Mercury and Freon: Temperature Emulation and Management for Server Systems *

Taliver Heath
Dept. of Computer Science
Rutgers University
taliver@cs.rutgers.edu

Ana Paula Centeno
Dept. of Computer Science
Rutgers University
anapaula@cs.rutgers.edu

Pradeep George
Dept. of Mechanical Engineering
Rutgers University
pradeepg@rci.rutgers.edu

Luiz Ramos
Dept. of Computer Science
Rutgers University
luramos@cs.rutgers.edu

Yogesh Jaluria
Dept. of Mechanical Engineering
Rutgers University
jaluria@jove.rutgers.edu

Ricardo Bianchini
Dept. of Computer Science
Rutgers University
ricardob@cs.rutgers.edu

Abstract

Power densities have been increasing rapidly at all levels of server systems. To counter the high temperatures resulting from these densities, systems researchers have recently started work on *software-based thermal management*. Unfortunately, research in this new area has been hindered by the limitations imposed by simulators and real measurements. In this paper, we introduce Mercury, a software suite that avoids these limitations by accurately emulating temperatures based on simple layout, hardware, and component-utilization data. Most importantly, Mercury runs the entire software stack natively, enables repeatable experiments, and allows the study of thermal emergencies without harming hardware reliability. We validate Mercury using real measurements and a widely used commercial simulator. We use Mercury to develop Freon, a system that manages thermal emergencies in a server cluster without unnecessary performance degradation. Mercury will soon become available from http://www.darklab.rutgers.edu.

Categories and Subject Descriptors D.4 [*Operating systems*]: Miscellaneous

General Terms Design, experimentation

Keywords Temperature modeling, thermal management, energy conservation, server clusters

1. Introduction

Power densities have been increasing rapidly at all levels of server systems, from individual devices to server enclosures to machine rooms. For example, modern microprocessors, high-performance disk drives, blade server enclosures, and highly populated computer racks exhibit power densities that have never been seen before. These increasing densities are due to increasingly power-hungry hardware components, decreasing form factors, and tighter packing. High power densities entail high temperatures that now must be countered by substantial cooling infrastructures. In fact, when hundreds, sometimes thousands, of these components are racked close together in machine rooms, appropriate cooling becomes the main concern.

The reason for this concern is that 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 mean times between failures (MTBFs) to decrease exponentially [1, 7]. Several factors may cause high temperatures: hot spots at the top sections of computer racks, poor design of the cooling infrastructure or air distribution system, failed fans or air conditioners, accidental overload due to hardware upgrades, or degraded operation during brownouts. We refer to these problems as "thermal emergencies". Some of these emergencies may go undetected for a long time, generating corresponding losses in reliability and, when components eventually fail, performance.

Recognizing this state of affairs, systems researchers have recently started work on *software-based thermal management*. Specifically, researchers from Duke University and Hewlett-Packard have examined temperature-aware workload placement policies for data centers, using modeling and a commercial simulator [21]. Another effort has begun investigating temperature-aware disk-scheduling policies, using thermal models and detailed simulations of disk drives [12, 16]. Rohou and Smith have implemented and experimentally evaluated the throttling of activity-intensive tasks to control processor temperature [26]. Taking a different approach, Weissel and Bellosa have studied the throttling of energy-intensive tasks to control processor temperature in multi-tier services [32].

Despite these early initiatives, the infrastructure for software-based thermal management research severely hampers new efforts. In particular, both real temperature experiments and temperature simulators have several deficiencies. Working with real systems requires heavy instrumentation with (internal or external) sensors collecting temperature information for every hardware component and air space of interest. Hardware sensors with low resolution and poor precision make matters worse. Furthermore, the environment

* This research has been supported by NSF under grant #CCR-0238182 (CAREER award).

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-451-0/06/0010…$5.00

where the experiments take place needs to be isolated from unrelated computations or even trivial "external thermal disruptions", such as somebody opening the door and walking into the machine room. Under these conditions, it is very difficult to produce repeatable experiments. Worst of all, real experiments are inappropriate for studying thermal emergencies. The reason is that repeatedly inducing emergencies to exercise some piece of thermal management code may significantly decrease the reliability of the hardware.

In contrast, temperature simulators do not require instrumentation or environment isolation. Further, several production-quality simulators are available commercially, e.g. Fluent [9]. Unfortunately, these simulators are typically expensive and may take several hours to days to simulate a realistic system. Worst of all, these simulators are not capable of executing applications or any type of systems software; they typically compute steady-state temperatures based on a fixed power consumption for each hardware component. Other simulators, such as HotSpot [30], do execute applications (bypassing the systems software) but only model the processor, rather than the entire system.

To counter these problems, we introduce *Mercury*, a software suite that emulates system and component temperatures in single-node or clustered systems in a repeatable fashion. As inputs, Mercury takes simple heat flow, air flow, and hardware information, as well as dynamic component utilizations. To enable thermal management research, the emulation is performed at a coarse grain and seeks to approximate real temperatures to within a few degrees while providing trend-accurate thermal behavior. Although this is sometimes unnecessary, users can improve accuracy by calibrating the inputs with a few real measurements or short simulations.

Mercury provides the same interface to user-level software as real thermal sensors, so it can be linked with systems software to provide temperature information in real time. In fact, in our approach *the entire software stack runs natively (without noticeable performance degradation) and can make calls to Mercury on-line,* as if it were probing real hardware sensors. Our suite also provides mechanisms for explicitly changing temperatures and other emulation parameters at run time, allowing the simulation of thermal emergencies. For example, one can increase the inlet air temperature midway through a run to mimic the failure of an air conditioner. Finally, Mercury is capable of computing temperatures from component-utilization traces, which allows for fine-tuning of parameters without actually running the system software. In fact, replicating these traces allows Mercury to emulate large cluster installations, even when the user's real system is much smaller.

We validate Mercury using measurements and simulations of a real server. The comparison against Fluent, a widely used commercial simulator, demonstrates that we can approximate steady-state temperatures to within 0.5^oC, after calibrating the inputs provided to Mercury. The comparison against the real measurements shows that dynamic temperature variations are closely emulated by our system as well. We have found emulated temperatures within 1^oC of the running system, again after a short calibration phase.

We are using Mercury to investigate policies for managing thermal emergencies without excessive (throughput) performance degradation. In particular, we are developing *Freon*, a system that manages component temperatures in a server cluster fronted by a load balancer. The main goal of Freon is to manage thermal emergencies without using the traditional approach of turning off the affected servers. Turning servers off may degrade throughput unnecessarily during high-load periods. Given the direct relationship between energy consumption and temperature, as an extension of Freon, we also develop a policy that combines energy conservation and thermal management. Interestingly, the combined policy does turn servers off when this is predicted not to degrade throughput.

To accomplish its goals, Freon constantly monitors temperatures and dynamically adjusts the request distribution policy used by the load balancer in response to thermal emergencies. By manipulating the load balancer decisions, Freon directs less load to hot servers than to other servers. Using our emulation suite, we have been able to completely develop, debug, and evaluate Freon.

In summary, this paper makes two main contributions:

- We propose Mercury, a temperature emulation suite. Mercury simplifies the physical world, trading off a little accuracy (at most 1^oC in our experiments) for native software stack execution, experiment repeatability, and the ability to study thermal emergencies; and
- We propose Freon, a system for managing thermal emergencies without unnecessary performance degradation in a server cluster. In the context of Freon, we propose the first policy to combine energy and thermal management in server clusters.

