

C-Oracle: Predictive Thermal Management for Data Centers *

Luiz Ramos and Ricardo Bianchini
Department of Computer Science, Rutgers University
{lramos,ricardob}@cs.rutgers.edu

Abstract

Designing thermal management policies for today's power-dense server clusters is currently a challenge, since it is difficult to predict the exact temperature and performance that would result from trying to react to a thermal emergency. To address this challenge, in this paper we propose C-Oracle, a software infrastructure for Internet services that dynamically predicts the temperature and performance impact of different thermal management reactions into the future, allowing the thermal management policy to select the best reaction at each point in time. We experimentally evaluate C-Oracle for thermal management policies based on load redistribution and dynamic voltage/frequency scaling in both single-tier and multi-tier services. Our results show that, regardless of management policy or service organization, C-Oracle enables non-trivial decisions that effectively manage thermal emergencies, while avoiding unnecessary performance degradation.

1. Introduction

The power density of the server clusters that run today's data centers has been increasing rapidly, as a result of their increasingly power-hungry hardware components, decreasing form factors, and tighter packing [24, 31]. High power densities entail high temperatures that now must be countered by sophisticated and expensive cooling infrastructures. Despite these infrastructures, several factors may cause high temperatures in data centers: hot spots at the top sections of racks, poor design of the cooling infrastructure or air distribution system, inadvertent blocking of air flows, failed fans or air conditioners, accidental overload due to hardware upgrades, or degraded operation during brownouts. We refer to these problems as "thermal emergencies".

It is important to deal with these emergencies, since high temperatures decrease the reliability of the affected components to the point that they start to behave unpredictably

or fail altogether. Even when components do not misbehave, operation outside the range of acceptable temperatures causes reliability to decrease exponentially [1, 8]. Thus, thermal management is critical in maintaining reliable operation without unnecessary performance losses.

Although some hardware components (most commonly the CPU) may have their individual thermal management mechanisms [3, 30], they do not leverage high-level information about the system as a whole, e.g. the temperature of multiple hardware components and servers, information about the resource requirements of the workload, the intensity of the current and projected load on the cluster, and the physical location of different servers. Furthermore, a common practice in thermal management is to simply turn off any server undergoing a thermal emergency (hereafter called a "hot server"), until the emergency can be manually addressed. Clearly, ignoring high-level information or relying on primitive thermal management may lead to poor resource utilization and unnecessary performance degradation under high enough load.

Recognizing this state of affairs, systems researchers have started to design more sophisticated (software) policies for managing thermal emergencies [6, 9, 12, 32]. Their policies worked well for the sample emergencies they considered. However, the policies typically implement a single pre-defined reaction to a thermal emergency. For example, the reaction may be to direct a certain amount of load away from a hot server. When one knows the details of the emergency (and the other aspects of the experiments, such as the load across the data center), as in these papers, it is easy to tune the reaction (the amount of load to direct away from the hot server in the above example) for ideal behavior. Unfortunately, in practice, a single pre-defined reaction does not suffice, since it is difficult to predict the exact temperature and performance result of the reaction for different emergencies. Reactions that are excessively severe may cause unnecessary performance degradation and/or generate emergencies in other parts of the system, whereas reactions that are excessively mild may take relatively long to become effective (if at all), compromising the reliability of the system. Even policies that include feedback control can suffer

*This research has been supported by NSF under grant #CCR-0238182 (CAREER award) and by a doctoral fellowship from CAPES, Brazil.

from these problems (as well as their traditional parameter-tuning difficulties). Thus, to decide on the best course of action, the thermal management policy needs the ability to evaluate multiple reactions to each thermal emergency dynamically and accurately.

To address this challenge, in this paper we propose C-Oracle (short for Celsius-Oracle), a software prediction infrastructure for a major class of data center-based systems, Internet services. Popular services must be available and serve the entire offered load all the time, so even rare thermal emergencies must be managed with minimal disruption.

C-Oracle dynamically predicts the temperature and performance impact of different reactions into the future, allowing the thermal management policy to select the best course of action at each point in time. C-Oracle makes predictions based on simple models of temperature, component utilization, and policy behavior that can be solved efficiently, on-line.

We evaluate C-Oracle for two thermal management policies and service organizations. First, we implement a C-Oracle version of Freon [12], a feedback-controlled policy that explicitly directs requests away from hot servers, while limiting the maximum number of concurrent requests that they receive. In this implementation, called C-Freon, C-Oracle predicts the temperature and throughput impact of different levels of load redistribution in Freon. We study C-Freon for a single-tier Web service. Second, we implement a C-Oracle version of LiquidN2, a novel feedback-controlled policy that combines CPU dynamic voltage/frequency scaling (DVFS), request redistribution, and request-admission control to manage emergencies. In this implementation, called C-LiquidN2, C-Oracle predicts the temperature and throughput impact of different scales of voltage/frequency in LiquidN2. We study C-LiquidN2 for a more complex three-tier online auction service.

Our results show that the C-Oracle temperature and performance predictions are very accurate. The results also demonstrate that, regardless of management policy or service organization, C-Oracle enables non-trivial decisions that effectively manage thermal emergencies, while avoiding unnecessary performance degradation.

2. Related Work

Data center thermal management. Prior work has considered two classes of thermal management policies for data centers: those that manage temperatures under normal operation and those that manage thermal emergencies.

The main goal of policies for managing temperatures under normal operation has been to reduce electricity costs. The key concern of these policies is to place workloads across the servers so that energy consumption is minimized; performance is assumed to be unaffected by the workload

placement. Along these lines, Sharma *et al.* [29] and Moore *et al.* [19, 18] have studied policies that optimize cooling energy costs, whereas Mukherjee *et al.* [20] also considered computing energy costs for bladed systems.

In contrast, the main goal of policies for managing thermal emergencies has been to control temperatures while avoiding unnecessary performance degradation. The key concern here is to avoid turning off hot servers, which would reduce processing capacity. In this context, Heath *et al.* have proposed policies that manage emergencies in Internet services using request redistribution and request-admission control [12]. Ferreira *et al.* considered similar policies in simulation [9]. For batch workloads, Choi *et al.* considered speeding up fans and CPU DVFS for managing emergencies using offline simulation [6]. Finally, Weissel and Bellosa studied the throttling of energy-intensive tasks to control CPU temperature in multi-tier services [32].

Our paper considers policies in this latter class. In particular, we propose LiquidN2, a novel policy that combines DVFS, request redistribution, and request-admission control for Internet services. Previous DVFS-based thermal management policies either focused on a single CPU or did not consider redistribution and admission control, e.g. [3, 6, 32]. More importantly, we create C-Oracle, an on-line temperature and performance prediction infrastructure, and demonstrate it in the context of Freon and LiquidN2.

Prediction-based thermal management. C-Oracle is the first infrastructure to allow on-line prediction of future temperatures in evaluating potential reactions to emergencies.

