Implementing a Thermal-Aware Scheduler in Linux Kernel on a Multi-Core Processor

LIANG XIA¹, YONGXIN ZHU¹, JUN YANG², JINGWEI YE¹ AND ZONGHUA GU³*

¹School of Microelectronics, Shanghai Jiao Tong University, People’s Republic of China
²Department of Electrical and Computer Engineering, University of Pittsburgh, PA, USA
³College of Computer Science, Zhejiang University, People’s Republic of China
*Corresponding author: zggu@zju.edu.cn

As power dissipation causes thermal issues in cooling costs, lifetime and reliability, thermal management has become an important issue in today’s OS and processor design. Early OS-level thermal management schemes were proposed and evaluated mainly with simulators or analytical models. In this paper, we implement a thermal-aware round-robin scheduling algorithm in the Linux kernel, and compare its performance with the ‘Heat-and-Run’ algorithm and the default Linux baseline scheduler on an Intel Core 2 Duo processor using representative benchmarks from SPEC2000, MiBench and NetBench. Our results indicate that the current Linux scheduler can easily be enhanced with thermal-awareness to show improved performance in terms of both the on-chip temperature condition and application throughput.

Keywords: thermal-aware; multicore; scheduling

Received 12 July 2009; revised 1 November 2009
Handling editor: Sung Woo Chung

1. INTRODUCTION

The ever increasing power consumption of today’s microprocessors has become a limiting factor to achievable chip performance. This is particularly constraining in multi-core processors (a.k.a. chip multiprocessor (CMP)), as low power cores typically imply simpler designs which lead to lower single-thread performance. A critical consequence of high power dissipation is the thermal management among different on-chip cores. High operating temperature not only induces high leakage current, but also impairs the reliability, lifetime and the cooling cost of the processors. Hence, proper thermal management schemes are important for maintaining healthy operation of a processor. A widely used dynamic thermal management technique is the hardware-level dynamic voltage and frequency scaling (DVFS), which down-scales the voltage and frequency of the chip when an embedded thermal sensor detects a temperature overshooting a safe threshold. We will demonstrate in this paper that a smarter OS-level management can improve the thermal conditions on-chip and alleviate the pressure of performance loss due to hardware DVFS.

In this paper, we carry out thermal-aware thread scheduling experiments to evaluate our OS-level scheduler for multiple threads (more than core count) on a physical Intel Core 2 Duo processor. We implemented a simple Round Robin (RR)-based scheduling algorithm in the Linux 2.6 kernel that supports multiple cores. We take 20 representative benchmarks from SPEC2K, MiBench and NetBench to evaluate the implementation of our algorithm and show that classic timing-aware only scheduling algorithms often miss tight thermal constraints on multi-core processors. Intuitive thermal-aware scheduling algorithms such as the Heat-and-Run (HR) algorithm also produce worse thermal behavior than our simple algorithm with careful implementation. In our actual implementation of the HR algorithm, we improve it by setting different temperature thresholds for dual cores, due to process variations.

The remainder of the paper is organized as follows. Section 2 discusses most relevant work to our proposal. Section 3 presents the concept of RR Algorithm. The implementation of thermal-aware RR Algorithm in Linux kernel is proposed in Section 4. Section 5 describes HR Algorithm [1] to compare...
with. Section 6 illustrates our experiment setup and analyzes the final experimental results. Finally, our conclusion is presented in Section 7.

2. RELATED WORK

There have been several proposals for OS-level thermal management. The main approach is to leverage the discrepancies of the temperature variations among different threads, and schedule them to keep the chip temperature low. For example, on single-core processors, Kumar et al. [2] proposed to manage the temperature by restricting the CPU time quota of hot jobs if temperature is in an alarm region. Yang et al. [3] presented a heuristic scheduling algorithm to reduce the triggering of DVFSs. On multi-core processors, Gomma et al. [1] proposed an HR scheme to assign and migrate task threads across multiple cores; Donald and Martonosi [4] improve processor throughput by combining various DVFS policies and thread migration policies.