The remainder of the paper is organized as follows. The next section describes the details of the Mercury suite. Section 3 presents our validation experiments. Section 4 describes the Freon policies. Section 5 demonstrates the use of Mercury in the Freon evaluation. Section 6 overviews the related work. Finally, Section 7 discusses our conclusions and the limitations of our current implementations.

2. The Mercury Suite

In this section we describe the physical model that Mercury emulates, overview the Mercury inputs and emulation approach, and detail our current implementation.

2.1 Physics

Modeling heat transfer and air flow accurately is a complex proposition. To create a very accurate simulation from basic properties, Mechanical Engineering tools must simulate everything from wall roughness to fluctuations of the air density to the detailed properties of the air flow. We believe that this level of detail is unnecessary for most software-based thermal management research, so we have simplified our model of the physical world to a few basic equations. This is the key insight behind Mercury.

Basic terminology. Since many systems researchers are unfamiliar with engineering and physics terminology, we will first define some basic concepts. The *temperature* of an object is a measure of the internal energy present in that object. Temperature is typically measured in degrees Celsius or Kelvin. *Heat* is energy that is transferred between two objects or between an object and its environment. Heat is measured in the same units as energy, such as Joules or Wh.

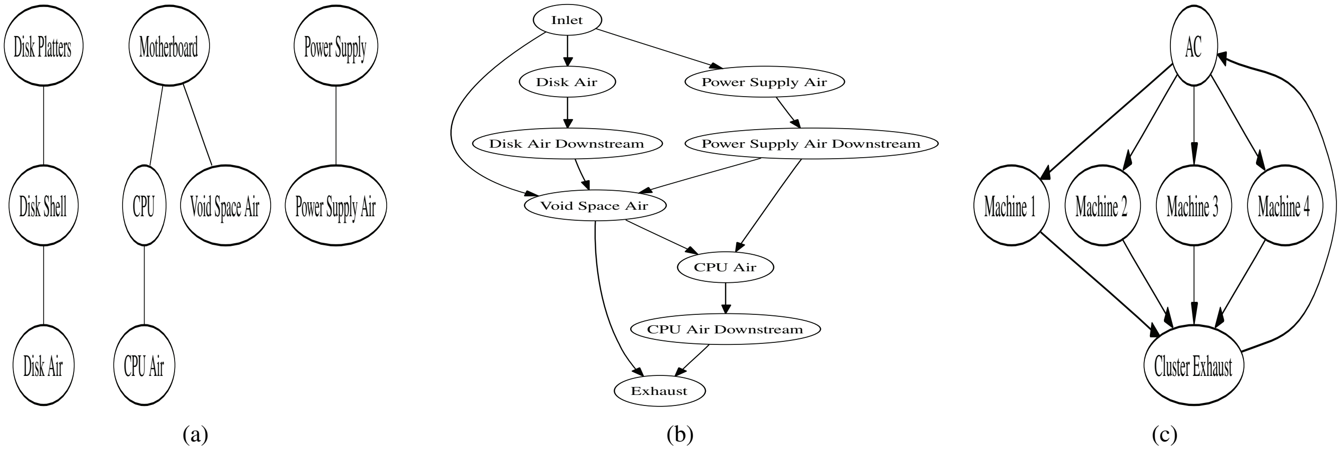
Conservation of energy. One of the most basic equations in heat transfer is the conservation of energy, which translates into the conservation of heat in our case. There are two sources of heat for an object in our system: it converts power into heat while it performs work, and it may gain (lose) heat 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 the component during a period of time and $Q_{component}$ is the amount of heat produced by performing work during a period of time. Next we define these two quantities in turn.

Newton's law of cooling. Heat is transferred in direct proportion to the temperature difference between objects or between an object and its environment. More formally:

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



(a) (b) (c)

Figure 1. Example intra-machine heat-flow (a), intra-machine air-flow (b), and inter-machine air-flow (c) graphs.

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 using typical engineering equations. However, in reality, k can vary with temperature and air-flow rates. We assume that k is constant in our modeling because the extra accuracy of a variable k would not justify the complexity involved in instantiating it for all temperatures and rates.

Energy equivalent. The heat produced by a component essentially corresponds to the energy it consumes, so we define it as:

$$Q_{component} = P(utilization) \times time \tag{3}$$

where $P(utilization)$ represents the average power consumed by the component as a function of its utilization.

For the components that we have studied in detail so far, a simple linear formulation has correctly approximated the real power consumption:

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

where P_{base} is the power consumption when the component is idle and P_{max} is the consumption when the component is fully utilized.

Researchers have used similar high-level formulations to model modern processors and server-style DRAM subsystems [8]. However, our default formulation can be easily replaced by a more sophisticated one for components that do not exhibit a linear relationship between high-level utilization and power consumption. For example, we have an alternate formulation for Q_{CPU} that is based on hardware performance counters. (Although our descriptions and results assume the default formulation, we do discuss this alternate formulation in Section 2.3.)

Heat capacity. Finally, since pressures and volumes in our system are assumed constant, the temperature of an object is directly proportional to its internal energy. More formally, we define the object's temperature variation as:

$$\Delta T = \frac{1}{m\,c} \times \Delta Q \tag{5}$$

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

2.2 Inputs and Temperature Emulation

Mercury takes three groups of inputs: heat- and air-flow graphs describing the layout of the hardware and the air circulation, constants describing the physical properties and the power consumption of the hardware components, and dynamic component utilizations. Next, we describe these groups and how Mercury uses them.

Graphs. At its heart, Mercury is a coarse-grained finite element analyzer. The elements are specified as vertices on a graph, and the edges represent either air-flow or heat-flow properties. More specifically, Mercury uses three input graphs: an inter-component heat-flow graph, an intra-machine air-flow graph, and possibly an inter-machine air-flow graph for clustered systems. The heat-flow graphs are undirected, since the direction of heat flow is solely dependent upon the difference in temperature between each pair of components. The air-flow graphs are directed, since fans physically move air from one point to another in the system.

Figure 1 presents one example of each type of graph. Figure 1(a) shows a heat-flow graph for the real server we use to validate Mercury in this paper. The figure includes vertices for the CPU, the disk, the power supply, and the motherboard, as well as the air around each of these components. As suggested by this figure, Mercury computes a single temperature for an entire hardware component, which again is sufficient for whole-system thermal management research. Figure 1(b) shows the intra-machine air-flow for the same server. In this case, vertices represent air regions, such as inlet air and the air that flows over the CPU. Finally, Figure 1(c) shows an inter-machine air flow graph for a small cluster of four machines. The vertices here represent inlet and outlet air regions for each machine, the cold air coming from the air conditioner, and the hot air to be chilled. The graph represents the ideal situation in which there is no air recirculation across the machines. Recirculation and rack layout effects can also be represented using more complex graphs.