In contrast, both Choi *et al.* [6] and Ferreira *et al.* [9] considered evaluating reactions offline. These approaches may work well given two assumptions: the system has a way to identify thermal faults precisely, and the set of all possible thermal faults is known in advance. In this paper, we make neither assumption. First, identifying thermal faults precisely requires instrumentation that is rarely available in real systems. For example, identifying failures in the air distribution system could require that air-flow sensors be placed all over the data center, including inside the servers' chassis. Second, many potential faults are difficult to anticipate and model. For example, while one can anticipate a failure that causes a case fan to stop running, it is much harder to anticipate the extent of the clogging of air filters at the inlet of different servers.

In [18], the authors used Weatherman, an on-line temperature prediction system based on a neural network, to evaluate different workload placements for reducing cooling costs in batch-processing data centers. C-Oracle has a thermal model component that is equivalent in functionality to Weatherman. However, an infrastructure for managing emergencies also needs to model the thermal management policy itself at a fine grain, in terms of both temperature and performance, which they did not consider.

Single-device thermal management. As temperature is a major concern for modern CPUs, several researchers have considered CPU thermal management at the architecture level, e.g. [3, 16, 28, 30]. Others have considered software-based CPU thermal management, e.g. [2, 10, 26, 32]. A few researchers have also considered thermal management of disks [11, 15] and memories [17].

Perhaps most similar to our work is [14], which proposed a framework for managing CPU temperature and energy that could combine many techniques. However, the choice of techniques was not based on on-line predictions of their behavior, but rather on a pre-established ordering based on their effectiveness.

Although these policies and framework have a role in managing device temperature at a fine grain, as discussed in the previous section, data center thermal management can be substantially more effective when higher level, system-wide information is considered.

Energy management for server clusters. Many works have been done on conserving energy in data centers, e.g. [4, 5, 7, 13, 22, 23, 27]. None of them considered thermal management (which is related but qualitatively different than energy management [12]) or selecting among multiple candidate policies. Heath *et al.* did consider one combined energy/thermal management policy [12].

Interestingly, for stand-alone systems, Pettis *et al.* did select the best energy-conservation policy based on their estimated behavior on the recent accesses to each I/O device [21]. However, given the differences in environment and goals, not much else is similar between our work and theirs.

3. Thermal Management Policies

In this section, we describe three policies for Internet services: turning hot servers off, Freon, and LiquidN2. In the end of the section, we discuss the need for predicting temperature and performance in the latter policies.

3.1. Turning Hot Servers Off

This policy simply turns off any server in which a component has reached its red-line temperature (the maximum temperature that does not cause a significant reliability degradation). When human operators detect an obvious thermal fault, such as the failure of one of the air conditioners, servers may be turned off even earlier than this point. Servers are only turned back on when operators have manually eliminated the source of the emergencies.

This policy is very simple but only works well when the servers that remain active have enough processing capacity to serve the entire offered load. When this is not the case, the policy causes performance degradation, i.e. decreased throughput, dropped requests, and/or increased re-

quest latency. (In this paper, we focus solely on throughput and dropped requests as performance metrics.) For example, consider a simple, single-tier Web service with homogeneous servers. When the average server utilization is below 50% under normal conditions, the service can withstand a thermal fault that causes half of its servers to turn off. If the average normal-conditions utilization is higher than 50%, turning off half of the servers will cause performance loss.

Therefore, for services that over-provision their clusters significantly beyond their normal offered load, this simple policy may never become a problem. Unfortunately, over-provisioning to this extent has multiple significant costs, such as management labor, power, and energy costs. For these reasons, Internet services have an incentive to operate with higher utilizations. (In fact, high operational costs and virtualization technology are prompting extensive server consolidation in today’s data centers [24].) For these services, turning hot servers off degrades performance under high enough load.

The Freon and LiquidN2 policies are motivated by these modestly over-provisioned systems. The policies are based on the observation that it is possible to keep hot servers active, as long as we reduce the heat they produce by a sufficient amount [9, 12]. Heat reduction can be accomplished by (1) running the hot servers at full speed but moving load away from them (as in Freon); or (2) voltage/frequency scaling hot servers down and moving less load away from them (as in LiquidN2). Getting back to the example above, if we can control temperatures by running half of the servers at 25% utilization (and full speed), the average normal-conditions utilization would have to be higher than 62.5% for performance to suffer.

More generally, for the performance of a homogeneous multi-tier service not to suffer under thermal emergencies, the loss in processing capacity resulting from thermal management of any tier has to be no greater than the available capacity of the servers unaffected by the emergencies in the same tier. Formally, the following must be true of every tier:

$$N_h(Avg_n - Avg_h) \leq (N - N_h)(1 - Avg_n)$$

where N and N_h are the total number of servers and the number of hot servers, respectively, in the tier; Avg_n is the average utilization of the most utilized hardware component (e.g., CPU, disk) of the tier under normal conditions; and Avg_h is the average utilization of the most utilized hardware component of the hot servers of the tier, after a thermal management policy has been activated.

We are interested in scenarios for which the simple policy of turning hot servers off causes performance degradation. For this policy, $Avg_h = 0$. Thus, we care about scenarios in which $Avg_n > (N - N_h)/N$ for at least one service tier.

Note that the above formula and examples make the simplifying assumption that the performance of a server only

starts to suffer when the utilization of one of its components reaches 100%. However, performance typically starts to suffer at lower utilizations, such as 80% or 90%. To adjust for this effect, we can simply replace the value 1 for 0.8 or 0.9 in the formula. We can also roughly approximate the effect of heterogeneity and voltage/frequency scaling policies by normalizing all the utilizations to the same reference. For example, we can normalize the utilization of a CPU running at half of its performance (in requests/second) to its full performance by dividing its utilization by 2.

3.2. Freon

Freon manages component temperatures using a combination of request redistribution, admission control, and feedback control. Although Freon can be used in multi-tier services, the implementation described in [12] managed a single-tier Web server cluster fronted by a load balancer that uses a weight-based request distribution policy. Besides the load balancer, Freon has a monitoring daemon, called `tempd`, running on each server and an admission-control daemon, called `admd`, on the load-balancer node. `Tempd` wakes up periodically (once per minute in our experiments) and measures the temperature of the CPU(s) and disk(s) of the server. `Tempd` is also responsible for triggering and deactivating the thermal reactions. `Admd` is responsible for configuring the load balancer as prompted by `tempd`. The load balancer we use currently is LVS, a kernel module for Linux, with weighted round-robin request distribution.

Freon defines three thresholds per monitored component (c): higher (T_h^c), lower (T_l^c), and red line (T_r^c). A thermal reaction is triggered when the temperature of any component has crossed T_h^c . At that point, `tempd` sends a UDP message to `admd` for it to adjust the load offered to the hot server. The daemon communication and load adjustment are repeated periodically (once per minute in our experiments) until the component’s temperature becomes lower than T_l^c . Below that point, `tempd` orders `admd` to eliminate any restrictions on the offered load to the server. For temperatures between T_h^c and T_l^c , Freon does not adjust the load distribution, as there is no communication between the daemons. Freon only turns off a hot server when the temperature of one of its components exceeds T_r^c . This threshold marks the maximum temperature that the component can reach without serious degradation to its reliability.

