let the current temperature be T_1 . Assume that we operate the system at a speed ℓ such that the system reaches temperature T_2 in no more than t_0 time units, and then operate the system such that the temperature is maintained at T_2 . Let ℓ_{exact} be the speed that reaches temperature T_2 from temperature T_1 in exactly time t_0 . Let the net work done (until time infinity) be $W(\ell)$, and $\ell_i = \ell(T_i)$ for $i = 1, 2$.*

- $T_2 > T_1$ (heating): *Let $\bar{\ell}$ maximize $\mathcal{F}(\ell, \ell_2, \ell_1, \alpha)$ for $\ell > \ell_2$. Then, $W(\ell)$ has a unique maximizer ℓ^* , where $\ell^* = \min\{1, \max\{\ell_{exact}, \bar{\ell}\}\}$.*
- $T_2 < T_1$ (cooling): *Let $\hat{\ell}$ minimize $\mathcal{F}(\ell, \ell_1, \ell_2, \alpha)$ for $\ell < \ell_2$. Then, $W(\ell)$ has a unique maximizer ℓ^* , where $\ell^* = \min\{\ell_{exact}, \hat{\ell}\}$.*

4.1 Main Results

In Section 3.2, we argued that a carefully constructed Zig-Zag policy may outperform Constant if the Zig-Zag is carried out in conjunction with a pre-emption. If job 1 is to be pre-empted by job 2 at time t_2 , Zig-Zag can outperform Constant if we reduce the speed (cooling) prior to time t_2 , and increase the speed (heating) after time t_2 . This allows us to trade-off the processing of the less important job (job 1) for the faster processing of the more important job (job 2). Analyzing the trade-offs involved, we develop and characterize the optimal Zig-Zag policy. This result is very important, for two reasons. Firstly, by comparing against the Constant policy, we describe the exact conditions for which Zig-Zag can dominate Constant. Secondly, when the Constant policy can not be implemented, we develop optimal speed-scaling algorithms.

To compare against Constant, we consider one iteration of a Zig-Zag policy that drops the temperature to T_m , and back up to T_0 ; see Figure 1. Let $\ell_0 = \ell(T_0)$ and $\ell_m = \ell(T_m)$. Let the speed during the cooling (heating) phase be ℓ_b (ℓ_a); we

have $\ell_a > \ell_0 > \ell_m > \ell_b$. Proposition 2 characterizes the difference in completion times (of jobs 1 and 2) between the Constant policy and the best Zig-Zag policy.

Proposition 2. *Compared to Constant, the best Zig-Zag policy increases completion time of job 1 by $(\underline{\mathcal{F}}(\ell_0, \ell_m, \alpha) - \overline{\mathcal{F}}(\ell_0, \ell_m, \alpha))/\ell_0$ and decreases completion time of job 2 by $\overline{\mathcal{F}}(\ell_0, \ell_m, \alpha)/\ell_0$.*

Since job 1 is pre-empted and gets completed after job 2, the increase in completion time (due to cooling) of job 1 is partially offset by the earlier completion of job 2. This explains the two terms in the expression for the completion time of job 1. Using Proposition 2, we can characterize exactly when it is advantageous to Zig-Zag in conjunction with a pre-emption.

Observe that the completion time of all jobs completed after job 2 increases (not just job 1). More precisely, the completion time of all jobs processed until the next idle time increases as much as job 1. We refer to all such jobs by L . If the gains in completion time (of job 2) offset the losses in completion time (of jobs in L), then one should Zig-Zag in anticipation of the pre-emption of job 1. This analysis naturally depends on the objective function.

Make-span and Maximum Response Time: Observe that the order in which the jobs are processed does not affect the make-span. Since pre-emption is not necessary, there are no performance improvements (over Constant) that can be achieved by a Zig-Zag policy. In fact, this is true for any natural objective function where the optimal policy does not involve any pre-emption. For instance, for minimizing the Maximum Response Time, it has been shown that scheduling in increasing order of r_i (release dates) is the optimal policy. We state the general result as the following theorem.