Constants. The discussion above has not detailed how the input graphs' edges are labeled. To label the edges of the heat-flow graph and instantiate the equations described in the previous subsection, Mercury takes as input the heat transfer coefficients, the components' specific heat capacities, and the components' surface areas and masses. The air-flow edges are labeled with the fraction of air that flows from one vertex to another. For example, the air-flow edge from the "Disk Air Downstream" vertex to the "CPU Air" vertex in Figure 1(b) specifies how much of the air that flows over the disk flows over the CPU as well. (Because some modern fans change speeds dynamically, we need a way to adjust the air-flow fractions accordingly. The thermal emergency tool described in Section 2.3 can be used for this purpose.) Besides these constants, the inlet air temperatures, fan speeds, and maximum and base power consumptions for every component must be provided.

There are several ways to find these constants. The heat transfer properties of a component are sometimes provided by the manufacturer (as they are in the case of the Intel CPUs). One may also use a fine-grained engineering tool to determine the heat- and air-flow constants. In some cases, the values may also be found using various engineering equations for forced convection [24].

Determining the constants using any of the previous methods can be time consuming and quite difficult for a computer scientist without a background in Mechanical Engineering. Thus, it 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 emulated readings match the calibration experiment. Since temperature changes are second-order effects on the constants in our system, the constants that result from this process may be relied upon for reasonable changes in temperature ($\Delta T < 40^oC$) for a given air flow [24].

A final set of input constants relate to the power consumption of the different components. In particular, Mercury needs the idle and maximum power consumptions of every component of interest. These values can be determined using microbenchmarks that exercise each component, power measurement infrastructure, and standard fitting techniques. The components' data sheets may sometimes provide this information, although they often exaggerate maximum power consumptions.

Component utilizations. When Mercury is running, it monitors all the component utilizations in the real system, which are used to determine the amount of power that each component is consuming according to Equation 4.

Emulation. Using discrete time-steps, Mercury 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 equations from the previous subsection, the input constants, and the dynamic component utilizations. The air-flow traversals assume a perfect mixing of the air coming from different machines (in case of a cluster), and computing temperatures using a weighted average of the incoming-edge air temperatures and fractions for every air-region vertex. All temperatures can be queried by running applications or the systems software to make thermal management decisions.

2.3 Implementation

The Mercury suite comprises four pieces of software: the solver, a set of component monitoring daemons, a few "sensor" library calls, and a tool for generating thermal emergencies explicitly. Figure 2 overviews the suite and the relationships between its parts.

Solver. The solver is the part of the suite that actually computes temperatures using finite-element analysis. It runs on a separate machine and receives component utilizations from a trace file or from the monitoring daemons. If the utilizations come from a file, i.e. the system is run offline, the end result is another file containing all the usage and temperature information for each component in the system over time. If the utilizations come from the monitoring daemons on-line, the applications or system software can query the solver for temperatures. Regardless of where the utilizations come from, the solver computes temperatures at regular intervals; one iteration per second by default. All objects and air regions start the emulation at a user-defined initial air temperature.

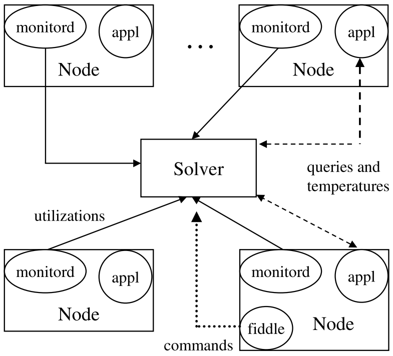


Figure 2. Overview of the Mercury suite.

```
int sd;
float temp;
sd = opensensor("solvermachine", 8367, "disk");
temp = readsensor(sd);
closesensor(sd);
```

Figure 3. An example use of the library calls.

As a point of reference, the solver takes roughly 100 μsec on average to compute each iteration, when using a trace file and the graphs of Figure 1. Because the solver runs on a separate machine, it could execute for a large number of iterations at a time, thereby providing greater accuracy. However, as we demonstrate in Section 3, our default setting is enough to produce temperatures that are accurate to within 1^oC. The thermal emergency tool can force the solver to set temperatures to user-defined values at any time.

The user can specify the input graphs to the solver using our modified version of the language `dot` [10]. Our modifications mainly involved changing its syntax to allow the specification of air fractions, component masses, etc. Besides being simple, the language enables freely available programs to draw the graphs for visualizing the system.

Monitoring daemon. The monitor daemon, called `monitord`, periodically samples the utilization of the components of the machine on which it is running and reports that information to the solver. The components considered are the CPU(s), disk(s), and network interface(s) and their utilization information is computed from /proc. The frequency of utilization updates sent to the solver is a tunable parameter set to 1 second by default. Our current implementation uses 128-byte UDP messages to update the solver.

Sensor library. Applications and system software can use Mercury through a simple runtime library API comprised by three calls: `opensensor()`, `readsensor()`, and `closesensor()`. The call `opensensor()` takes as parameters the address of the machine running the solver, the port number at that machine, and the component of which we want the temperature. It returns a file descriptor that can then be read with `readsensor()`. The read call involves communication with the solver to get the emulated sensor reading. The call `closesensor()` closes the sensor. With this interface, the programmer can treat Mercury as a regular, local sensor device. Figure 3 illustrates the use of these three calls to read the temperature of the local disk.

Our current UDP implementation of `readsensor()` has an average response time of 300 μsec. This result compares favorably with the latency of some real temperature sensors [2]. For instance, the Mercury response time is substantially lower than the average access time of the real thermal sensor in our SCSI disks, 500 μsec.

Thermal emergency tool. To simulate temperature emergencies and other environmental changes, we created a tool called `fiddle`. `Fiddle` can force the solver to change any constant or

```
#!/bin/bash
sleep 100
fiddle machine1 temperature inlet 30
sleep 200
fiddle machine1 temperature inlet 21.6
```

Figure 4. An example `fiddle` script.

temperature on-line. For example, the user can simulate the failure of an air conditioner or the accidental blocking of a machine's air inlet in a data center by explicitly setting high temperatures for some of the machine inlets. Figure 4 shows an example `fiddle` script that simulates a cooling failure by setting the temperature of machine1's inlet air to 30^oC 100 seconds into the experiment. The high inlet temperature remains for 200 seconds, at which point we simulate a restoration of the cooling.

`Fiddle` can also be used to change air-flow or power-consumption information dynamically, allowing us to emulate multi-speed fans and CPU-driven thermal management using voltage/frequency scaling or clock throttling. We discuss this issue in Section 7.

Mercury for modern processors. As we mentioned above, computing the amount of heat produced by a CPU using high-level utilization information may not be adequate for modern processors. For this reason, we have recently developed and validated a version of Mercury for the Intel Pentium 4 processor. In this version, `monitord` monitors the hardware performance counters and translates each observed performance event into an estimated energy [2]. (The software infrastructure for handling the Intel performance counters has been provided by Frank Bellosa's group.) To avoid further modifications to Mercury, the daemon transforms the estimated energy during each interval into an average power, which is then linearly translated into a "low-level utilization" in the $[0\% = P_{base}, 100\% = P_{max}]$ range. This is the utilization reported to the solver.

3. Mercury Validation

We validated Mercury using two types of single-server experiments. The first type evaluates Mercury as a replacement for real measurements, which are difficult to repeat and prevent the study of thermal emergencies. In these validations, we compare the Mercury temperatures to real measurements of benchmarks running through a series of different CPU and disk utilizations.

