Predictive Thermal Management for Hard Real-Time Tasks

Albert Mo Kim Cheng and Chen Feng
Real-Time System Laboratory, Department of Computer Science
University of Houston, Houston, TX 77204, USA
{cheng, scfeng}@cs.uh.edu

Abstract

Dynamic thermal management (DTM) techniques help a system to operate within a safe range of temperature by reducing the performance of the CPU dynamically when the system is too hot. Dynamic voltage scaling (DVS) and localized toggling are both DTM techniques. DVS is easier to use for real-time systems since the performance degradation can be controlled accurately so that tasks are still meeting their deadlines. Localized toggling changes architectural configurations of a CPU to a less optimized setting. Its performance degradation is harder to measure and to control accurately. In this paper, we propose a method which applies various localized toggling techniques to real-time systems while still able to meet task deadlines. Our method activates DTM when the temperature over the execution of a job is predicted to be too high at the start time. When DTM is activated, our method measures the performance degradation of different toggling techniques during the slack time to select the most effective technique and still be able to meet deadline. We use instructions per cycle (IPC) as the performance measure. Our method is evaluated on the SimpleScalar CPU simulator with Wattch, the energy simulator, and HotSpot, the thermal model.

1. Why Thermal Management?

Thermal management maintains the temperature of a system to be below a predefined trigger temperature. Although cooling systems such as CPU fans, heat sinks, and packaging that help airflow are for the same purpose, they are often designed for the worst-case workload. Modern CPUs have increasingly faster clock rates and consume more power and they produce more heat. [Gunther et al. 2001] have shown that the cost of the cooling system will increase dramatically if they have to keep up with the worst-case workload. Thermal management can guarantee the temperature of the system by managing the workload so that cooling systems can be designed for an average workload, thus reducing cost.

2. Triggering and Response Techniques

Thermal management involves triggering and response techniques. Triggering technique decides when to invoke the response technique and when to turn it off. It constantly samples the actual temperature of the system. There is usually a predefined trigger temperature which is well below the temperature that can cause the hardware to malfunction. A triggering technique can be either reactive or predictive. A reactive technique invokes the response technique at the moment when the actual temperature becomes higher than the trigger temperature. A predictive technique predicts when the actual temperature might become higher then the trigger temperature, and invokes the response technique early as a preventive method. It has been shown in [Srinivasan 2003] that a predictive technique has the advantage that more time is available for the response technique to react to the trigger so that slower response methods such as DVS (higher overhead compared to other techniques) can be used. Our method is predictive.

A CPU can be viewed as a combination of various components (i.e., registry file, instruction pipeline, caches, functional units, DVS, etc.). A set of parameters and values for all components in the CPU is an *architectural configuration*. A response technique sets the CPU to a lower-performance architectural configuration with the goal of reducing temperature. There are three classes of response techniques: throttling the whole CPU, throttling parts of the CPU, and DVS. Throttling of the whole CPU stops the fetching of the instructions until the temperature is lowered [Brooks 2001]. Throttling parts of the CPU turns off various optimizations. Some methods which have been tried in this class include: turning off the branch prediction [Brooks 2001], speculative execution, instruction cache [Sanchez 1997], data cache, functional units [Skadron 2003] and part of the registry file, reducing the length of the instruction pipeline, and decreasing the instruction parallelism. DVS reduces CPU voltage which in turn reduces heat dissipation and clock rate [Hughes 2001].

3. The Problem

Reducing system temperature by managing workload usually results in performance degradation. Real-time systems require controlled performance degradation so that task deadlines can still be met with a reduced performance. DVS can be used in a controlled way while guaranteeing task deadline as demonstrated in various energy-saving papers. However, many researches show that DVS cannot effectively lower system temperature because it decreases the temperature of the whole CPU instead of targeting at a thermally-hot spot. DVS' long latency also makes it unsuitable for a fast response to a thermal crisis. Throttling the whole CPU has less latency; its effect on performance degradation

can be measured accurately. However, it still suffers from not targeting the thermally-hot spot. Throttling parts of the CPU can target a thermally-hot spot, thus, it can decrease system temperature more effectively than DVS and complete throttling. However, the performance degradation of this technique is harder to measure and to control. This makes the whole class of techniques less suitable in a real-time environment where tasks must meet their deadlines.