More recently, deep nanometer semiconductor process brings significant process variations that produce within-die differences in performance and thermal behavior of multiple cores. Teodorescu and Torrellas [5] proposed variation-aware scheduling and power management DVFS algorithms to save power or improve throughput. All these algorithms were evaluated in a simulation environment or with an analytical model. Kursun and Cher [6] proposed a technique that utilizes the existing on-chip sensor infrastructure to improve the inherent thermal imbalances among different cores in a multicore architecture. Their experimental analysis on a test chip shows that the efficiency of thermal management techniques such as activity migration and thermal-aware scheduling can be improved through such variation-awareness. Choi et al. [7] performed experiments of thermal-aware task scheduling on a Power5 dual-core processor. However, only simple core hopping between two threads were evaluated, and the algorithms do not consider the case where the number of threads is larger than the number of cores (which is more likely the case in a real system), so that multiple threads need to share one core.

There are also many works on OS-level task scheduling with the objective of achieving a higher throughput while holding a relatively low core temperature. Musoll et al. [8] also studied two load balancing schemes including RR and lower-index first on their event-driven simulator. Isci et al. [9] evaluated several different policies for global multi-core power management with different objectives, such as prioritization, fairness and maximal throughput in their hierarchical feedback-control-based framework. Stavrou and Trancoso [10] studied how to assign tasks onto a CMP with a large number of cores (around 64). This static task assignment takes into consideration of both the core temperature and the heat generation of neighborhood cores to achieve the maximal efficiency which is inversely proportional to the number of thermal violations. Mulas et al. [11] expressly designed and implemented thermal-aware balancing policy for an MPSoC running streaming applications, in which meeting deadline is a critical constraint. Their lightweight thermal balancing policy can attain a low core temperature with small run-time migration cost. Merkel and Bellosa [12] developed and evaluated their energy-aware scheduling algorithm on a multiprocessor system running Linux OS.

Additionally, Chrobak et al. [13] proposed both static offline and dynamic on-line scheduling techniques for real-time jobs to satisfy their deadlines under a given thermal threshold. Zhang and Chatha [14] also formulated static offline temperature-aware scheduling as an integral optimization problem with nonlinear continuous constraint. They solved their formulations by dynamic programming and a new fully polynomial time approximation scheme. Moreover, Refs. [15–17] presented library/simulator/profiler that support task migration on multiprocessors.

3. BASIC RR ALGORITHM

The RR algorithm is a simple and widely used scheduling algorithm. All tasks are kept in one ready queue. The CPU scheduler picks the first task and allocates it one time slice. When a task uses up its time slice, it is moved to the tail of the queue, and then the scheduler chooses the next task to run. All time slices have the same length, so that all tasks can occupy the CPU equally until they finish. On a multi-core system, every task can run on one of the cores for an equal time slice. Figure 1 shows an example execution trace.

4. THERMAL-AWARE RR IMPLEMENTATION IN LINUX KERNEL

We implemented our scheduling algorithm on Intel Core 2 Duo E4500 @ 2.2 GHz running Fedora Core 7. The Linux kernel version is 2.6.21. In this section, we first discuss the Linux-2.6 scheduler and its deficiency, and then enable temperature monitoring in Linux kernel. Finally, we will explain our modifications in detail.

4.1. Linux scheduler and its deficiency

The Linux OS scheduler is designed to meet scheduling goals such as efficiency, interactivity and fairness. Each core has a runqueue, which has two priority arrays called ‘active array’ and ‘expired array’. The active array keeps active tasks whereas the expired array holds expired tasks. The scheduler assigns a
time slice to every task, the default value of which is 100 ms. The scheduler always chooses the first task in the active array to run. After the task uses up its time slice, it is put to the expired array, and the scheduler selects the next task on the active array to run. If the active array contains no tasks, it will become the expired array. At the same time, the expired array becomes the active array.

In single CPU systems, Linux scheduler picks tasks of the same priority in an RR way to meet the fairness goal and prevent starvation. For multi-CPU systems, Linux scheduler supports SMP scheduling, when it schedules tasks across multiple CPUs on the same motherboard. The goal of the scheduling algorithm is to make the best use of processor time. Thus, major consideration of choosing the target CPU for a task to run on is cache, scheduling a given task on the same CPU as often as possible. This is known as affinity scheduling. Because of this, load balancer in Linux scheduler does not constantly move a task from one CPU to another, unless it finds the CPUs in the same group far from balanced. Therefore, a task is more likely to stay on one CPU for its entire execution time.