The second type of validations evaluate Mercury as a replacement for time-consuming simulators, which are typically not capable of executing applications or systems software. In these validations, we compare the Mercury (steady-state) temperatures for a simple server case against those of a 2D simulation of the case at several different, but constant component power consumptions.

3.1 Real Machine Experiments

To validate with a real machine, we used a server with a single Pentium III CPU, 512 MBytes of main memory, and a SCSI 15K-rpm disk. We configured Mercury with the parameters shown in Table 1 and the intra-machine graphs of Figure 1(a) and 1(b). The thermal constants in the table were defined as follows. As in [12], we modeled the disk drive as a disk core and a disk shell (base and cover) around it. Also as in [12], we assumed the specific heat capacity of aluminum for both disk drive components. We did the same for the CPU (plus heat sink), representing the specific heat capacity of the heat sink. We assumed the specific heat capacity of FR4 (Flame Resistant 4) for the motherboard. The k constant for the different heat transfers was defined using calibration, as described below. We estimated the air fractions based on the layout of the components in our server. All the component masses were measured; the CPU was weighed with its heat sink. The disk power consumptions came from the disk's datasheet, whereas we measured the CPU power consumptions. We also measured the power consumption of the motherboard without any removable components (e.g., the CPU and memory chips) and the power supply at 44 Watts. From this number, we estimated the power supply consumption at 40 Watts, representing a 13% efficiency loss with respect to its 300-Watt rating.

Component	Property	Value	Unit
Disk Platters	Mass	0.336	kg
	Specific Heat Capacity	896	$\frac{J}{Kkg}$
	(Min, Max) Power	(9, 14)	Watts
Disk Shell	Mass	0.505	kg
	Specific Heat Capacity	896	$\frac{J}{Kkg}$
CPU	Mass	0.151	kg
	Specific Heat Capacity	896	$\frac{J}{Kkg}$
	(Min, Max) Power	(7, 31)	Watts
Power Supply	Mass	1.643	kg
	Specific Heat Capacity	896	$\frac{J}{Kkg}$
	(Min, Max) Power	(40, 40)	Watts
Motherboard	Mass	0.718	kg
	Specific Heat Capacity	1245	$\frac{J}{Kkg}$
	(Min, Max) Power	(4, 4)	Watts
Inlet	Temperature	21.6	Celsius
	Fan Speed	38.6	ft^3/min

From/To	To/From	k $(Watts/Kelvin)$
Disk Platters	Disk Shell	2.0
Disk Shell	Disk Air	1.9
CPU	CPU Air	0.75
Power Supply	Power Supply Air	4
Motherboard	Void Space Air	10
Motherboard	CPU	0.1

From	To	Air Fraction
Inlet	Disk Air	0.4
Inlet	PS Air	0.5
Inlet	Void Space Air	0.1
Disk Air	Disk Air Downstream	1.0
Disk Air Downstream	Void Space Air	1.0
PS Air	PS Air Downstream	1.0
PS Air Downstream	Void Space Air	0.85
PS Air Downstream	CPU Air	0.15
Void Space Air	CPU Air	0.05
Void Space Air	Exhaust	0.95
CPU Air	CPU Air Downstream	1.0
CPU Air Downstream	Exhaust	1.0

From	To	Air Fraction
AC	Machine 1	0.25
AC	Machine 2	0.25
AC	Machine 3	0.25
AC	Machine 4	0.25
Machine 1	Cluster Exhaust	1.0
Machine 2	Cluster Exhaust	1.0
Machine 3	Cluster Exhaust	1.0
Machine 4	Cluster Exhaust	1.0

Table 1. Constants used in the validation and Freon studies.

We ran calibration experiments with two microbenchmarks. The first set exercises the CPU, putting it through various levels of utilization interspersed with idle periods. The second does the same for the disk. We placed a temperature sensor on top of the CPU heat sink (to measure the temperature of the air heated up by the CPU), and measured the disk temperature using its internal sensor.

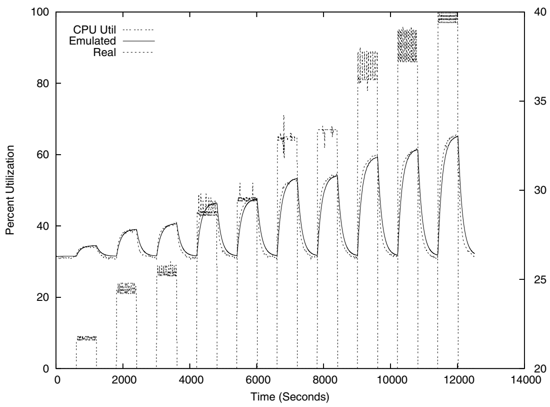


Figure 5. Calibrating Mercury for CPU usage and temperature.

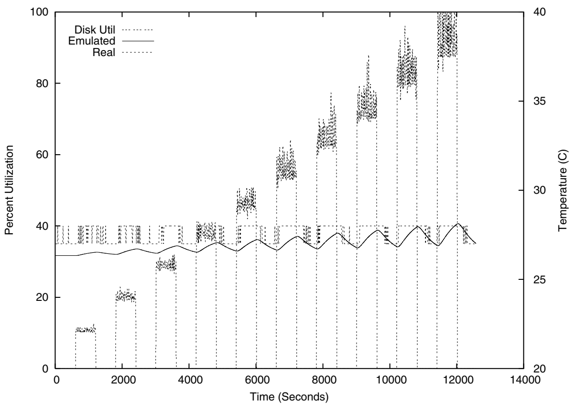


Figure 6. Calibrating Mercury for disk usage and temperature.

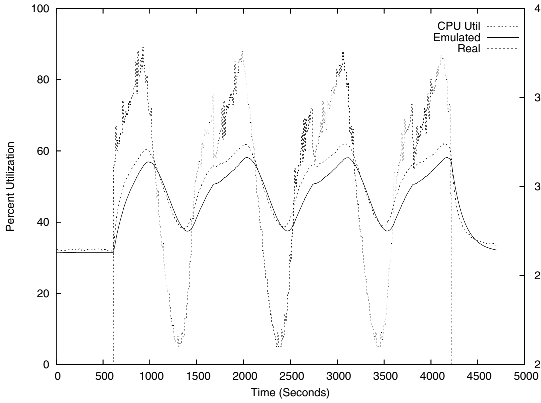


Figure 7. Real-system CPU air validation.

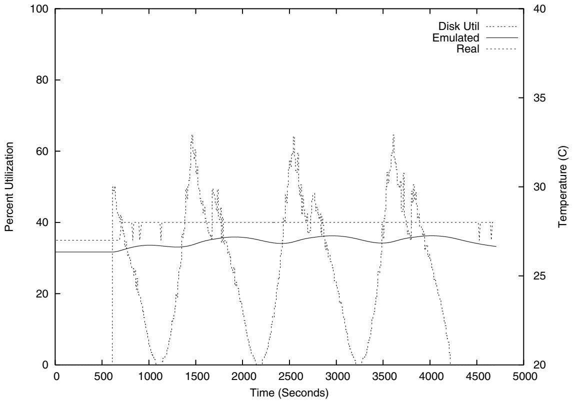


Figure 8. Real-system disk validation.