The specific information that `tempd` sends to `admd` is the output of a PD (Proportional and Derivative) feedback controller. The output of the PD controller is computed by:

$$output^c = \max(k_p(T_{curr}^c - T_h^c) + k_d(T_{curr}^c - T_{last}^c), 0)$$

$$output = \max\{output^c\}$$

where k_p and k_d are gain constants (set at 0.1 and 0.2, respectively, in our experiments), and T_{curr}^c and T_{last}^c are the

current and the last observed temperatures. Note that the controller only runs when the temperature of a component is higher than T_h^c and forces *output* to be non-negative.

Based on *output*, `admd` forces LVS to adjust its request distribution by setting the hot server’s weight so that it receives only $1/(output + 1)$ of the load it is currently receiving. The reduction in weight naturally shifts load away from the hot server, increasing the loads of other servers as a result. In addition, to guarantee that an increase in overall offered load does not negate this redirection effect, Freon also orders LVS to limit the maximum allowed number of concurrent requests to the hot server at the average number of concurrent requests directed to it over the last time interval (one minute in our experiments). To determine this average, `admd` wakes up periodically (every five seconds in our experiments) and queries LVS. If the emergency affects all servers or the unaffected servers cannot handle the entire offered load, requests are unavoidably lost.

3.3. LiquidN2

Freon is capable of managing thermal emergencies at a fine grain, using request-based load redistribution and admission control. However, load redistribution and admission control become more difficult to effect precisely when services involve session state, i.e. state that is associated with each user’s session with the service. For example, e-commerce services typically implement the notion of a shopping cart; the cart is a piece of state associated with a particular user session. In multi-tier services, the cart is often stored at one (or a couple) of the application servers, since storing it at the client would consume excessive network bandwidth and storing it at the database would unnecessarily load this tier. Thus, requests that belong to sessions started with hot servers still have to be routed to these servers; only new session requests can be moved away to manage temperatures.

Given the difficulty in managing loads in multi-tier services, we propose a new policy, called LiquidN2, that relies on DVFS to help request redistribution and admission control in managing temperatures. LiquidN2 assumes that the service has a load-balancing device in front of each tier of servers. (Instead of having multiple devices, using LVS, it is also possible to use a single load-balancing node, as done in our experiments.) The balancers implement least-connections distribution with “sticky sessions”, which guarantee that requests from the same session are processed by the same server in each tier.

Because hot servers run slower, producing less heat than servers running at full speed for the same workload, LiquidN2 moves less load away from hot servers than Freon. Furthermore, the load redistribution in LiquidN2 is a result of back pressure from the hot servers, which in turn affects

the admission control decisions, rather than explicit changes to the load balancer weights as in Freon.

Nevertheless, LiquidN2 was inspired in Freon, so the two policies have similarities in terms of software components and their functionalities. Specifically, a monitoring daemon runs on each server and an admission-control daemon runs on each load-balancer node. The communication between the daemons occurs in the same way as in Freon. LiquidN2 also defines the same temperature thresholds for high (T_h^c), low (T_l^c), and red-line (T_r^c) temperatures.

The policy is triggered when the temperature of any component at any server has crossed T_h^c . The reaction to this emergency is to change the CPU voltage and frequency of the affected server. LiquidN2 calculates the change needed using the same feedback control loop used by Freon (we also keep the k_p and k_d gains the same in our experiments). Assuming that the CPU has a limited set of discrete operational frequencies (with their corresponding voltages), we set the new frequency to the highest frequency that is lower than $current_frequency / (output + 1)$. Since the server's speed is reduced, we adjust the admission control so that the server can execute as much work as it is able at the new frequency. We compute this correction value based on the estimated utilization under the new frequency, *estimated*, and a target of 85% utilization as $correction = 1 - ((estimated - 85) / estimated)$. With this value, we limit the maximum number of concurrent requests that is allowable at a hot server to *correction* times the average number of concurrent requests in the previous observation interval (one minute in our experiments).

Note that LiquidN2 slows down the CPU even if the thermal emergency is occurring in another hardware component. The assumption is that doing so will reduce the amount of load sent to the hot server and, thereby, indirectly reduce the load on the hot component.

3.4. The Need For Prediction

One might argue that predictions are unnecessary for policies such as Freon and LiquidN2, since their feedback control could automatically adjust the intensity of reactions to manage temperatures with as little performance degradation as possible. However, this argument overlooks many of the complexities associated with feedback control and software-based thermal management:

1. Feedback control as in these policies relies on careful tuning of the gain constants, which can be tricky and time-consuming, especially when done by trial-and-error;
2. The gain constants may have to be tuned differently for different types of emergencies;
3. The output of the controller needs to be transformed in a thermal reaction that will affect the load balancing directly

(as in Freon) and the voltage/frequency setting of the CPUs (as in LiquidN2). This transformation also needs to be tuned for proper behavior;

4. The intensity of the reaction and the temperature thresholds relate to the monitoring interval of the thermal management system. For example, in Freon and LiquidN2, T_h^c is typically set just below T_r^c , e.g. $2^\circ C$ lower, depending on how quickly the component's temperature is expected to rise in between observations. However, a weak reaction to an emergency may cause a temperature that is just below T_h^c to increase by more than $T_r^c - T_h^c$ during one observation interval, forcing the shutdown of the server; and

5. The ideal intensity for a reaction may depend on the status of the rest of the system. For example, when a thermal fault is localized at a single server, a strong reaction may be most appropriate, since the large amount of load moved to the other servers would cause no new emergencies. However, when a fault affects multiple servers, the weakest reaction that will bring the highest temperatures down is the best course of action, to avoid unnecessarily causing dangerous temperatures at other servers.

Given these complexities, we argue that it is simpler to quickly evaluate a number of gain settings and/or reaction intensities on-line and select the best combination based on predictions of their temperature and performance impact.

4. C-Oracle

In this section, we overview C-Oracle and describe its temperature and performance models, the current implementation, and predictive versions of Freon and LiquidN2.

4.1. Overview

C-Oracle is a software infrastructure that allows the thermal management policy of an Internet service to predict the implications of different potential reactions to a thermal emergency. The predictions are quickly computed at runtime, and can be requested by the policy at any moment. Whenever more than one reaction or reaction configuration is possible, the policy may send one prediction request per possibility to C-Oracle. For each request, C-Oracle then simulates the corresponding reaction into the future to predict the temperature of each component of the service, as well as the service performance. Given a summary of these predictions, the thermal management policy can select the best reaction to be performed in the real service.