Tasks usually have different thermal intensity, as CPU-intensive tasks are considered hot tasks while memory-intensive tasks are considered cool ones. If such a ‘hot’ task is scheduled to occupy one processor core for a sufficiently long time, the temperature of the core will reach a threshold. At this time, it is necessary for the scheduler to migrate this hot task to another core. Unfortunately, when the workload is balanced in a group of CPUs, the load balancer will not migrate any tasks, which probably causes temperature imbalance. What’s more, load balancer is not able to move a currently running task until it becomes an awaiting task in the runqueue. Consequently, Linux scheduler lacks the ability to schedule tasks according to their thermal intensity, so that we have some room to improve its balancing policy, adding thermal-awareness into the scheduler.

4.2. Enabling temperature monitoring

We add a Core Temp [18] patch to the kernel to get temperature information of both cores through digital thermal sensor (DTS) located in each individual processing core. The accuracy of DTS is one degree. In Linux kernel, an OS timer interrupt called scheduler tick interrupts every millisecond. Choi et al. [7] observe that rise time and fall time of temperatures of cores are in the order of hundreds of milliseconds. Therefore, we can schedule a hot task in the scheduler tick to cooler cores in time, avoiding high temperature on a single core. We sample the DTS on Intel Core 2 Duo every 10 ms in the scheduler tick to catch the temperature.

4.3. Thermal-aware RR algorithm on dual-core processor

To balance temperature in a group of CPUs and avoid peak temperature, our idea is to schedule a task to each CPU for equal amount of time periodically once the temperature on any core exceeds the predefined threshold. Our thermal-aware RR algorithm is described in Algorithm 1. The algorithm can be viewed as two parts. One is when number of tasks is more than two, the other is when task number is no more than two.

Algorithm 1 N-Task 2-Core RR algorithm.

```plaintext
1: Input: task number n=N
2: while scheduler tick at every 10ms do
3: if (any core exceeds temperature threshold) then
4: if n==0 then
5: break;
6: else if n==1 then
7: Task T executes for time slice;
8: Swap in idle thread
9: Migrate T to the other core;
10: else if n==2 then
11: Two tasks run on each core for time slice;
12: Swap in idle threads
13: Swap the tasks;
14: else
15: T0 = Core0.current_task;
16: T1 = Core1.current_task;
17: if T0 is swapped out (on current core) then
18: Migrate T0 to Core1 (i.e. next core);
19: T0 = Core0.current_task;
20: end if
21: if T1 is swapped out (on current core) then
22: Migrate T1 to Core0 (i.e. next core);
23: T1 = Core1.current_task;
24: end if
25: end if
26: end if
27: end while
```

When number of tasks is more than number of cores (i.e. \( n > 2 \)), task switch happens on the core that bears multiple tasks. We modified the timer interrupt handler code, adding the above algorithm into it. Two task pointers point to current tasks of each core. When the handler finds that the pointed task is context switched out, it migrates the task to the other core. Since Linux default scheduler assigns 100 ms time slice to each user level task, we set a check point at every 10 ms in the OS scheduler tick. The execution time of each task on each core is 100 ms.

When number of tasks are no more than number of cores, as is mentioned previously, load balancer is not capable of moving a currently running task, our RR algorithm includes a special procedure for task migration. A flag is added into per-core runqueue. When the flag is set, it triggers context switch, the special procedure, which swaps out current task and swaps in idle thread on each core, before the task is migrated to the other core. Since swapping in idle thread brings extra overhead
in every task migration, we do not include this procedure in the actual implementation and regard that RR ends when there are less than three tasks in the dual core system. However, we describe this situation to complete the whole picture.

At present, many of the multi-CPU systems are CMP architectures, which have multiple cores on a single chip. A super speed bus connects the cores and most cores have shared caches. For instance, there is a shared L2 cache on Intel Core 2 Duo. Therefore, task migration can easily be performed among closely coupled cores and incurs few cache miss.