Figures 5 and 6 illustrate the component utilizations and measured temperatures during the experiments with the CPU and the disk, respectively. The figures also show the Mercury temperatures for the CPU air and the disk, after a phase of input calibrations based on the real measurements. The calibrations were easy to do, taking one of us less than an hour to perform. From these figures, we observe that Mercury is able to compute temperatures that are very close to the measurements for the microbenchmarks.

The calibration allows us to correct inaccuracies in all aspects of the thermal modeling. *After the calibration phase, Mercury can be used without ever collecting more real temperature measurements,* i.e. Mercury will consistently behave as the real system did during the calibration run.

To validate that Mercury is indeed accurate after calibration, we ran a more challenging benchmark without adjusting any input parameters. Figures 7 (CPU air) and 8 (disk) compare Mercury and real temperatures for the benchmark. The benchmark exercises the CPU and disk at the same time, generating widely different utilizations over time. This behavior is difficult to track, since utilizations change constantly and quickly. Despite the challenging benchmark, the validation shows that Mercury is able to emulate temperatures within 1^oC at all times (see the Y-axis on the right of Figures 7 and 8). For comparison, this accuracy is actually better than that of our real digital thermometers (1.5^oC) and in-disk sensors (3^oC).

We have also run validation experiments with variations of this benchmark as well as real applications, such as the Web server and workload used in Section 5. In these (less-challenging) experiments, accuracy was at least as good as in Figures 7 and 8.

3.2 Simulated Machine Experiments

The other main use of the Mercury system is to replace complex simulators. Thus, in this set of validations, we use Fluent, a widely used commercial simulator [9]. The Fluent interface does not allow us to easily vary the power consumption of the simulated CPU, so we compare Mercury and Fluent in steady-state for a variety of fixed power consumptions.

We modeled a 2D description of a server case, with a CPU, a disk, and a power supply. Once the geometry was entered and meshed, Fluent was able to calculate the heat-transfer properties of the material-to-air boundaries. Our calibration of Mercury involved entering these values as input, with a rough approximation of the air flow that was also provided by Fluent. The steady-state temperatures were then found by running Mercury and Fluent for given values of power supply, CPU, and disk power consumptions. Note that here we validated CPU temperatures, rather than CPU *air* temperatures as in the previous subsection. Since the server case was modeled in 2D, no air flows over the simulated CPU.

Our validation results for 14 experiments, each one for a different combination of CPU and disk power consumptions, show that Mercury approximates the Fluent temperatures quite well (within 0.25^oC for the disk and 0.32^oC for the CPU). Note that Mercury achieves this high accuracy despite the fact that Fluent models many hundreds of mesh points, doing a complex analysis of

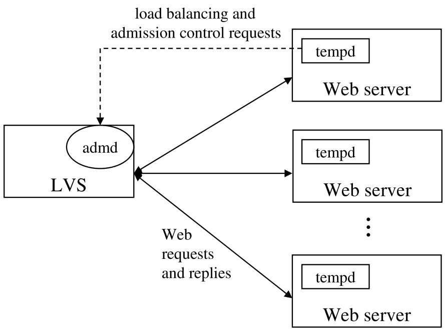


Figure 9. Overview of the Freon system.

temperature and (turbulent) air flows, pressure gradients, boundary conditions, and other effects. For more details on these validations, please refer to our technical report [13].

4. Managing Thermal Emergencies

We are currently using Mercury to study policies and systems for managing thermal emergencies in server clusters. Our research is based on the observation that the traditional approach to dealing with thermal emergencies, i.e. turning the affected servers off, may degrade throughput under high enough load.

Our first effort in this direction is called *Freon*. Freon manages component temperatures in a Web server cluster fronted by a load balancer that uses a request distribution policy based on server weights. The traditional use of weights in server clusters is to account for heterogeneity; roughly speaking, a server that is twice as fast as another should be given a weight that is twice as high. The load balancer we use currently is LVS, a kernel module for Linux, with weighted least-connections request distribution [33].

The main rationale behind Freon's thermal management policy is that periodic temperature monitoring and feedback control can tackle the complex (and often unpredictable) conditions resulting from thermal emergencies. Furthermore, Freon relies on the direct relationship between the rate of client requests directed to a server and the utilization (and resulting temperature) of its hardware components. Finally, for portability reasons, Freon's policy is completely implemented in user-space without modification to legacy codes (LVS, Linux, and server software).

Besides the base version of Freon, we have designed and implemented *Freon-EC*, a system that seeks to conserve energy as well as manage thermal emergencies in server clusters. Energy consumption has become a critical concern for large cluster installations [4, 5, 6, 22], not only because of its associated electricity costs, but also because of the heat to which it translates. Given the direct relationship between energy and temperature, management policies that take both into account are clearly warranted.

Our goal with Freon and Freon-EC is to demonstrate that *relatively simple policies can effectively manage thermal emergencies without unnecessary performance degradation*, even when they also seek to conserve energy. Next, we describe Freon and Freon-EC.

4.1 Freon

As Figure 9 illustrates, Freon is comprised by a couple of communicating daemons and LVS. More specifically, a Freon process, called `tempd` or temperature daemon, at each server monitors the temperature of the CPU(s) and disk(s) of the server. `Tempd` wakes up periodically (once per minute in our experiments) to check component temperatures. The Freon thermal management policy is triggered when `tempd` observes the temperature of any component c at any server to have crossed a pre-set "high threshold", T_h^c. At that point, `tempd` sends a UDP message to a Freon process at the load-balancer node, called `admd` or admission control daemon, 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_h^c. When the temperature of all components becomes lower than their pre-set "low thresholds", T_l^c, `tempd` orders `admd` to eliminate any restrictions on the offered load to the server, since it is now cool and the thermal emergency may have been corrected. 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 a pre-set "red-line threshold", T_r^c. This threshold marks the maximum temperature that the component can reach without serious degradation to its reliability. (Modern CPUs and disks turn themselves off when these temperatures are reached; Freon extends the action to entire servers.) T_h^c should be set just below T_r^c, e.g. 2^oC lower, depending on how quickly the component's temperature is expected to rise in between observations.

Although Freon constantly takes measures for the component temperatures not to reach the red lines, it may be unable to manage temperatures effectively enough in the presence of thermal emergencies that are too extreme or that worsen very quickly. For example, an intense thermal emergency may cause a temperature that is just below T_h^c to increase by more than $T_r^c - T_h^c$ in one minute (Freon's default monitoring period). Further, it is possible that the thermal emergency is so serious that even lowering the utilization of the component to 0 would not be enough to prevent red-lining. For example, suppose that the inlet area of a server is completely blocked. However, note that any thermal management system, regardless of how effectively it manages resource states and workloads, would have to resort to turning servers off (and potentially degrading throughput) under such extreme emergencies.

Details. 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 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 we only run the controller when the temperature of a component is higher than T_h^c and force *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* (this requires accounting for the weights of all servers). Because LVS directs requests to the server i with the lowest ratio of active connections and weight, $min(Conns_i/Weight_i)$, the reduction in weight naturally shifts load away from the hot server.

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 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 about this statistic.