Theorem 4. *For any objective function where the optimal schedule (sequence in which jobs are processed) does not involve pre-emption, the Constant policy dominates any Zig-Zag policy.*

Weighted Response time: Here, we consider the minimization of weighted response time of tasks. Since the response time of a job is the difference between its completion time and its release date, any algorithm that minimizes the weighted response time also minimizes the weighted completion time.

It is well known that the weighted response time is optimized by scheduling the jobs in increasing order of their remaining processing time, scaled by their weights. Thus, at any time, all available jobs are sorted in increasing order of w_i/γ_i and then processed in that sequence. Since γ_i/w_i can be interpreted as the density of job i (value of unit work), this algorithm is also referred to as HDF (Highest Density First). Implementing HDF, the optimal schedule often includes pre-emption. Suppose that job 1 is being processed (with remaining work \hat{w}_1) when job 2 is released. If $w_2/\gamma_2 < \hat{w}_1/\gamma_1$, then job 2 pre-empts job 1. The following theorem characterizes whether we should Zig-Zag in anticipation of a pre-emption, and is the main theoretical result in this work.

Theorem 5. *The best Zig-Zag policy should Zig-Zag in anticipation of the pre-emption of job 1 by job 2 only if there exists a speed $\ell_m < \ell_0$ for which the following condition holds: $\delta > 0$, where*

$$\delta = \overline{\mathcal{F}}(\ell_0, \ell_m, \alpha)(\gamma_2 - \sum_{i \in L} \gamma_i) - \underline{\mathcal{F}}(\ell_0, \ell_m, \alpha)(\sum_{i \in L} \gamma_i). \quad (3)$$

If the condition is satisfied, we choose ℓ_m as the speed that maximizes δ , $\tau(\ell_m)$ as the temperature to which the processor should cool down prior to pre-emption; the cooling and heating speeds are chosen as in Theorem 3.

From Lemma 1.2 and Theorem 5, we can prove that δ is no greater than 0 when all weights are 1. This proves the following result for the case of homogeneous tasks, and for the minimization of a large class of natural objective functions, including the special case of average response time.

Theorem 6. *If the tasks are of equal importance (weights $\gamma_i = 1$, $\forall i$), then the Constant policy dominates any Zig-Zag policy.*

Even though the condition in Theorem 5 may be satisfied for arbitrary weights, it is more likely that it is not. The decrease in completion time of job 2 may offset the increase in completion time of job 1, but this potential advantage is often nullified by the fact that all jobs in L complete later in the Zig-Zag policy. As a result, in practice, it does not pay to Zig-Zag unless there are idle times in the system. In the presence of idle times, L is often small, and the condition in Theorem 5 may be satisfied. Nevertheless, Theorem 5 is an exact characterization of the conditions when a Zig-Zag policy outperforms a Constant policy, and is a significant result.

5 Forced Zig-Zag

In the previous section, we showed that, in most cases, the Constant policy dominates the best Zig-Zag policy. However, our analysis of the best Zig-Zag policy is quite useful when one is forced to Zig-Zag. This often happens in practice since a processor can operate only in one of a small discrete set of operating speeds. Let this set of feasible speeds be S . If the processor can scale the speed continuously, then it could alternate between increasingly small cooling and heating phases, staying as close to ℓ_0 as possible. In practice, there is usually a cost associated with changing the speed, and can (equivalently) be modeled using a time threshold Δ between successive speed changes. We denote the largest permissible speed less than ℓ_0 as ℓ_{lb} . Formally, $\ell_{lb} = \max\{\ell \in S : \ell \leq \ell_0\}$. System-throttling can now be formalized as consisting of:

- Heating phase. Every Δ time units, set the speed to 1 (full speed).
- Cooling phase. If temperature hits T_{max} , the processor operates at speed ℓ_{lb} .

We refer to this scheduling policy as Naive. To illustrate the preceding analysis in a computational setting, we compare the Naive algorithm against the best

possible Zig-Zag algorithm, which tries to minimize the work lost during the cooling phase, and maximize the work done in the heating phase. Since we are operating in a setting where a Constant policy dominates any Zig-Zag policy, there is no incentive to start a cooling phase unless forced to do so. Hence, the best Zig-Zag policy differs from Naive only in the choice of the speed during heating. To summarize, our algorithm, which we call BestZig, operates as follows.