Policies make decisions based on predictions of the system status several, say 5 or 10, minutes into the future. During this period, certain system conditions may change, e.g. the offered load to the service may increase significantly or a new emergency may develop, rendering the previous predictions inaccurate. To make sure that inaccurate predictions

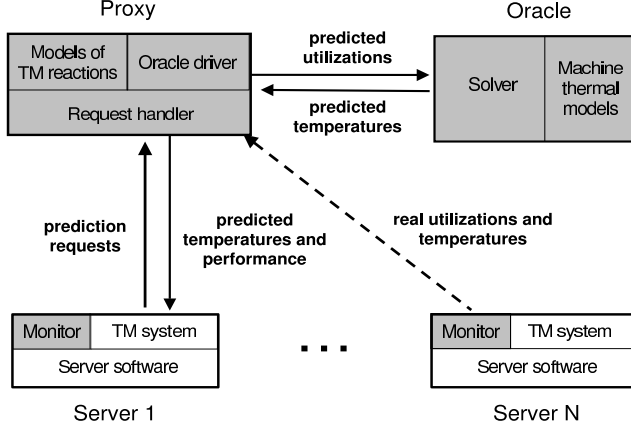


Figure 1. C-Oracle infrastructure (shaded). TM = thermal management.

are corrected quickly, C-Oracle also maintains temperature and performance predictions for every intermediate simulated minute. These intermediate predictions may be compared to the real state of the system at the appropriate times to determine whether the last choice of reaction should be reconsidered. If so, a new set of predictions may be immediately requested and a new decision made.

4.2. Design and Implementation

Our current design is depicted in Figure 1. It includes an *oracle* subsystem that predicts future temperatures, a *proxy* subsystem that receives requests from the thermal management policy, predicts performance, and drives the oracle, and a *monitor* daemon that collects temperature and utilization data from the servers and sends them to the proxy.

4.2.1. Oracle Subsystem

The main challenge of C-Oracle is to predict accurate temperatures on-line. Clearly, using conventional simulators to predict temperatures is impractical, due to their high running times. Real measurements are not possible either, since they cannot be accelerated to produce the predictions. The only remaining option is modeling. Fortunately, Heath *et al.* have proposed a thermal model that is simple enough to be solved on-line and yet produces predictions that are accurate enough for software-based thermal management [12]. Our insight in this paper is that their model can be leveraged in predicting future temperatures efficiently. We thus use their model (and their model solver), which is publicly available on the Web.

Thermal model. The model simplifies the real world into 5 simple equations. The first deals with conservation of energy, which translates into the conservation of heat in this

case. There are two sources of heat for a object in our system: it converts power into heat while it performs work, and it may gain (lose) heat from (to) its environment. This can be expressed as:

$$Q_{gained} = Q_{transfer} + Q_{component} \quad (1)$$

where $Q_{transfer}$ is the amount of heat transferred from/to a hardware component for a period of time and $Q_{component}$ is the amount of heat produced by performing work for a period of time.

The second equation is Newton’s law of cooling, which states that the amount of heat transferred between two objects or between an object and its environment is directly proportional to the temperature difference between them:

$$Q_{transfer,1 \rightarrow 2} = k \times (T_1 - T_2) \times time \quad (2)$$

where $Q_{transfer,1 \rightarrow 2}$ is the amount of heat transferred between objects 1 and 2 or between object 1 and its environment during a period of time, T_1 and T_2 are their current temperatures, and k is a constant that embodies the heat transfer coefficient and the surface area of the object. The k constant can be approximated by experiment, simulation, or rough numerical approximation.

The third equation defines the heat produced by a component, which corresponds to the energy it consumes:

$$Q_{component} = P_{utilization} \times time \quad (3)$$

where $P_{utilization}$ represents the average power consumed by the component as a function of its utilization.

The fourth equation assumes a simple linear formulation for the average power, which has approximated accurately enough the real power consumption of the components that we have studied so far:

$$P_{utilization} = P_{base} + utilization \times (P_{max} - P_{base}) \quad (4)$$

where P_{base} is the power consumption when the component is idle and P_{max} is the consumption when the component is fully utilized. (The default linear formulation can be easily replaced by a more sophisticated one.) For the CPU component, P_{base} and P_{max} are defined for each level of voltage/frequency.

Finally, the fifth equation defines an object’s temperature variation. Since pressures and volumes are assumed constant, the model defines the temperature of an object as directly proportional to its internal energy:

$$\Delta T = \frac{1}{mc} \times \Delta Q \quad (5)$$

where m is the mass of the object and c is its specific heat capacity.

Inputs and calibration. The model takes three groups of inputs: simple heat- and air-flow graphs describing the rough layout of the hardware and the air circulation, constants describing the physical properties of the flows and the power consumption of the hardware components, and dynamic component utilizations.

Defining the physical-property constants may be difficult, so is often useful to have a calibration phase, where a single, isolated machine is tested as fully as possible, and then the heat- and air-flow constants are tuned until the simulated temperatures match the real measurements. The constants describing the power consumption of the components can be determined from their data sheets or using microbenchmarks that exercise each component, power measurement infrastructure, and standard fitting techniques.

Model solver. Using discrete time-steps, the solver does three forms of graph traversals: the first computes inter-component heat flow; the second computes intra-machine air movement and temperatures; and the third computes inter-machine air movement and temperatures. The heat-flow traversals use the model equations, the input constants, and the dynamic component utilizations. The air-flow traversals assume a perfect mixing of the air coming from different servers, and compute temperatures using a weighted average of the incoming-edge air temperatures and amounts for every air-region vertex.

Heath *et al.* proposed their thermal model and solver for temperature emulation, not prediction. Next, we discuss how the proxy exercises the oracle to obtain temperature predictions.

4.2.2. Proxy Subsystem

The proxy subsystem collects component utilization and temperature information from the monitors; receives, processes, and responds to prediction requests from the thermal management policy; and drives the oracle subsystem.

To fulfill a prediction request, the proxy derives a large set of predicted temperatures and performance, e.g. one temperature prediction per server component across the entire service. However, rather than overwhelming the thermal management system with all that information, the proxy summarizes it. Specifically, in our current implementation, the proxy replies to each request with: (1) the highest observed temperature during the simulated period, (2) the identifier of the machine that reached the highest temperature, (3) the final temperature of that machine, and (4) the estimated amount dropped load.

To predict the temperatures associated with a request, the proxy repeatedly calls the oracle’s thermal model solver (once per simulated second in our experiments), passing predicted component utilizations to it. Predicting utilizations could consider their current values, predicted varia-

```

1 # for each server s in the cluster
2 if getTemp(s,disk) > high(s,disk) then
3   W = 1/(getTemp(s,disk)-getPrevTemp(s,disk)+1)
4   setLBWeight(s,W)
5 else
6   if getTemp(s,disk) < low(s,disk) then
7     W = maximum weight value
8     setLBWeight(s,W)
9 # compute new utilizations and dropped load
10 reBalanceLoad(s)

```

Figure 2. Model of thermal reaction.

tions in the intensity of the load offered to the service, and the reactions taken by the management policy.