By changing weights and limiting concurrent requests, Freon dynamically moves load away from hot servers, increasing the load on other servers as a result. If the emergency affects all servers

```
while (1) {
  receive utilization and temperature info from tempd;

  if (need to add a server) and (at least one server is off)
    in round-robin fashion, select a region that (a) has at
      least one server that is off, and (b) preferably is
      not under an emergency;
    turn on a server from the selected region;

  if (temperature of $c$ just became $> T_h^c$)
    increment count of emergencies in region;
    if (all servers in the cluster need to be active)
      apply Freon's base thermal policy;
    else
      if (cannot remove a server)
        turn on a server chosen like above;
      turn off the hot server;
  else
    if (temperature of $c$ just became $< T_l^c$)
      decrement count of emergencies in region;
      apply Freon's base thermal policy;

  if (can still remove servers)
    turn off as many servers as possible in increasing
      order of current processing capacity;
}
```

Figure 10. Pseudo-code of admd in Freon-EC.

or the unaffected servers cannot handle the entire offered load, requests are unavoidably lost.

Finally, note that, under light load, Freon could completely exclude a hot server from the load distribution, allowing it to cool fast. However, this could make it difficult for Freon to control temperatures smoothly when the server eventually needs to be reincluded to service a moderate or high offered load. (This is equivalent to the problem of overloading newly added nodes in least-connection scheduling.) Despite this difficulty, Freon-EC does turn nodes off under light enough load to conserve energy, as we shall see next.

4.2 Freon-EC

Freon-EC combines energy conservation and thermal management. This combination is interesting since, for best results, energy must play a part in thermal management decisions and vice-versa. To conserve energy, Freon-EC turns off as many servers as it can without degrading performance, as in [4, 6, 14, 22, 25, 27]. However, unlike these and all other previous works in cluster reconfiguration, Freon-EC selects the servers that should be on/off according to their temperatures and physical locations in the room. In particular, Freon-EC associates each server with a physical "region" of the room. We define the regions such that common thermal emergencies will likely affect all servers of a region. For example, an intuitive scheme for a room with two air conditioners would create two regions, one for each half of the servers closest to an air conditioner. The failure of an air conditioner would most strongly affect the servers in its associated region.

With respect to thermal management, Freon-EC uses Freon's base policy when all servers are needed to avoid degrading performance. When this is not the case, Freon-EC actually turns hot servers off, replacing them with servers that are potentially unaffected by the emergency, i.e. from different regions, if possible.

Details. Freon-EC has the same structure as Freon. Further, in Freon-EC, tempd still sends feedback controller outputs to admd when a component's temperature exceeds T_h^c. However, tempd also sends utilization information to admd periodically (once per minute in our experiments). With this information, admd implements the pseudo-code in Figure 10.

Turning off a server involves instructing LVS to stop using the server, waiting for its current connections to terminate, and then shutting it down. Turning on a server involves booting it up, waiting for it to be ready to accept connections, and instructing LVS to start using the server. Because turning on a server takes quite some time, Freon-EC projects the current utilization of each component (averaged across all servers in the cluster) into the future, when the overall load on the cluster appears to be increasing. Specifically, Freon-EC projects utilizations two observation intervals into the future, assuming that load will increase linearly until then.

The decision of whether to add or remove servers from the active cluster configuration is based on these projected component utilizations. We add a server when the projected utilization of any component is higher than a threshold, U_h (70% in our experiments). Looking at the current (rather than projected) utilization, Freon-EC removes servers when the removal would still leave the average utilization of all components lower than another threshold, U_l (60% in our experiments). Pinheiro *et al.* [22] used a similar approach to cluster reconfiguration. Recently, Chen *et al.* [5] proposed a more sophisticated approach to cluster reconfiguration that considers multiple costs (not just energy costs) and response times (instead of throughput). Furthermore, practical deployments have to consider the location and management of servers that should not be turned off if possible, such as database servers. These considerations can be incorporated into Freon-EC.

4.3 Freon vs CPU Thermal Management

It is important to compare Freon with thermal management techniques for locally controlling the temperature of the CPU, such as throttling or scaling voltage/frequency [2, 3, 26, 30]. In a server cluster with least-connections load balancing, these techniques may produce a load distribution effect similar to Freon's. Next, we describe the advantages of the Freon approach.

Interestingly, Freon does implement a form of throttling by reducing the amount of load directed to hot servers and, as a result, reducing their CPU utilizations. Because throttling is performed by the load balancer rather than the hot CPUs themselves, we call this approach "remote throttling". The main advantages of remote throttling are that it does not require hardware or operating system support and it allows the throttling of other components besides the CPU, such as disks and network interfaces.

Voltage/frequency scaling is effective at controlling temperature for CPU-bound computations. However, scaling voltage/frequency down slows the processing of interrupts, which can severely degrade the throughput achievable by the server. In addition, CPUs typically support only a limited set of voltages and frequencies. Remote throttling is more flexible; it can generate a wide spectrum of CPU power consumptions from idle to maximum power (albeit at the maximum voltage/frequency). Moreover, unlike remote throttling, scaling requires hardware and possibly software support and does not apply to components other than the CPU.

Finally, high-level thermal management systems can leverage information about the workload, the temperature of multiple hardware components (as in Freon and Freon-EC), the intensity of the current and projected load on the cluster (as in Freon-EC), and the physical location of different servers (as in Freon-EC). So far, we have not leveraged the characteristics of the workload. For example, in the face of a hot CPU, the system could distribute requests in such a way that only memory or I/O-bound requests were sent to it. Lower weights and connection limits would only be used if this strategy did not reduce the CPU temperature enough. The current version of Freon does not implement this two-stage policy because LVS does not support content-aware request distribution. We will consider content-aware load balancers in our future work.

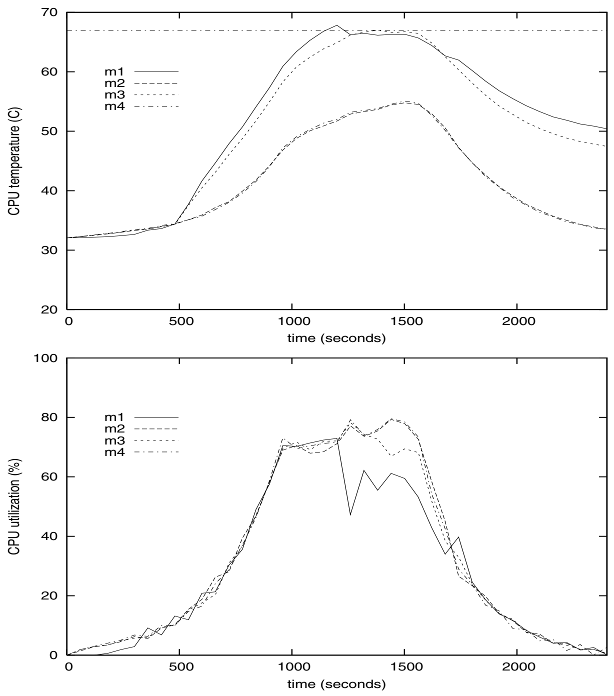


Figure 11. Freon: CPU temperatures (top) and utilizations (bottom).