4. Contributions

CPU optimizations such as instruction pipelining and out-of-order issuing along with compiler optimizations allow modern CPUs to be capable of executing more than one instruction per CPU clock cycle. The variation of IPCs (Instructions Per Cycle) throughout the execution of a program is often used to measure CPU workload and code performance [Ghiasi 2000, Gunther et al 2001]

This research has the following contributions. First, we present a method which uses the IPC measurement in a time-window to control the performance degradation of throttling techniques. We also show a way to find the thermally-hot spot during the slack-time. Combining these two techniques gives us a way to manage system temperature effectively while guaranteeing task deadlines. Second, we present a predictive algorithm that can trigger the response technique before the thermal crisis takes place. Since we have ample time to respond, a slow method such as DVS can also be used. Last, we evaluate our approach on a multi-tasking real-time CPU simulator. Researches that we know of only evaluate their techniques with non-real-time single-task CPUs [Srinivasan 2003].

5. Definitions

Given a set of n periodic tasks T = {T1 … Tn}. Let i = 1 … n. WCET(Ti) is the relative worst-case execution time of task Ti. deadline(Ti) is the smaller of the relative deadline and the period of task Ti. Each task is partitioned into a number of blocks so Ti = (B1 U B2 U … U Bm). Let j = 1 … m. WCET(Bj) is the relative worst-case execution time of block Bj. A hyper-period is the least-common-multiple (LCM) of the periods of all the tasks so hyper-period = LCM(period(Ti)) for i = 1 … n. The schedules are the same in different hyper-periods for periodic tasks. At execution-time, a task Ti is divided into a number of jobs in each hyper-period. Ji,k indicates the kth instance of the task Ti in a hyper-period.

6. Our Method

Our method involves an off-line process and an on-line process. The off-line process partitions a task into blocks and measures the maximum temperature increase and the WCET of each block. The on-line process schedules tasks using EDF. Before the start of each block, our method predicts the highest temperature that might be reached during the execution of the block. If the prediction is higher than the trigger temperature, then we need to activate thermal control throughout the execution of the block. The thermal control uses the slack time that is available for the block to evaluate different response techniques. Each response technique is called an architectural configuration. We evaluate the performance of each architectural configuration to select one to use for the remaining execution of the block. In order to guarantee that the task can meet its deadline, we execute the block in a reduced performance for a limited amount of time such that if the remaining work is executed with the maximum performance, then the block will be able to finish within its WCET. If all blocks of a job finish within their WCETs, the job will meet its deadline.

The off-line process partitions a task into blocks of instructions. Neighboring blocks having the same average IPC should be combined into one block. We measure WCET and the maximum temperature increase for each block. The on-line process is an event-driven process implemented at the OS level. This process relies on the CPU to provide performance values including the total number of instructions executed since the start of the system and the total number of cycles elapsed since the start of the system. We also need to know the current CPU temperature provided by a sensor. Here is a list of global values and their meanings.

Name	Meaning	Where does it come from
Slacks	Maintains a set of slack times and expirations of the slack times.	Our code.
Window_time	The number of CPU clock cycles per time window.	This is a constant which should be set to be less than the shortest WCET of blocks.
Inst_count	The total number of instructions executed since the system start.	This value should be maintained by the CPU.
cycle_count	The total number of CPU clock cycles elapsed since the system start.	This value should be maintained by the CPU.
Curr_temp	The current CPU temperature in degrees.	This value comes from the sensor which is attached to CPU.
trigger_temp	The trigger CPU temperature in degrees. Thermal control activates when the current CPU temperature is higher than the trigger CPU temperature.	This is constant which should be set lower than the maximum temperature which the CPU can work without flaw.

The on-line process first computes the schedule using EDF. It then executes the schedule. Throughout the execution of the schedule the following events may happen:

Block-Start Event: triggers at the start time of a block. We predict the maximum temperature and activate thermal control if necessary.