- Heating phase. Every Δ time units, let T be the current temperature. The processor sets the speed to ℓ_{opt} , where ℓ_{opt} maximizes the work done in the heating phase; calculated as:

$$\ell_{opt} = \arg \max_{x \geq \ell_0 | x \in S} \mathcal{F}(x, \ell_0, \ell(T), \alpha) \quad (4)$$

- Cooling phase. If temperature hits T_{max} , the processor operates at speed ℓ_{lb} .

Observe that the choice of ℓ_{opt} follows directly from Theorem 3. Moreover, the computational effort involved in calculating ℓ_{opt} is trivial (can be done in $\mathcal{O}(|S|)$ time by evaluating \mathcal{F} for all feasible values in S). To illustrate both algorithms, we present them pictorially in Figure 4. The algorithms BestZig and Naive differ in the choice of ℓ_{heat} , while both choose $\ell_{cool} = \ell_{lb}$. For BestZig, $\ell_{heat} = \ell_{opt}$, and for Naive, $\ell_{heat} = 1$.

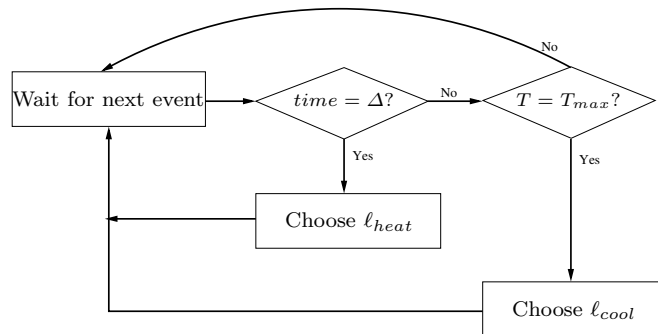


Fig. 4. Operation of system

6 Computations

Now, we present extensive computational results illustrating the performance of BestZig. The data is taken from the STG (Standard Task Graph) dataset, available at <http://www.kasahara.elec.waseda.ac.jp/schedule/index.html>. These include problems with 50, 100, 300, 500, 750, and 1000 tasks. We refer to the number of jobs in the instance as the size of the problem. There are 180 random instances for each size. We assume that the processor can operate only at speeds $\{0, G, 2G, \dots, 1\}$, and can change the speed every Δ units of time. For the base runs of BestZig, we choose $G = 0.1$ and $\Delta = 2$. For the heat

model, we choose $\rho = 0.1$ and $\beta = 12$, resulting in the maximum equilibrium speed $\ell_0 = 0.69$. Initially, we assume that all jobs are available at time 0; i.e., $r_j = 0, \forall j$.

6.1 Make-Span Minimization

For minimization of make-span on a single processor system, the sequence in which the jobs are processed is irrelevant; pre-emption is not necessary in this setting. To compare Naive and BestZig, we contrast against Constant, the theoretical best that any Zig-Zag policy can achieve. In Figure 5; we plot the percentage deviation in make-span when compared to the Constant policy in the y -axis. We present the average over all instances for a particular run; our experiments indicated that the standard deviation across instances was insignificant. We see that BestZig outperforms Naive by about 50%, and that this improvement in performance is robust to changes in data.