To verify our RR scheduling algorithm without losing generality, we consider a taskset with three tasks, named Task A, Task B and Task C, and assume that the cores have thermal violation. Figure 1 shows RR scheduling as a state machine with six states forming a circle in the state space. For example, in the first state, Task A occupies Core0, Task B occupies Core1 and Task C is waiting on Core0. In this figure, we see that each task executes on each core for equal length of time, thus forming RR scheduling.

### 4.4. Extending thermal-aware RR scheduling to quad-core processor

Having defined the N-Task 2-Core RR Algorithm, we may further define 4-Core RR algorithm. The key concept is when the temperature on any core exceeds predefined threshold, all the tasks on multip-core system will be rotated from one core to a next core so that each task, hot, warm or cool, has equal chance to run on any one of the cores to dissipate almost the same amount of heat, thus, the temperature on any core will not be too high. When number of tasks are more than number of cores, migration happens after the context switch by Linux default scheduler, and when tasks are less than cores, the swap-in-idle-thread mechanism enables the migration of a task running alone from one core to a next core.

Figures 3–6 describe the thermal-aware RR scheduling on Quad Core when number of tasks are no more than number of cores. The gray-shaded box (Core 0) on the upper left corner of the quad-core processor figure denotes the core with a thermal violation, i.e. its temperature exceeds the temperature threshold. If multiple cores have a thermal violation simultaneously, then any one among them can be chosen as Core 0. 1-Task and 4-Task RR scheduling are easy to understand, while we may encounter different situations in 2-Task and 3-Task RR scheduling. For example, tasks may be running on neighboring cores or on opposite cores. In Fig. 4a, the clockwise RR scheduling can easily force Core 0 to be idle in order to lower its temperature, whereas the counter-clockwise RR scheduling cannot have this function. On the other hand, counter-clockwise RR scheduling will have effect if Task A is on Core 3. Consequently, RR scheduling should have its direction in the actual implementation to remove the thermal intensity more quickly, thus lower the temperature of the core that has a temperature violation. In Fig. 5, one more step before real RR scheduling is added that if the core of which the temperature exceeds the threshold happens to be the mid-core in the RR scheduling direction, the task on the core will be first migrated to the opposite idle core, and then the normal RR takes effect. This mechanism removes the thermal intensity quickly.

Figure 7 describes the situation that tasks are more than cores. Similar to Algorithm 1, the RR scheduling here will migrate the task after it is context switched by Linux default scheduler. For instance, when Task A is switched out, it is migrated to the
Implementing a Thermal-aware Scheduler in Linux Kernel on a Multi-core Processor

next core (Core 1) and becomes the awaiting task. Meanwhile, the original awaiting task (Task E) becomes the running task on Core 0. The last state in this figure is just the state that has one step in front of the first state. Besides, the idea is the same in 6-Task or more RR scheduling. The actual implementation of thermal-aware RR scheduling on Quad Core will be the future work.

5. HR ALGORITHM

The HR algorithm-based [1] scheduler migrates tasks away from the overheated core and assigns it to the nonheated core. A temperature threshold is defined and migration happens when the temperature of one core is above the threshold and the temperature of the other core is much lower than the threshold. We have some modifications when implementing this algorithm in our Linux kernel. HR does not differentiate between hot and cool tasks, so it can migrate any task while one core is overheated. We argue that migrating hot task is better for cooling CPU in a short time than migrating an arbitrary task. We observe that the performance of the cores on our machine is not exactly the same. For instance, we bind the same task to one of the cores, first time Core0 and second time Core1. We find that the execution time as well as the peak temperature for each core are not the same. The peak temperature for Core0 is higher than that of Core1 for about 1 to 2°C, so we think that when we have $T_0 = T_1 + 1$ (where $T_0$ for Core0 and $T_1$ for Core1), the heat of both cores is balanced. Because of this, we set the migration threshold to be $T_H R_0 = T_H R_1 + 1$. The scheduling algorithm is shown in Algorithm 2, where $T_H R_0$ and $T_H R_1$ are thresholds of Core0 and Core1, respectively.