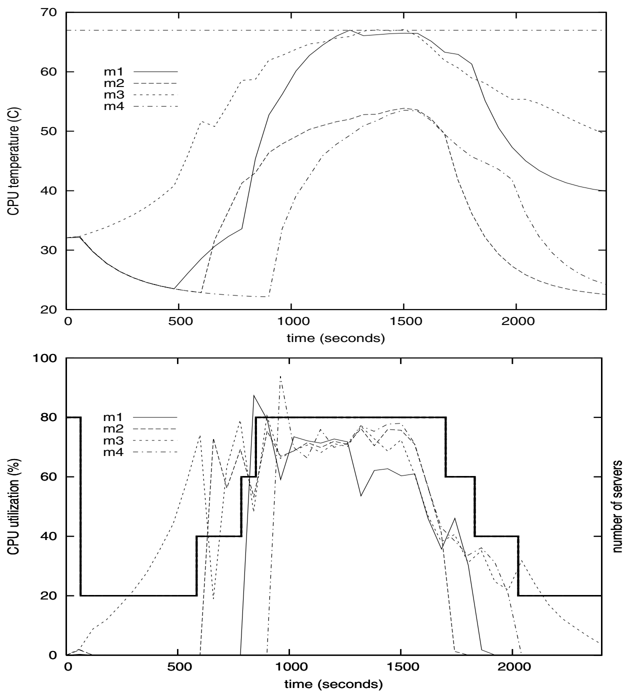


Figure 12. Freon-EC: CPU temperatures (top) and utilizations (bottom).

Despite these benefits, *the best approach to thermal management in server clusters should probably be a combination of software and hardware techniques*; the software being responsible for the higher-level, coarser-grained tasks and the hardware being responsible for fine-grained, immediate-reaction, low-level tasks. The exact form of this combination is also a topic of our future research.

5. Freon Results

In this section, we demonstrate the use of Mercury in evaluating the Freon policies. Specifically, we ran Freon experiments with 4 Apache servers behind an LVS load balancer. The servers contained Pentium III processors and 15K-rpm SCSI disks, whereas the load balancer is a Pentium 4-based machine. Mercury was deployed on the server nodes and its solver ran on yet another Pentium III machine. The inputs to Mercury were those of Table 1. We set $T_h^{CPU} = 67^oC$, $T_l^{CPU} = 64^oC$, $T_h^{disk} = 65^oC$, and $T_l^{disk} = 62^oC$, which are the proper values for our components.

To load the servers, we used a synthetic trace, as we could not find a publicly available Web server trace including dynamic content requests. Our trace includes 30% of requests to dynamic content in the form of a simple CGI script that computes for 25 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.

5.1 Base Policy Results

The results of Freon's base policy are shown in Figure 11. Since the CPU is the most highly utilized component in our experiments, Figure 11 includes graphs of the CPU temperatures (top graph) and CPU utilizations (bottom graph), as a function of time. The horizontal line in the top graph marks T_h^{CPU}. Each point in the utilization graph represents the average utilization over one minute.

From the temperature graph, we can see that all CPUs heat up normally for the first several hundred seconds. At 480 seconds, `fiddle` raised the inlet temperature of machine 1 to 38.6^oC and machine 3 to 35.6^oC. (The emergencies are set to last the entire experiment.) The emergencies caused the affected CPU temperatures to rise quickly. When the temperature of the CPU at machine 1 crossed T_h^{CPU} at 1200 seconds, 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 of the workload than machine 1, as the CPU-utilization graph illustrates. As a result of the extra load and the higher inlet temperature, the temperature of the CPU at machine 3 crossed T_h^{CPU} at 1380 seconds. This shifted even more load to machines 2 and 4 *without affecting machine 1*. Around 1500 seconds, the offered load started to subside, decreasing the utilizations and temperatures.

Throughout this experiment, Freon was able to manage temperatures and workloads smoothly. Further, the fact that Freon kept the temperature of the CPUs affected by the thermal emergencies just under T_h^{CPU} demonstrates its ability to manage temperatures with as little potential throughput degradation as possible. In fact, Freon was able to serve the entire workload without dropping requests.

For comparison, we also ran an experiment assuming the traditional approach to handling emergencies, i.e. we turned servers off when the temperature of their CPUs crossed T_r^{CPU}. Because $T_r^{CPU} > T_h^{CPU}$, machine 1 was turned off at time 1440 seconds, whereas machine 3 was turned off just before 1500 seconds. The system was able to continue handling the entire workload after the loss of machine 1, but not after the loss of machine 3. Overall, the traditional system dropped 14% of the requests in our trace.

5.2 Freon-EC Results

The results of Freon-EC for the same trace and thermal events are also positive. In these experiments, we grouped machines 1 and

3 in region 0 and the others in region 1. Figure 12 shows these results. Note that the CPU utilization graph (bottom of Figure 12) also includes a thick line illustrating the number of server nodes in the active cluster configuration, as a function of time.

The CPU utilization graph clearly shows how effectively Freon-EC conserves energy. During the periods of light load, Freon-EC is capable of reducing the active configuration to a single server, as it did at 60 seconds. During the time the machines were off, they cooled down substantially (by about 10^oC in the case of machine 4). As the load increased, Freon-EC responded by quickly enlarging the active configuration up to the maximum number of servers, without dropping any requests in the process. As aforementioned, turning on a machine causes its CPU utilization (and temperature) to spike quickly, an effect that we avoid in the base Freon system. At the peak load, the CPU temperatures of machines 1 and 3 again crossed T_h^{CPU}, being handled correctly by the base thermal policy. This time, machine 3 required two load-distribution adjustments. During the latter part of the experiment, the decrease in load allowed the active configuration to again be reduced to one server.

6. Related Work

As far as we know, Mercury is the first temperature emulation suite for systems research. Other computer system temperature studies (both in Engineering and Computer Science) have either relied on real measurements or detailed simulations. Next, we discuss related works that have used these two approaches. We also overview the related thermal management works.

Simulators. As we mentioned before, the Engineering fields use detailed simulators extensively [9, 15]. Because these simulators typically focus on steady-state conditions, they do not actually execute applications or systems software.

Weatherman [20] also simulates steady-state conditions, but does so significantly faster than the Engineering simulators. Specifically, Weatherman computes the temperature of the inlet air at all servers in a data center using a neural network. Although training the network can be seen as an automatic approach to calibration, it requires extensive instrumentation (i.e. inlet sensors at all servers) or numerous detailed simulations to provide training data.

A few simulators have been developed in the computer architecture community for studying thermal management policies in microprocessor design [3, 30]. These simulators typically execute one application at a time directly on a simplified CPU, i.e. they do not model multiprogramming and bypass the systems software.

Real measurements. Like Weatherman, ConSil [19] uses a neural network to infer the temperature at the servers' inlets in a data center. ConSil's inputs are the component utilizations and readings of motherboard thermal sensors at each server. For network training, readings of the inlet sensors at a number of servers are also used. ConSil requires extensive instrumentation (i.e. inlet sensors at a large number of servers physically spread across the data center and motherboard sensors at all servers) and seems inappropriate for studying thermal emergencies.

Bellosa's group [2, 18, 32] also studied temperature using real measurements. However, their real measurements are not of temperature, but rather of processor performance counters. Their approach avoids the repeatability (and high overhead [2]) problems of using real thermal sensors. However, the limitations of their approach are that (1) it only applies to estimating the temperature of CPUs (and only those that provide the required performance counters), and (2) it relies on a complex, case-by-case extrapolation from the values of the counters to the energy the CPU consumes. Mercury can emulate the temperature of all components and important air spaces, doing so in a simple and general way.