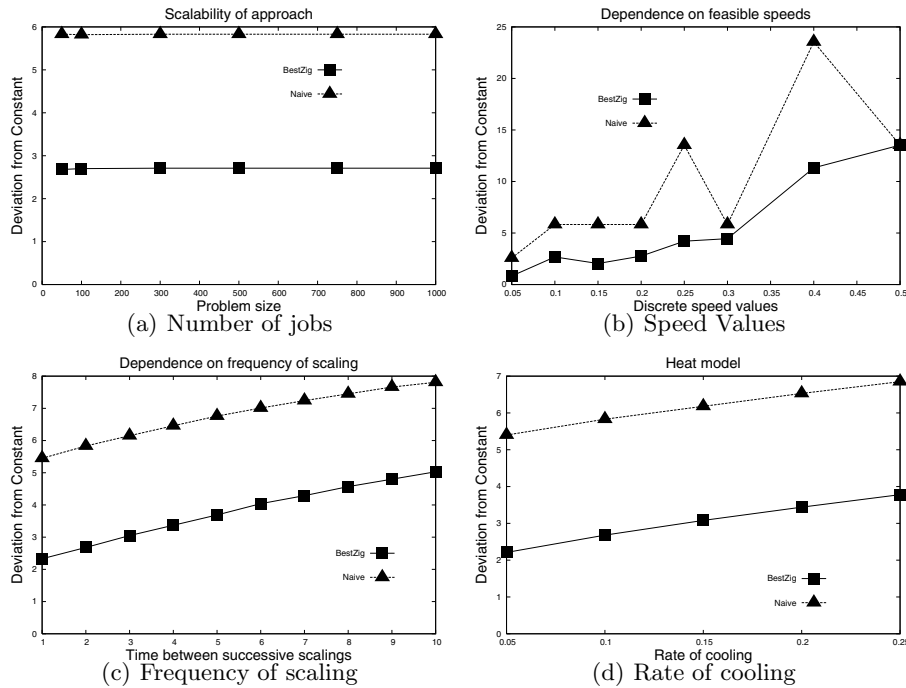


Fig. 5. Summary of computational experiments

First, we present computations which illustrate the effectiveness of our algorithm, and its robustness with respect to the number of jobs. We present a summary of the results in Figure 5(a). On the x -axis, we present the number of jobs in the instance. As we can see, both algorithms are highly insensitive to problem size. However, BestZig outperforms Naive by 50% in all the instances.

We also performed a series of experiments to understand the sensitivity of the performance of BestZig to changes in other data; we present the results of problems of size 50 (the behavior is similar for the other instances). First, we analyze the performance of the algorithm as a function of the possible operating states of the processor. This is characterized by the parameter G ; the processor operates only at speeds $\{0, G, 2G, \dots, 1\}$. In Figure 5(b), we plot G on the x -axis. We see that as G increases, the performance of both Naive and BestZig deteriorates, since larger G implies that the processor can operate at fewer number of feasible speeds. Nevertheless, the performance of BestZig is often 50% better than Naive, and is much more robust. In fact, only when the number of operating speeds is small ($G \geq 0.4$) does BestZig perform more than 5% worse than Constant. On the other hand, the performance of Naive is very sensitive to the value of G .

We also analyze the performance of the algorithm as a function of the time between successive speed changes (Δ). In Figure 5(c), we plot Δ on the x -axis. As Δ increases, the performance of both algorithms deteriorates, since increasing Δ decreases the ability of any Zig-Zag algorithm to mimic a Constant policy. Interestingly, both algorithms are robust to increases in Δ ; a ten-fold increase in Δ (from 1 to 10 seconds) worsens the performance of the algorithms only by a few percentages. As before, BestZig outperforms Naive by about 50%.

Finally, we analyze the dependence of the algorithms on parameters of the heat model. In Figure 5(d), we plot ρ on the x -axis; larger ρ implies better cooling. Since any Zig-Zag policy is dominated by Constant in our setting, faster cooling/heating (larger ρ) only worsens the performance of a Zig-Zag policy. If it was advantageous to Zig-Zag, BestZig would have performed better with increasing ρ . As before, BestZig continues to perform 50% better than Naive.

6.2 Response Time Minimization

To demonstrate the performance of BestZig (when compared to Naive and Constant) for other objective functions, we consider the minimization of the average response time of the tasks. In this scenario, we proved that Constant still dominates any Zig-Zag policy (see Theorem 6). Since we are interested in minimizing flow time, for each job, we choose its release date r_j randomly from 0 to $M \times \kappa$, where M is the sum of all the processing times of the jobs, and κ is a constant. By varying κ , we can control the spread of the release times of the jobs. κ can also be thought of as an indicator of the amount of idle time in the system.