Time-Window-Start Event: triggers at the start of a time window. If the thermal control is active and the block is executing in the slack time, then we pick an architectural configuration to evaluate for the current time window.

Before-Preemption Event: triggers before a block is preempted by another block. We need to prepare the transferring of the slack time and prepare some information for resuming the block once the higher priority job completes.

Before-Resume Event: triggers before a block resumes after the higher priority job completes. We basically do the same as in Time-Window-Start. We need to predict the maximum temperature again even if thermal control was active before the block got preempted. Since it is possible that the temperature has been lowered by the blocks of the higher priority job, it is possible that we do not need to activate thermal control again.

Time-Window-End Event: triggers at the end of a time window. If thermal control is active, then we need to measure IPC for the architectural configuration that we are evaluating. If the IPC of the architectural configuration is one that we can use for the rest of the execution of the block and still be able to finish before the WCET of the block, then we can pick the architectural configuration.

Block-Complete Event: triggers at the completion of a block. If the block finishes before its WCET, then we can transfer some slack time for the future blocks of the current task or other tasks.

Job-Complete Event: triggers at the completion of a job. If the job finishes before its period (deadline) then we can transfer some slack time for the future blocks of other tasks.

In each event we need to maintain some block specific information. We define the following variables.

WCET(b) Worst-case execution time of block b. This is measured off-line.

IPC(b) Average IPC for block b without any performance loses. This is measured off-line.

remaining_inst_count(b) The number of remaining instructions for block b in the worst-case.

remaining_time(b) Remaining execution time in number of CPU clock cycles for the execution of block b in the worst-case.

max_temp_inc(b) Maximum temperature increase, in degrees, throughout the execution of block b. This is measured off-line.

thermal_ctrl(b) A Boolean value which indicates if thermal control is active (true) or inactive (false).

candidate_arch_config(b) An integer which is no less than -1. This is an index into the arch_configs list (see arch_config(b,i)). This is the candidate architectural configuration that we are currently evaluating the performance of.

arch_configs(b, i) A list of architectural configurations for block b in the order of increasing potential performance slowdown. i is an integer no less then -1 which indexes the list of architectural configurations. For all blocks, arch_config(b, -1) is the configuration of no performance slowdown.

picked_arch_config(b) An integer which is no less than -1. This is an index into the arch_configs list (see arch_config(b,i)). This is the architectural configuration that we have picked to be used to control the temperature.

start_inst_count(b) This is the inst_count value at the beginning of a time window for block b. At the end of the time window we use this value to calculate an average IPC for block b at the current window.

window_start_time(b) This is the cycle_count value at the beginning of a time window for block b.

arch_config_max(b) This is the upper-bound of the index for arch_configs, candidate_arch_config, and picked_arch_config.

slack(b) This is the slack time available for block b in number of cycles.

IPC_picked_arch_config(b) This is the IPC for the architectural configuration picked_arch_config.