Software-based thermal management. Because temperature issues only recently became an overwhelming concern for systems software researchers, very few papers have been written in this area to date. Gurumurthi et al. [12] have considered temperature-aware disk-scheduling policies using modeling. More recently, the same group [16] studied the disk temperature implications of real I/O workloads in simulation. Also using simulation, Powell et al. [23] studied thread scheduling and migration in chip multiprocessors with simultaneous multithreading cores. Unlike our work, these studies focused on single-component scenarios and did not actually implement their proposed policies in real software systems.

In contrast, researchers have experimentally studied the throttling of tasks to control CPU temperature in single-node systems [2, 26] and multi-tier services [32]. Unfortunately, they did not consider request redistribution away from hot servers (thermal emergencies apparently reduce the incoming client traffic in their experiments) or components other than the CPU.

In the context of data centers, Sharma et al. [29] and Moore et al. [20, 21] studied temperature-aware workload placement policies under normal operating conditions, using modeling and a commercial temperature simulator; no application or system software was actually executed. Further, the main goal of thermal management policies for normal operation (reducing cooling costs) and thermal emergencies (preventing reliability degradations) differ substantially. Sharma et al. [29] proposed the notion of regions (borrowed in Freon-EC) and a policy for keeping servers on during thermal emergencies, but did not discuss a real implementation. Finally, these works treat each server as black box with a fixed power consumption or utilization, without explicitly considering the dynamic behaviors of their different internal components.

Architecture-level thermal management. As temperature is a critical concern for modern CPUs, several researchers have considered thermal management at the architectural level in simulation, e.g. [3, 17, 28, 30, 31]. The high level of detail of these simulators is probably unnecessary for most software systems research.

Furthermore, although the power density of other components is not increasing as fast as that of CPUs, these components also need thermal management. For example, the power density of disk drives is proportional to their rotational speed squared [11], which has been increasing substantially recently and needs to continue to increase [12]. To make matters worse, these components are usually densely packed in groups to try to keep up with CPU speeds. Disk array and blade server enclosures are good examples of dense packing of multiple types of components.

7. Conclusions

In this paper, we proposed and validated Mercury, a temperature emulation suite for research on software-based thermal management. *We believe that Mercury has the potential to allow research on temperature issues to flourish in the systems community.* For this reason, we will make the Mercury source code available soon from http://www.darklab.rutgers.edu. Using Mercury, we developed and evaluated Freon, a system for managing thermal emergencies in server clusters without unnecessary performance degradation.

Limitations and future work. Our experience with Mercury to date has been very good. The system is easy to use and has saved us substantial hassle in studying Freon. Despite these benefits, the current version of Mercury needs a few extensions and additional evaluations. In particular, we are currently extending our models to consider clock throttling and variable-speed fans. Modeling throttling and variable-speed fans is actually fairly simple, since these behaviors are well-defined and essentially depend on temperature, which Mercury emulates accurately albeit at a coarse grain. In fact, these behaviors can be incorporated either internally (by modifying the Mercury code) or externally (via fiddle). We also plan to study the emulation of chip multiprocessors, which will probably have to be done in two levels, for each core and the entire chip.

Our real implementation of Freon demonstrated that relatively simple policies and systems can effectively manage thermal emergencies. However, the current version of Freon also needs a few extensions. In particular, Freon needs to be extended to deal with multi-tier services and to include other policies. We are currently addressing these extensions, as well as considering ways to combine Freon with CPU-driven thermal management.

Acknowledgements

We would like to thank Frank Bellosa, Enrique V. Carrera, Steve Dropsho, Cosmin Rusu, and the anonymous reviewers for comments that helped improve this paper significantly. We are also indebted to Frank Bellosa, Andreas Merkel, and Simon Kellner for sharing their performance-counter infrastructure with us. Finally, we thank Enrique V. Carrera and Diego Nogueira who contributed to the early stages of this work.

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, S. Kellner, M. Waitz, and A. Weissel. Event-Driven Energy Accounting of Dynamic Thermal Management. In *Proceedings of COLP*, September 2003.

[3] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proceedings of HPCA*, January 2001.

[4] J. Chase, D. Anderson, P. Thackar, 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] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy-Efficient Server Clusters. In *Proceedings of PACS*, February 2002.

[7] Ericsson Microelectronics. Reliability Aspects on Power Supplies. Technical Report Design Note 002, April 2000.

[8] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A Performance-Conserving Approach for Reducing Peak Power Consumption in Server Systems. In *Proceedings of ICS*, June 2005.

[9] Fluent. Fluent: The Right Answer in CFD. http://www.fluent.com/-index.htm.

[10] E. Gansner, E. Koutsofios, and S. North. Drawing Graphs with Dot. http://www.graphviz.org/Documentation/dotguide.pdf.

[11] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proceedings of ISCA*, June 2003.

[12] S. Gurumurthi, A. Sivasubramaniam, and V. Natarajan. Disk Drive Roadmap from the Thermal Perspective: A Case for Dynamic Thermal Management. In *Proceedings of ISCA*, June 2005.

[13] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and Freon: Temperature Emulation and Management for Server Systems. Technical Report DCS-TR-596, Department of Computer Science, Rutgers University, January 2006, Revised July 2006.

[14] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr., and R. Bianchini. Energy Conservation in Heterogeneous Server Clusters. In *Proceedings of PPoPP*, June 2005.

[15] T. Icoz and Y. Jaluria. Numerical Simulation of Boundary Conditions and the Onset of Instability in Natural Convection Due to Protruding Thermal Sources in an Open Rectangular Channel. *Numerical Heat Transfer*, 48:831–847, 2005.

[16] 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.

[17] 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.

[18] A. Merkel and F. Bellosa. Balancing Power Consumption in Multiprocessor Systems. In *Proceedings of Eurosys*, April 2006.

[19] J. Moore, J. Chase, and P. Ranganathan. ConSil: Low-Cost Thermal Mapping of Data Centers. In *Proceedings of SysML*, June 2006.

[20] J. Moore, J. Chase, and P. Ranganathan. Weatherman: Automated, Online, and Predictive Thermal Mapping and Management for Data Centers. In *Proceedings of ICAC*, June 2006.

[21] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making Scheduling Cool: Temperature-Aware Resource Assignment in Data Centers. In *Proceedings of USENIX*, April 2005.

[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 published in Proceedings of COLP 2001, September 2001.

[23] M. Powell, M. Gomaa, 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.

[24] F. K. Price. *Heat Transfer and Fluid Flow Data Books*. Genium Publishing Corp., 1990.

[25] K. Rajamani and C. Lefurgy. On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters. In *Proceedings of ISPASS*, March 2003.

[26] E. Rohou and M. D. 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. Jha. Characterization and Management of On-Chip Networks. In *Proceedings of Micro*, December 2004.

[29] R. Sharma, C. Bash, C. Patel, R. Friedrich, and J. Chase. Balance of Power: Dynamic Thermal Management for Internet Data Centers. *IEEE Internet Computing*, 9(1), January 2005.

[30] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-Aware Microarchitecture. In *Proceedings of ISCA*, June 2003.

[31] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of ISCA*, June 2004.

[32] A. Weissel and F. Bellosa. Dynamic Thermal Management for Distributed Systems. In *Proceedings of TACS*, June 2004.

[33] W. Zhang. Linux Virtual Server for Scalable Network Services. In *Proceedings of the Linux Symposium*, July 2000.