The current implementation of the proxy subsystem does not attempt to predict future load intensities; it assumes that the current component utilizations are stable and only affected by the thermal management policy. This simplification is not a serious limitation because predictions are periodically verified and, when inaccurate, they are recomputed starting with the updated real utilizations.

For the proxy to model the thermal management policy and its effect on utilizations, the policy designer must provide an operational model of each of the possible thermal reactions. These models are executed by the proxy to compute predicted utilizations. Each model can be defined by simply taking the original policy source code and replacing (1) accesses to real temperatures and utilizations, and (2) changes to the load balancing configurations, with primitives provided by the proxy. The primitives for (1) access the simulated temperatures and utilizations, whereas those for (2) guide the computing of predicted utilizations.

To see a simple example of a thermal-reaction model, consider Figure 2. In line 2, the model compares the current simulated temperature of the disk with its pre-set high threshold. In lines 3 and 4, the model changes the simulated load-balancer weight associated with server *s* to reduce its load proportionally to the temperature increase since the last observation interval. The weight is restored when the simulated temperature falls below the pre-set low threshold (lines 6 to 8). Line 10 computes the new simulated utilizations and the percentage of dropped load. `getTemp()`, `getPrevTemp()`, `high()`, `low()`, `setLBWeight()`, and `reBalanceLoad()` are some of the primitives that the proxy currently provides. There are also some primitives to simulate admission control and DVFS. The reaction models are currently compiled with the proxy, but we may enable dynamic insertion and removal of simulation models in the future.

As the proxy estimates the future utilizations (as part of the `reBalanceLoad()` primitive), it also estimates the performance impact of each reaction. Specifically, it estimates the amount of load that may be dropped by a reaction by accumulating the utilization that would exceed a pre-set maximum (say 85%) at each server. This simple estima-

```

1 # given a prediction p belonging to the set {p1, p2}
2 if all highestTemperatures > "red line" then
3   select p with lowest highestTemperature
4 else
5   if all finalTemperatures < "higher" then
6     if amounts of droppedLoad are not equal then
7       select policy with lowest droppedLoad
8     else
9       select policy with lowest finalTemperature
10  else
11    select policy with lowest finalTemperature

```

Figure 3. Decision algorithm of C-Freon.

tion assumes that any load balancing policy will attempt to even out the load offered to the servers that are operating normally, while restricting the load offered to servers undergoing thermal emergencies according to the thermal management policy decisions.

To deal with mispredictions, the proxy maintains a temperature prediction per server, per component, per simulated observation interval, while the prediction has not expired. The proxy compares those values against the real temperatures measured at the end of each real observation interval. A prediction is invalidated if a real temperature differs from the corresponding predicted temperature by more than $1^{\circ}C$. The policy can query the status of a prediction using a library call provided by the proxy.

4.2.3. Monitor Daemon

A monitor daemon runs on each server in the cluster. The daemon measures the utilization and temperature of hardware components and periodically sends that information to the proxy (every second in our experiments). Temperatures can be measured with real thermal sensors or obtained from the Mercury emulator [12]. Our current implementation takes the latter approach, which gives us the ability to repeat experiments more consistently and the ability to study thermal faults without harming the reliability of our real hardware. We collect utilization information from the operating system by reading `/proc`.

Another monitor daemon (not shown in Figure 1) runs on the load-balancer node(s), collecting information such as load-balancer weights, number of active concurrent connections, number of established connections, and the number of incoming and outgoing bytes. In our experiments with C-Freon, we only needed the load-balancer weights to perform predictions, whereas in the experiments with C-LiquidN2, we did not use any data from the daemons that monitor the load balancers.

4.3. Predictive Policies

The proxy provides predictions that can be used by thermal management policies. C-Freon and C-LiquidN2, the predictive versions of our policies, use the predictions.

```

1 # temperature is back to normal
2 if getTemp(s,component) < low(s,component)
3   setFreq(s,CPU,maxfreq)
4   setAdm(s,0)
5 else
6   # 'output' from the feedback controller
7   newfreq = ceil(getFreq(s,CPU)/(output+1))
8   if newfreq < minfreq
9     newfreq = minfreq
10  else
11    newfreq = next freq lower than newfreq
12  # estimated = utilization due to change in freq
13  estimated = getUtil(s,component) x
14             getFreq(s,CPU) / newfreq
15  factor = 1 - ((estimated - 85) / estimated)
16  # adjust admission control by factor
17  setAdm(s,getAvg(s) * factor)
18  setFreq(s,CPU,newfreq)
19 # compute new utilizations and dropped load
20 reBalanceLoad(s)

```

Figure 4. C-LiquidN2's weak reaction.

4.3.1. C-Freon

C-Freon exploits C-Oracle to decide between two reaction intensities, called *weak* and *strong*. The weak reaction was described in Section 3.2: it changes the load-balancer weight of a hot server so that the server receives only $1/(output + 1)$ of its current load. The strong reaction moves 6 times more load away from a hot server than the weak reaction. (Note that C-Freon could compare more reaction configurations, including a few that varied the gain constants as well, but two configurations are enough to make our points.) C-Freon models the two reaction strengths using a similar structure and primitives as in Figure 2.

The algorithm that C-Freon uses to select a reaction is depicted in Figure 3. The idea is to select the candidate reaction that brings component temperatures below their T_h^c with the smallest performance degradation (lines 5 to 9). In addition, if all reactions would eventually cross the red-line temperature, C-Freon selects the reaction with the smallest highest temperature (lines 2 and 3).

C-Freon also periodically invokes the proxy to check the status of the prediction corresponding to the reaction last selected. In case the proxy finds the prediction to be inaccurate, C-Freon requests a new set of predictions and may select a different reaction as a result.

4.3.2. C-LiquidN2

Like C-Freon, C-LiquidN2 uses our C-Oracle infrastructure to decide between two thermal reactions, called *weak* and *strong*. The weak reaction is that described in Section 3.3: it adjusts the voltage/frequency of the CPU to the highest frequency that is lower than $current_frequency/(output + 1)$. The strong reaction reduces the voltage/frequency 4 times more aggressively.

Figure 4 shows the model of the weak reaction in C-LiquidN2. This figure demonstrates that applying C-Oracle

to a complex policy and service (like LiquidN2 and the auction) is not much more difficult than doing so for a simple policy and service (like Freon and a Web service). In fact, the service complexity is hidden behind the `reBalanceLoad()` primitive (line 20), which computes the simulated rebalancing for the tier in which server s resides. Further evidence of the flexibility and simplicity of our infrastructure is in the fact that C-LiquidN2 uses the same reaction-selection algorithm (Figure 3) and approach to checking predictions as in C-Freon.