The following is the pseudo-code for our event-driven on-line process. % indicates comments.

```
%at the start of the system
schedule <- EDF_scheduler(T)
%schedule tasks T with EDF
execute(schedule) %execute schedule

%at the start of a block b at time t
event Block-Start(b, t)
   slack(b) <- available slack time
               for b in number of
               cycles
   remaining_inst_count(b) <- WCET(b)*
```

```
                                IPC(b)
   remaining_time(b) <- WCET(b)
%decide if thermal control for b is
needed
if max_temp_inc(b) + curr_temp >
   trigger_temp then
      thermal_ctrl(b) <- true
      candidate_arch_config(b) <- -1
      picked_arch_config(b) <- -1
      IPC_picked_arch_config(b) <-
         IPC(b)
end event

%at each time window of the execution
of a block b at time t
event Time-Window-Start(b, t)
   start_inst_count(b) <- inst_count
   window_start_time(b) <- t
   if thermal_ctrl(b) then
      if slack(b) > 0 then
      % try the next one
      candidate_arch_config(b) <-
      candidate_arch_config(b) + 1
      % if we run out of config to try
      and still didn't find the best
      % one, then try from beginning
         if candidate_arch_config(b) =
            arch_config_count(b) &&
            picked_arch_config(b) = -1
               then
         candidate_arch_config(b) <- 0
         % if there are more config to
         evaluate so try the next one
         if candidate_arch_config(b) <
            arch_config_count(b) - 1
            then
            set CPU to use
            candidate_arch_config(b)
            configuration
         else % no more slack for b
        if picked_arch_config(b) != -1
            then
            % need to decide if we
            % need to turn-off thermal
            % control to catch up with
            % the deadline
        if remaining_time(b) <
           remaining_inst_count(b) /
           ipc_picked_arch_config(b)
            then
           picked_arch_config(b) <- -1
           set CPU to use
           configuration -1
           %the fastest possible
           % configuration
            thermal_ctrl(b) <- false
        else % no good architectural
           configuration to use
           thermal_ctrl(b) <- false
end event

%before a block b is preempted by
another block c at time t
event Before-Preemption(b, c, t)
% before the start of block c (the
higher priority block)
remaining_inst_count(b) <-
   remaining_inst_count(b) -
      (inst_count - start_inst_count(b))
% decrease slack time
   slack(b) <- slack(b) - (t -
      window_start_time(b))
% decrease remaining time if there are
no more slack left
if slack(b) < 0 then
   remaining_time(b) <-
      remaining_time(b) + slack(b)
   slack(b) <- 0
end event

%before a block b resumes after block
c finishes at time t
event Before-Resume(b, c, t)
% after the end of block c (the higher
priority block)
slack(b) <- available slack time for b
in cycles
% decide if we need to do thermal
control for b (c might have lowered
% the temperature)
if max_temp_inc(b) + curr_temp >
   trigger_temp then
   thermal_ctrl(b) <- true
   % we need to redo all evaluation
   since c might have heated up
   % different parts of CPU
   candidate_arch_config(b) <- -1
   picked_arch_config(b) <- -1
else % prediction no longer higher
   than trigger temperature
   thermal_ctrl(b) <- false
%invoke start time window event for b
Time-Window-Start(b, t)
end event

%at the end of a time window of the
execution of a block b
event Time-Window-End(b, t)
if thermal_ctrl(b) then
% check if we can pick the current
config under evaluation
if candidate_arch_config(b) != -1 then
IPC <- (inst_count -
start_inst_count(b)) / time_window
solve for t: remaining_inst_count =
IPC * t + IPC(b) *
(remaining_time(b) - t)
if t > 0 then
   picked_arch_config(b) <-
   candidate_arch_config(b)
   ipc_picked_arch_config(b) <- IPC
   % decrease slack time by 1
   time_window
   slack(b) <- slack(b) - time_window
   % decrease remaining time if there
   is no more slack left
if slack(b) < 0 then
   remaining_time(b) <-
      remaining_time(b) + slack(b)
   slack(b) <- 0
   % keep track of how many
   instructions have been executed
   remaining_inst_count(b) <-
      remaining_inst_count(b) -
      (inst_count -
       start_inst_count(b))
```

```
end event

%at the completion of a block b at
time t
event Block-Complete(b, t)
% decrease slack time
slack(b) <- slack(b) - (t
window_start_time(b))
% decrease remaining time if there are
no more slack left
if slack(b) < 0 then
   remaining_time(b) <-
      remaining_time(b) + slack(b)
   slack(b) <- 0
   % keep the slack of this block for
   intra or inter task use
if WCET(b) - t > 0 then
   s <- create_slack()
   length(s) <- WCET(b) - t
   expire(s) <- period(job(b)) % this
      slack time expires at the
      % deadline of the job of block b
      register_slack(s, slacks)
end event
%at the completion of job j at time t
event Job-Complete(j, t)
% keep the slack time for this job for
inter-task use
if period(j) - t > 0 then
   s <- create_slack()
   length(s) <- period(j) - t
   expire(s) <- period(j)
   register_slack(s)
end event
```

7. A Thermal Control Example

We use an example to illustrate the main ideas of the above algorithm.