The results of this experiment are presented in Figure 6(a). In the x -axis, we plot the value of κ . We see that the performance of the Zig-Zag policies depends quite heavily on the value of κ ; however, BestZig consistently outperforms Naive. Furthermore, its performance is even stronger in the cases where Naive performs poorly; BestZig performs 2% to 8% from Constant, whereas Naive is more than 20% off when $\kappa = 1$. The shape of the curve can be explained as follows. When $\kappa = 0$, all jobs are released at time 0, and there is no idle time in the system. When κ is large (≥ 2), there is a large amount of idle time in the system. In both scenarios, there is no/minimal pre-emption involved in the processing. As a result, maximizing the amount of work is a reasonable surrogate for minimizing

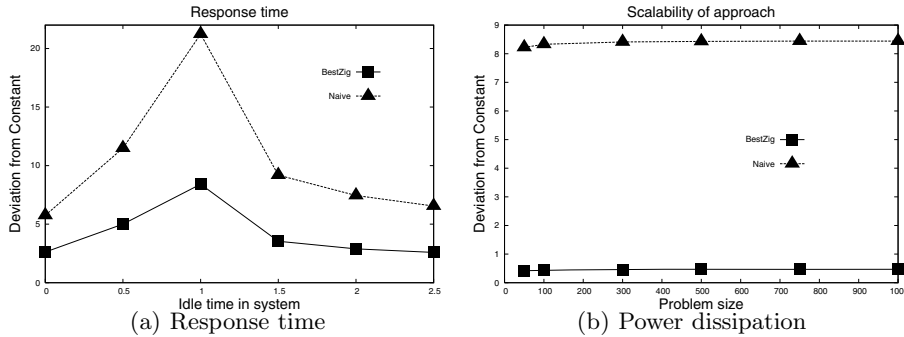


Fig. 6. Other objective functions

response time, and thus the Zig-Zag policies perform similar to the case of make-span minimization (BestZig 3% and Naive 6% from optimal). For intermediate κ , there is much more pre-emption, and thus both algorithms perform worse (compared to Constant) since they are forced to Zig-Zag.

6.3 Power Minimization

In our analysis (and in the algorithm BestZig), we do not optimize the power dissipated by the processor. However, we track the amount of heat dissipated by the processor; we present this in Figure 6(b) for both Naive and BestZig. We repeated the experiment for all problem sizes (x -axis), and plot the percentage deviation (in power dissipated) from Constant in the y -axis. We see that both Naive and BestZig dissipate more power than Constant, but the performance is very robust. Furthermore, BestZig (only 0.5% from Constant) significantly outperforms Naive (8%). These results are quite encouraging, since they indicate that the optimal Zig-Zag policy (for minimizing a variety of natural objective functions) does not increase the power dissipated (compared to Constant).

7 Conclusions

In this paper, we modeled the temperature of the system as a function of the power dissipated by the processor and the cooling system, and described how this information may be used by the scheduler to design efficient algorithms. Our analysis, and the computational illustration of such an algorithm for a real scenario are the key contributions of this paper.

We argued that simple system-throttling rules are effective scheduling policies even when few assumptions apply. These assumptions are reasonably general, but fail to accommodate two key features of real systems. Firstly, in the presence of release dates for jobs, we show that an intelligently constructed Zig-Zag policy can outperform a Constant policy (implementing system-throttling). However, by characterizing the exact necessary conditions, we show that even in this case,

Zig-Zag policies are often dominated by Constant. However, this analysis is quite useful since it allows us to develop the best Zig-Zag policy.

Secondly, in many settings, the system design does not permit the implementation of a Constant policy. This is particularly true of processors which allow only a discrete set of states of operation. As a result, the system is forced to Zig-Zag between alternate phases of cooling and heating to ensure that the system temperature does not exceed the threshold. Using our analysis on the trade-offs of a Zig-Zag policy, we develop an algorithm that calculates and implements the best Zig-Zag policy under any setting. This algorithm is quite general, and determines what speed to scale to, based on the current state of the system.

In this work, we characterized the exact trade-offs of implementing a Zig-Zag policy (with respect to Constant), and used this to derive the optimal Zig-Zag policy. In the future, we would like to extend our analysis in two directions. First, we would like to extend our analysis to real systems, by considering heterogeneous tasks (i.e., the amount of heat dissipated by the executions of a job given a certain amount of processing is different for different jobs) and multiprocessor systems. Second, we would like to incorporate power considerations into the decision making of our algorithms, thus minimizing power dissipation while ensuring that the temperature threshold is not violated.