Algorithm 2 The implementation of Heat-and-Run.
1: if $T_0 \geq T_H R_0$ and $T_0 > T_1 + 1$ then
2: Find Hot task on Core0 and migrate it to Core1.
3: end if
4: if $T_1 \geq T_H R_1$ and $T_1 + 1 > T_0$ then
5: Find Hot task on Core1 and migrate it to Core0.
6: end if

6. EXPERIMENTAL SETUP AND RESULTS

To evaluate the performance of our thermal-aware RR algorithm, we choose Linux baseline algorithm and HR algorithm to compare with. Linux baseline algorithm is introduced in Section 1.4 and HR algorithm is presented in Section 5, so we will setup benchmarks for different workload and then show the experimental results.

6.1. Benchmark classification

We choose 15 SPEC2K benchmarks and several benchmarks from Mibench, Mediabench and Netbench. We start these benchmarks at CPU idle temperature of about 40°C and run each of them for more than 8 min on Intel Core 2 Duo processor. We use Linux shell command ‘taskset’ to bind them to Core0 and sample the temperature every 10s. Figure 8 shows the temperature profile for the above benchmarks. Since thermal management of Intel Core 2 Duo processor is more effective than that of Pentium 4 processor, the temperature is much lower than the maximum ratings, 80°C, as is compared with the temperature profiling in Yang et al. [3], even though we include all the hot benchmarks.

To classify these benchmarks into different thermal groups, we compute the average temperature of all the benchmarks when the programs have reached steady temperatures. The average temperature and classification of thermal groups of all benchmarks are shown in Table 1. Since most tasks are warm, we further classify warm tasks as either warm-hot or warm-cool.

We use different combinations of benchmarks, and each combination consists of three benchmarks as shown in Table 2.

6.2. Experimental results

The experiments were carried out in a room with air temperature 27°C. When both cores of the Intel Core 2 Duo E4500 processor are idle, it is interesting to find that the idle temperature of Core0 and Core1 are different, i.e. 37°C and 34°C respectively. Kursun and Cher [6] also observed a temperature variation between the two cores on their test chip in that Core 1 is almost always hotter than Core0. We implemented our RR scheduler, Linux baseline scheduler and HR scheduler into the Linux kernel and set a flag to trigger each of them for temperature monitoring. We record the total task execution time of a workload in different scheduling algorithms including the baseline.

FIGURE 8. Temperature profile for benchmarks.
TABLE 1. Average temperature and classification of program thermal intensity.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>T (°C)</th>
<th>Thermal group</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim</td>
<td>63.59</td>
<td>Hot</td>
</tr>
<tr>
<td>equake</td>
<td>62.63</td>
<td></td>
</tr>
<tr>
<td>art</td>
<td>62.08</td>
<td></td>
</tr>
<tr>
<td>applu</td>
<td>59.80</td>
<td>Warm–Hot</td>
</tr>
<tr>
<td>mgrid</td>
<td>59.65</td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>59.34</td>
<td></td>
</tr>
<tr>
<td>sixtrack</td>
<td>59.17</td>
<td></td>
</tr>
<tr>
<td>vpr</td>
<td>59.08</td>
<td></td>
</tr>
<tr>
<td>wupwise</td>
<td>58.81</td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td>58.65</td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td>58.07</td>
<td>Warm–Cool</td>
</tr>
<tr>
<td>mesa</td>
<td>57.97</td>
<td></td>
</tr>
<tr>
<td>apsi</td>
<td>57.87</td>
<td></td>
</tr>
<tr>
<td>twolf</td>
<td>57.86</td>
<td></td>
</tr>
<tr>
<td>g721</td>
<td>57.65</td>
<td></td>
</tr>
<tr>
<td>ammp</td>
<td>57.43</td>
<td></td>
</tr>
<tr>
<td>bitcnts</td>
<td>56.90</td>
<td>Cool</td>
</tr>
<tr>
<td>fft</td>
<td>56.68</td>
<td></td>
</tr>
<tr>
<td>md5</td>
<td>55.87</td>
<td></td>
</tr>
<tr>
<td>url</td>
<td>55.25</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 2. Different combinations of workload according to program thermal intensity.