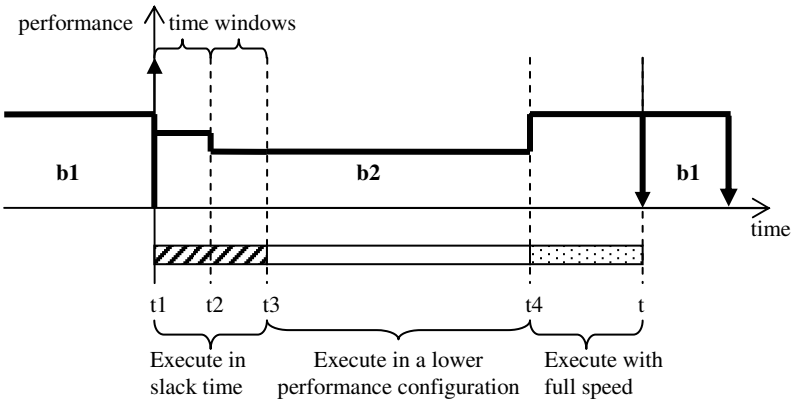

Figure 1. A thermal control

Figure 1 shows an example of thermal control. In this example, b1 and b2 are blocks for tasks T1 and T2, respectively, and they belong to jobs J1 and J2, respectively. The priority of T2 is higher than the priority of T1 so period(T2) < period(T1) and deadline(J2) < deadline(J1); therefore, J2 preempts J1. For simplicity, b2 is the only block in T2. At time t1, b2 starts and preempts b1. During the time interval [t1, t3), b2 evaluates different architectural configurations in the slack time (will be explained). During the time interval [t3, t4), b2 executes the selected architectural configuration at a reduced performance for thermal control. During the time interval [t4, t5), b2 turns off thermal control and executes using the fastest configuration in order to complete in WCET(b2) and meet the deadline at t5. At t5, b1 resumes execution. At t6, b1 completes.

At time t1, the executing block b1 is preempted by a higher priority block b2. Since t1 is the start time of block b2, our method predicts the highest temperature that could be reached during b2's execution. In this example, the predicted highest temperature is above the trigger temperature so we activate thermal control for b2 at time t1. Since J2 preempts J1, there might be some slack time which comes from earlier blocks (excluding b1) of J1 that completes before their WCETs. This slack is transferred for use by b2. The order of events at t1 is Before-Preemption(b1, b2, t1) followed by Block-Start(b2, t1) followed by Time-Window-Start(b2, t1).

b2 uses time windows [t1, t2), and [t2, t3) to evaluate different architectural configurations. For example, at time t1, Time-Window-Start(b2, t1) is triggered to set the CPU to arch_config(b2, 0) (the 1st configuration) for the execution of b2 in [t1, t2). At time t2, Time-Window-End(b2, t2) is triggered. The IPC of the arch_config(b2, 0) is checked to see that if this IPC is used during [t3, t5), then is it possible to find a point t4 in [t3, t5) such that if we switch to the fastest configuration at t4 we can finish the rest of the execution before t5? Notice that some work of b2 has been completed during [t1, t2) so we only need to consider the remaining work in this calculation. The following equation describes this relationship:

```
remaining_inst_count(b) = IPC * t +
IPC(b) * (remaining_time(b) - t)
```

Here, remaining_inst_count (b) is the remaining work of block b, IPC is the IPC of the architectural configuration that we evaluated, IPC(b) is the IPC when

block b executes without any thermal control, remaining_time(b) is the worst-case execution time of block b, and t is the length of time that we want to execute the rest of block b in a lower performance configuration. With all values given except t we are trying to solve for t. If t is a possible number then it tells us that t4 = t3 + t and we can execute b2 using this architectural configuration for [t3, t4) and still meets the deadline at t5. The same set of events triggers for the time interval [t2, t3), namely: Time-Window-Start(b2, t2), Time-Window-End(b2, t3).