5. Experimental Results

In this section, we first evaluate the performance of the C-Oracle predictions. After that, we evaluate C-Oracle in terms of its prediction accuracy and the ability of C-Freon and C-LiquidN2 to manage thermal emergencies effectively.

5.1. Prediction Performance

Our experiments demonstrate that the proxy can produce temperature and performance predictions efficiently. For the small services (4-5 servers) that we study in this paper, one prediction for 5 minutes into the future is computed in roughly 40 milliseconds (measured at the thermal management policy) when the proxy and the oracle share a server with two 2.8-GHz Pentium 4 Xeon processors. Time increases sub-linearly with number of predictions. For example, 16 predictions can be performed in only 350 milliseconds under the same assumptions. Time increases linearly with how long into the future we request each prediction.

Even though we made no attempts to improve this performance, it is clear that, for services with hundreds or thousands of servers, parallel processing of the temperature predictions may be necessary. Although the sequence of oracle solver calls involved in a prediction needs to be performed sequentially, each call can be processed in parallel (the simulation of the component temperatures of different servers can be easily parallelized). Predictions corresponding to different thermal reactions can also be processed in parallel. The current implementation executes each solver call sequentially and creates one process for each prediction.

5.2. C-Freon

Methodology. We run the Freon and C-Freon experiments using a single-tier Web service with 4 Apache servers behind an LVS load balancer. The Apache servers and LVS run on machines with a 2.8-GHz Pentium 4 Xeon processor, 1GB of RAM, and a 10K-rpm SCSI disk, whereas our proxy and oracle share a dual 2.8-GHz Pentium 4 Xeon-based server with 2GB of RAM. As the workload, we use

a synthetic Web trace that includes 30% of requests to dynamic content in the form of a simple CGI script that computes for 14 ms and produces a small reply. The timing of the requests mimics the well-known traffic pattern of most Internet services, consisting of recurring load valleys (over night) followed by load peaks (in the afternoon). The load peak is set at 70% utilization with 4 servers, leaving spare capacity to handle unexpected load increases or a server failure. The trace is submitted to the load balancer by a client process running on a separate machine.

During the first half of the experiment, we inject an emergency at one server by raising its inlet temperature to $44^{\circ}C$. The emergency lasts until the end of the experiment. During the second half, we inject an emergency at another server by raising its inlet temperature to $40^{\circ}C$. This emergency also lasts until the end of the experiment.

When configuring Freon and C-Freon, we set $T_h^{CPU} = 68^{\circ}C$, $T_l^{CPU} = 62^{\circ}C$, $T_r^{CPU} = 72^{\circ}C$, $T_h^{disk} = 65^{\circ}C$, $T_l^{disk} = 62^{\circ}C$ and $T_r^{disk} = 67^{\circ}C$, which are the proper values for our components, according to their specifications. Given our workload, the emergencies we inject, and these threshold settings, we find that the CPU is the most seriously threatened component in our experiment. As described in Section 4.3.1, C-Freon evaluates two reactions, weak and strong, before reacting to a thermal emergency.

Note that we use the Mercury Suite [12] to emulate server temperatures, as opposed to measuring them using real thermal sensors. This allows us to inject thermal emergencies into the system without fear of harming the real hardware, while making the experiments repeatable and the instrumentation simpler. Mercury has been validated to within $1^{\circ}C$ of real temperatures for Pentium III-based servers [12]. Before using Mercury in this paper, we extended it to handle DVFS, calibrated it for our Xeon-based servers, and again validated it to within $1^{\circ}C$ of real temperatures. We do not present all the validation results here due to space limitations; the interested reader can find them at <http://www.darklab.rutgers.edu/MERCURY/>.

Results. Figures 5 and 6 show the results of our C-Freon experiment. Figure 5 displays one curve for the CPU temperature of each server, dots representing the CPU temperature predictions (one dot per future minute) for each server, and the CPU temperature setpoint (the high threshold, $T_h^{CPU} = 68^{\circ}C$). For each server, Figure 6 displays one curve for the CPU utilization, and dots representing the CPU utilizations it predicts (one dot per future minute) in making its temperature predictions.

From the temperature graph, we can see that the first emergency caused the temperature of the CPU of server 4 to rise quickly. When the temperature crossed the high threshold, C-Freon adjusted the LVS load distribution to reduce the load on that server. Only one adjustment was necessary. This caused the other servers to receive a larger fraction

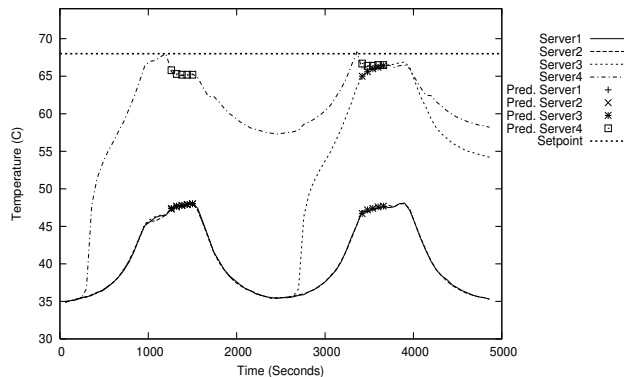


Figure 5. C-Freon CPU temperatures.

of the workload, as the CPU-utilization graph illustrates. Around 1500 seconds, the offered load started to subside, decreasing the utilizations and temperatures. When the temperature became lower than the low threshold, C-Freon reset the load balancer settings. Around 2500 seconds, the offered load started to increase again. At that point, the temperature of the CPU of server 4 was already high. The second emergency caused the temperature at server 3 to increase as well. When server 4 again crossed the high threshold, C-Freon made a new adjustment, which lasted until temperatures started to subside at the end of the experiment.

We can make two main observations here. First, C-Oracle produces temperature predictions that are very accurate, whereas its predicted utilizations are not as accurate but are still close to the real average utilizations. Second, C-Freon effectively manages the thermal emergencies, while not causing any dropped requests. To manage the first emergency, C-Freon decided to use the strong reaction, since it would bring server 4 down to a lower temperature than the weak reaction after 5 minutes. For the second emergency, C-Freon decided to use the weak reaction, since the strong reaction would have forced server 3 to cross the high threshold as well. We do not show figures for Freon with the weak and strong reactions independently, due to space limitations.

Note that, as confirmed by our simple analysis in Section 3.1, turning server 4 off during the first half would have produced little or no performance degradation ($0.7 \leq 0.75$, for $Avg_n = 0.7$, $N = 4$, and $N_h = 1$). However, during the second half, turning server 4 off would have caused server 3 to be turned off, leading to a scenario in which a relatively large fraction of requests would be dropped ($0.7 > 0.5$, for $Avg_n = 0.7$, $N = 4$, and $N_h = 2$).