References

1. Belady, C.: Cooling and power consideration for semiconductors into the next century. In: Macii, E., De, V., Irwin, M.J. (eds.) ISLPED, pp. 100–105. ACM, New York (2001)
2. Ma, M., Gunther, S.H., Greiner, B., Wolff, N., Deutsche, C., Arabi, T.: Enhanced thermal management for future processors. In: 2003 Symposium on VLSI Circuits, pp. 201–204 (2003)
3. Brooks, D., Martonosi, M.: Dynamic thermal management for high-performance microprocessors. In: HPCA (2001)
4. Skadron, K.: Hybrid architectural dynamic thermal management. In: DATE 2004: Proceedings of the conference on Design, automation and test in Europe, Washington, DC, USA, pp. 10–15. IEEE Computer Society, Los Alamitos (2004)
5. Rajan, D., Yu, P.S.: Temperature-aware scheduling: When is system-throttling good enough? Technical Report RC24331, IBM T. J. Watson Research Center (2007)
6. Hsu, C., Feng, W.: Reducing overheating-induced failures via performance-aware CPU power management. In: The 6th International Conference on Linux Clusters: The HPC Revolution 2005 (April 2005)
7. Weissel, A., Bellosa, F.: Process cruise control: Event-driven clock scaling for dynamic power management. In: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002), Grenoble, France (2002)
8. Merkel, A., Bellosa, F.: Balancing power consumption in multiprocessor systems. In: First Association for Computing Machinery (ACM) SIGOPS EuroSys Conference, pp. 403–414. ACM Press, New York (2006)

9. Gomaa, M., Powell, M.D., Vijaykumar, T.N.: Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system. In: ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, pp. 260–270. ACM Press, New York (2004)
10. Yao, F., Demers, A., Shenker, S.: A scheduling model for reduced cpu energy. In: FOCS 1995. Proceedings of the 36th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, p. 374. IEEE Computer Society, Los Alamitos (1995)
11. Bunde, D.P.: Power-aware scheduling for makespan and flow. In: Gibbons, P.B., Vishkin, U. (eds.) SPAA, pp. 190–196. ACM, New York (2006)
12. Bansal, N., Pruhs, K., Stein, C.: Speed scaling for weighted flow time. In: SODA (2007)
13. Bansal, N., Pruhs, K.: Speed scaling to manage temperature. In: Diekert, V., Durand, B. (eds.) STACS 2005. LNCS, vol. 3404, pp. 460–471. Springer, Heidelberg (2005)
14. Lorch, J., Smith, A.J.: A new approach to dynamic voltage scaling. *IEEE Transactions on Computers* 53(7), 856–869 (2004)
15. Jejurikar, R., Gupta, R.K.: Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. In: Joshi, R.V., Choi, K., Tiwari, V., Roy, K. (eds.) ISLPED, pp. 78–81. ACM, New York (2004)
16. Flautner, K., Mudge, T.: Vertigo: Automatic performance-setting for linux. In: OSDI 2002: Proceedings of the 5th symposium on Operating systems design and implementation, pp. 105–116. ACM Press, New York (2002)
17. Patel, C.D., Ranganathan, P.: Enterprise power and cooling. ASPLOS Tutorial (2006)
18. Bianchini, R., Rajamony, R.: Power and energy management for server systems. *Computer* 37, 68–74 (2004)
19. Bradley, D.J., Harper, R.E., Hunter, S.W.: Workload-based power management for parallel computer systems. *IBM J. Res. Dev.* 47(5-6), 703–718 (2003)
20. Seargeant, J.E., Krum, A.: *Thermal Management Handbook*. McGraw-Hill, New York (1998)
21. Bellosa, F., Weissel, A., Waitz, M., Kellner, S.: Event-driven energy accounting for dynamic thermal management. In: COLP 2003. Proceedings of the Workshop on Compilers and Operating Systems for Low Power, New Orleans, LA (2003)
22. Brooks, D.M., Bose, P., Schuster, S.E., Jacobson, H., Kudva, P.N., Buyuktosunoglu, A., Wellman, J.D., Zyuban, V., Gupta, M., Cook, P.W.: Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro* 20(6), 26–44 (2000)
23. Polyanin, A.D., Zaitsev, V.F.: *Handbook of Exact Solutions for Ordinary Differential Equations*. Chapman & Hall/CRC Press (2003)