Time-Window-Start(b2, t3) is triggered after Time-Window-End(b2, t3). Here we notice that there is no more slack time left for evaluating architectural configurations, so we must pick one to use for the remaining execution of b2. Hopefully we have picked a configuration. If all of the configurations that we evaluated cannot provide a t4 that helps b2 to meet the deadline, then we run the rest of b2 with no thermal control. There is still a chance that the remaining of b2 might not cause the temperature to go above the trigger temperature for two reasons: first, our prediction was not accurate, and second, the lower performance configurations that we used during [t1, t3) have already lowered the temperature. At time t5, Block-Complete(b2, t5) is triggered. If b2 completes before its WCET then we can transfer some time to be used by future inter-task blocks. Job-Complete(J2, t5) is triggered following Block-Complete(b2, t5) since our example assumes that b2 is the only block of J2. If J2 completes before its deadline, then we can transfer some time to be used by future inter-task blocks. Events Block-Resume(b1, t5) and Time-Window-Start(b1, t5) are triggered afterwards to continue the execution of b1.

8. Evaluation

Since it is still not clear to us how to automate the partition of tasks we plan to handcraft a few tasks which are partitioned nicely for the off-line process. We plan to evaluate the on-line process of our method on a CPU simulator with benchmark applications running in real-time. We choose to integrate the Watcch CPU simulator with the HotSpot Thermal Model. HotSpot is the current state-of-the-art thermal model for CPU. It is able to output simulated temperature readings on different parts of the CPU floor-plan and the sink. Wattch is a modified version of the SimpleScalar CPU simulator with energy usage simulation. The per-cycle energy usage values are used as inputs into HotSpot to create the temperature values.

Wattch supports CPU configurations that are close to the current state-of-the-art CPU architectures (such as pipelining and out-of-order execution). However, it executes one program at a time from start to finish and there is currently no OS ported to it yet. This makes it impossible to study multi-program interaction on Wattch. SIMCA - The SImulator for Multi-threaded Computer Architectures is a multithreaded version of the SimpleScalar simulators, implemented by the ARCTiC Group(http://www.mount.ee.umn.edu/~lilja/SIMCA/index.html). However, there is no energy simulator for it. Therefore, we choose to modify Wattch with the ability to do both time-sharing multitasking and EDF scheduling.

MiBench is a free benchmark suite for embedded applications maintained by the University of Michigan [Guthaus 2001]. We have successfully modified Wattch to produce IPC plots for MiBench programs including qsort and susan (an image processing program). We have difficulty on the temperature plot at this time. The temperature plot we get does not fluctuate with the workload, which is not what we have expected. Our modified Wattch can perform time-sharing multitasking on certain applications. Due to some memory addressing issue, it fails on some combination of programs and large programs. We suspect that it is due to the data in the cache being fetched into the wrong program. Thus we have not started the implementation of the real-time scheduler. However, we have planned how to do it and it should be straightforward once we fix the multitasking bug. Once our simulator is ready to produce per-cycle IPC as well as temperature readings and is able to execute programs in real-time, we will evaluate our method and report the results in an upcoming full paper.

Acknowledgments

This material is supported in part by a grant from the Institute for Space Systems Operations.

References

[Gunther 2001] S. H. Gunther et al. Managing the impact of increasing microprocessor power consumption. Intel Technology Journal, 1st Quarter, 2001.

[Srinivasan 2003] J. Srinivasan and S. V. Adve. Predictive dynamic thermal management for multimedia applications. Proc. 17th ACM Intl. Conf. Supercomputing, June 2003

[Sanchez 1997] H. Sanchez et al. Thermal management system for high performance PowerPC microprocessors. Proc. IEEE Compcon'97 Digest of Papers, February 1997.

[Brooks 2001] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors, 2001.

[Hughes 2001] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications, 2001.

[Ghiasi 2000] S. Ghiasi et al, Using IPC variation in workloads with externally specified rates to reduce power consumption. Proc. Workshop on Complexity-Effective Design, June 2000.

[Guthaus 2001] M. R. Guthaus et al, MiBench: A free, commercially representative embedded benchmark suite, 2001. Available at http://www.eecs.umich.edu/jringenb/mibench/.

[Skadron 2003] K. Skadron et al, Temperature-Aware Microarchitecture, in ISCA, June 2003.