5.3. C-LiquidN2

Methodology. For the experiments with LiquidN2 and C-LiquidN2, we use an online auction service and a client emulator, both from [25]. The service is organized into 3 tiers

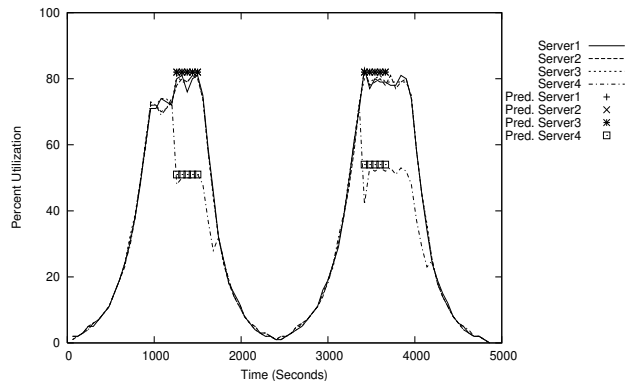


Figure 6. C-Freon CPU utilizations.

of servers: Web, application, and database. In the first tier, we use 2 servers running Apache, each with a 2.8-GHz Pentium 4 Xeon processor, 1GB of RAM, and a 10K-rpm SCSI disk. In the second tier, we use 2 servers with the same hardware configuration, running the Tomcat servlet server. Finally, in the database tier, we use a single server with a dual 2.8-GHz Pentium 4 Xeon, 2GB of RAM, and a 10K-rpm SCSI disk, running the MySQL relational database. A node with the same configuration of the Web and application servers runs LVS to balance load between the first and second tiers and between the clients and the first service tier. Our proxy and oracle subsystems share a machine with the same configuration as the database server. The CPUs of all servers have 8 voltage/frequency levels, ranging from 2.8 GHz to 350 MHz in 350-MHz steps.

We use a client emulator to exercise the service. The workload consists of a number of concurrent clients that repeatedly open sessions with the service. After opening a session, each client issues a request belonging to that session, receives and parses the reply, waits some time (think-time), and follows a link contained in the reply.

During the first half of the experiment, we impose an offered load of 2400 requests per second, which reverts into a different CPU utilization for each tier of the service. In the absence of emergencies, the first tier exhibits an average utilization of 22% per server, whereas each server of the second tier and the third-tier server have average utilizations around 70% and 80%, respectively. During the second half of the experiment, the offered request rate is 3000 requests per second, imposing average CPU utilizations per machine of 23%, 80%, and 90%, respectively, on the first through third tier in the absence of emergencies. To disturb the system, we inject a thermal emergency at one application server by raising its inlet temperature to $43^{\circ}C$, at time 60 seconds. The emergency lasts until the end of the experiment.

When configuring LiquidN2 and C-LiquidN2, we set the temperature thresholds to the same values as for Freon and C-Freon. With these settings, workload, and emergencies,

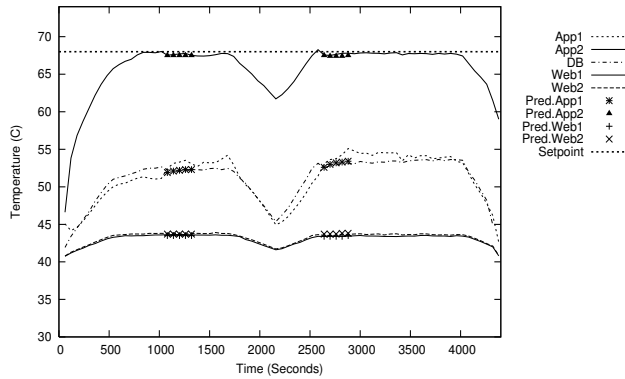


Figure 7. C-LiquidN2 CPU temperatures.

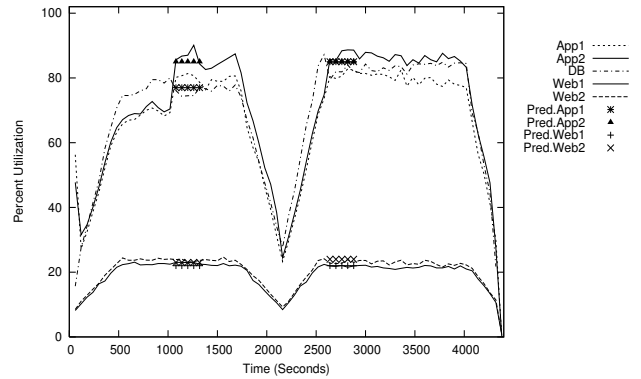


Figure 8. C-LiquidN2 CPU utilizations.

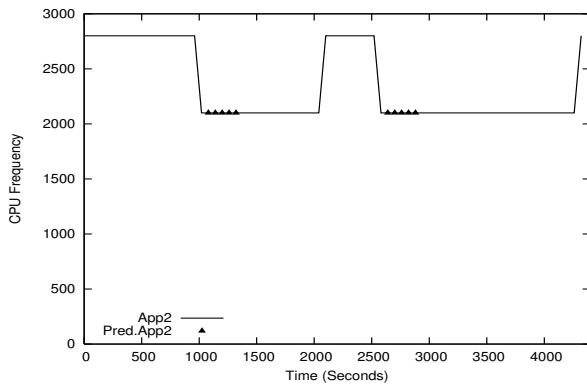


Figure 9. C-LiquidN2 CPU frequencies.

the CPU is the most threatened component so the policies can manage the emergencies it directly. As described in Section 4.3.2, C-LiquidN2 evaluates two reactions, weak and strong, before reacting to a thermal emergency. We again use Mercury for temperature emulation.

Results. Figures 7, 8, and 9 show the results of our C-LiquidN2 experiment. Figures 7 and 8 use the same format as the corresponding figures for C-Freon. Figure 8 depicts CPU utilizations computed with respect to the frequency/voltage setting at each point in time. Figure 9 shows the CPU settings selected by application server 2 (App2) over time, as well as the predicted settings; the other servers remained at the maximum frequency/voltage.

Figure 7 shows that the thermal emergency caused the temperature of the CPU of App2 to rise quickly. When the temperature crossed the high threshold around 1000 seconds, C-LiquidN2 reduced the frequency (and associated voltage) of App2 to 2.1 GHz and restricted its maximum number of concurrent requests, as shown in Figure 9. This adjustment forced some load to be redirected to App1, as Figure 8 illustrates. (Note that the increase in utilization at App2 after it crosses the high threshold is due to the lower frequency/voltage under which it starts to operate, not an in-

crease in the load directed to that server.) Only one adjustment was necessary to keep the temperature under the high threshold. Around 1700 seconds, the offered load started to subside, decreasing the utilizations and temperatures. When the temperature became lower than the low threshold, C-LiquidN2 reset the frequency (and associated voltage) of App2 and eliminated any limits on its number of concurrent requests. Around 2200 seconds, the offered load started to increase again. At that point, the temperature of the CPU of App2 was already high. As Figure 9 shows, when App2 again crossed the high threshold, C-LiquidN2 made a new adjustment to the CPU frequency (and voltage) and admission control. Coincidentally, this adjustment also sent the CPU frequency to 2.1 GHz.

We can make three main observations from these figures. First, we observe that LiquidN2 is an effective thermal management policy for multi-tier services. LiquidN2's use of DVFS enables it to manage the thermal emergencies even without complete control of the load distribution. Second, we observe that C-Oracle again produces temperature and utilization predictions that are accurate. Third, C-LiquidN2 effectively manages the thermal emergencies, while not causing any dropped requests. To manage the first crossing of the high threshold, C-LiquidN2 used the strong reaction, since it would bring the hot server (App2) down to a lower temperature than the weak reaction after 5 minutes. For the second crossing, C-LiquidN2 used the weak reaction, since the strong one would have caused throughput loss by saturating App1 and App2. Again, we do not show figures for LiquidN2 with the weak and strong reactions independently, due to space limitations.

6. Conclusions

In this paper, we proposed and implemented C-Oracle, an infrastructure that allows the pre-evaluation of different thermal management reactions to a thermal emergency, in terms of their future performance and temperature impli-

cations. The thermal management policy can use the predictions to select the best available course of action. We used C-Oracle to implement predictive versions of Freon, a previously proposed thermal management policy, and LiquidN2, a novel policy that we proposed in this paper. Our results showed that C-Oracle produces accurate predictions efficiently. They also demonstrated that the predictive policies can leverage C-Oracle to manage thermal emergencies more effectively than their non-predictive counterparts.

Acknowledgements: We thank Fabio Oliveira for his help with the auction service. We also thank Luiz Barroso, Frank Bellosa, and Yung-Hsiang Lu for comments that helped improve this paper.

References

- [1] D. Anderson, J. Dykes, and E. Riedel. More than an Interface – SCSI vs. ATA. In *Proceedings of FAST*, March 2003.
- [2] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-Driven Energy Accounting for Dynamic Thermal Management. In *Proceedings of COLP*, 2003.
- [3] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proceedings of HPCA*, February 2001.
- [4] J. Chase, D. Anderson, P. Thacker, A. Vahdat, and R. Boyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of SOSP*, October 2001.
- [5] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing Server Energy and Operational Costs in Hosting Centers. In *Proceedings of Sigmetrics*, June 2005.
- [6] J. Choi, Y. Kim, A. Sivasubramaniam, J. Srebric, Q. Wang, and J. Lee. Modeling and Managing Thermal Profiles of Rack-mounted Servers with ThermoStat. In *Proceedings of HPCA*, February 2007.
- [7] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy Conservation Policies for Web Servers. In *Proceedings of USITS*, March 2003.
- [8] Ericsson. Reliability Aspects on Power Supplies. Technical Report Design Note 002, Ericsson Microelectronics, April 2000.
- [9] A. Ferreira, D. Mosse, and J. Oh. Thermal Faults Modeling using a RC model with an Application to Web Farms. In *Proceedings of RTS*, July 2007.
- [10] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. In *Proceedings of ASPLOS*, October 2004.
- [11] S. Gurumurthi, A. Sivasubramaniam, and V. K. Natarajan. Disk Drive Roadmap from the Thermal Perspective: A Case for Dynamic Thermal Management. In *Proceedings of ISCA*, June 2005.
- [12] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and Freon: Temperature Emulation and Management for Server Systems. In *Proceedings of ASPLOS*, October 2006.
- [13] T. Heath, B. Diniz, E. V. Carrera, W. M. Jr., and R. Bianchini. Energy Conservation in Heterogeneous Server Clusters. In *Proceedings of PPOPP*, June 2005.
- [14] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In *Proceedings of MICRO*, December 2000.
- [15] Y. Kim, S. Gurumurthi, and A. Sivasubramaniam. Understanding the Performance-Temperature Interactions in Disk I/O of Server Workloads. In *Proceedings of HPCA*, February 2006.
- [16] Y. Li, D. Brooks, Z. Hu, and K. Skadron. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *Proceedings of HPCA*, February 2005.
- [17] J. Lin, H. Zheng, Z. Zhu, H. David, and Z. Zhang. Thermal Modeling and Management of DRAM Memory Systems. In *Proceedings of ISCA*, June 2007.
- [18] J. Moore, J. S. Chase, and P. Ranganathan. Weatherman: Automated, Online and Predictive Thermal Mapping and Management for Data Centers. In *Proceedings of ICAC*, 2006.
- [19] J. D. Moore, J. S. Chase, P. Ranganathan, and R. K. Sharma. Making Scheduling Cool: Temperature-Aware Workload Placement in Data Centers. In *Proceedings of USENIX*, June 2005.
- [20] T. Mukherjee, Q. Tang, C. Ziesman, and S. K. S. Gupta. Software Architecture for Dynamic Thermal Management in Datacenters. In *Proceedings of COMSWARE*, 2007.
- [21] N. Pettis, J. Ridenour, and Y.-H. Lu. Automatic Run-Time Selection of Power Policies for Operating Systems. In *Proceedings of DATE*, 2006.
- [22] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Dynamic Cluster Reconfiguration for Power and Performance. In L. Benini, M. Kandemir, and J. Ramanujam, editors, *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, August 2003. Earlier version in the Proceedings of COLP, September 2001.
- [23] K. Rajamani and C. Lefurgy. On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters. In *Proceedings of ISPASS*, March 2003.
- [24] P. Ranganathan and N. Jouppi. Enterprise IT Trends and Implications for Architecture Research. In *Proceedings of HPCA*, February 2005.
- [25] Rice University. Dynaserver project, <http://www.cs.rice.edu/cs/systems/dynaserver>. 2003.
- [26] E. Rohou and M. Smith. Dynamically Managing Processor Temperature and Power. In *Proceedings of FDO*, November 1999.
- [27] C. Rusu, A. Ferreira, C. Scordino, A. Watson, R. Melhem, and D. Mosse. Energy-Efficient Real-Time Heterogeneous Server Clusters. In *Proceedings of RTAS*, April 2006.
- [28] L. Shang, L.-S. Peh, A. Kumar, and N. K. Jha. Thermal Modeling, Characterization and Management of On-Chip Networks. In *Proceedings of Micro*, December 2004.
- [29] R. Sharma, C. Bash, C. Pateland, R. Friedrich, and J. Chase. Balance of Power: Dynamic Thermal Management for Internet Data Centers. In *Technical Report HPL-2003-5, HP Labs*, 2003.
- [30] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-Aware Microarchitecture. In *Proceedings of ISCA*, June 2003.
- [31] Uptime Institute. Heat-Density Trends in Data Processing, Computer Systems, and Telecommunications Equipment. Technical report, 2000.
- [32] A. Weissel and F. Bellosa. Dynamic Thermal Management in Distributed Systems. In *Proceedings of TACS*, 2004.