<table>
<thead>
<tr>
<th>H,H,H</th>
<th>swim,equake,art</th>
</tr>
</thead>
<tbody>
<tr>
<td>H,Wh,Wh</td>
<td>art,vpr,bzip2</td>
</tr>
<tr>
<td>H,Wh,Wc</td>
<td>swim,wupwise,twol</td>
</tr>
<tr>
<td>H,Wh,C</td>
<td>art,gzip,bizcnts</td>
</tr>
<tr>
<td>H,Wc,Wc</td>
<td>swim,apxi,ammp</td>
</tr>
<tr>
<td>H,Wc,C</td>
<td>equake,mesa,fft</td>
</tr>
</tbody>
</table>

6.2.1. Temperature reduction

We observe that different scheduling algorithm can reduce the peak temperatures (PTs) on one or both cores for each workload. More evidently, with effective thermal-aware scheduling, the time each workload spends staying at their PT can greatly be reduced. These improvements indicate that thermal-aware scheduling can prevent the chip from being overly stressed by high temperatures, and hence, can improve the reliability and lifetime of the chip. Also, reducing the PT of the chip also implies that less DVFS (and less performance loss) will be encountered.

Table 3 shows the PTs and the aggregated time each workload stays in its PT (‘PT Stay Time’). We find that both HR and RR can either reduce or maintain the PT on both cores. Reductions are seen in workloads H-H-H (1°C on both cores), H-Wh-Wc (1°C on both cores) and H-Wc-C (1°C on Core 0). Furthermore, RR is stronger than HR in PT reduction. For example, in workload H-Wc-Wc, HR did not reduce the PT of the cores, but RR lowered it for Core 1 by 1°C. Similar result can be seen in H-Wc-C as well.

Another metric in thermal-aware scheduling is the reduction in PT stay time. As we can see, even if HR or RR does not reduce the PT of the cores in workload H-Wh-C, they both greatly reduced the PT stay time, e.g. from 28 s on Core 0 to 3.4 (HR) and 3.3 (RR)s. Also, when HR and RR has the same PT, RR often has significantly lower PT stay time than HR has. For example, in H-Wh-Wc workload, the PT of HR and RR is both 69°C, 1°C lower than that of the baseline. The PT stay time for HR and RR are 101 and 2.4 s respectively.

More intuitively, Fig. 9 shows the time split for different temperature values of all workload combinations under the three schedulers, especially red for PT stay time, yellow for second PT stay time and so on. For each workload combination, three vertical bar stands, respectively, for baseline scheduler, HR scheduler and RR scheduler. PT of Core0 is 70°C, while PT of Core1 is 69°C. RR scheduler performs a little better than HR scheduler in that RR’s red part (PT stay time) is less, while baseline scheduler contains much more red part, thus causing longest PT stay time.

Figure 10 depicts the temperature profile for workload combination H-Wh-Wc on dual core, which more evidently shows the PTs for this workload combination under HR and RR schedulers are much lower than that under baseline scheduler. What is more, the PT stay time under RR scheduler is considerably less than that under HR scheduler for both cores.

In summary, the baseline scheduler performs worst among the three schedulers in terms of thermal behavior. Although the HR scheduler migrates hot tasks from overheated cores to...
Implementing a Thermal-aware Scheduler in Linux Kernel on a Multi-core Processor

FIGURE 9. Time split for different temperature values of all workload combinations under baseline scheduler, HR scheduler and RR scheduler.

FIGURE 10. Temperature Profile for H-Wh-Wc on dual core.

balance the chip temperature, migration happens only when temperature is above a predefined threshold. However, our RR scheduler performs RR scheduling between the two cores to proactively control the thermal behavior of the chip when any core exceeds predefined temperature threshold. Thus, heat is almost balanced throughout the execution of a workload to achieve better performance than the HR scheduler.

6.2.2. Latency and throughput
Since thermal-aware scheduling enforces context switches for improving the thermal conditions on-chip, we measure its impact on both workload latency and throughput. The results are also shown in Tables 4 and 5. The latency of each workload is calculated by summing up the execution time of each job. Hence, the total latency overhead due to context switches for all jobs are considered. The throughput is calculated as the inverse of the workload finishing time, which is the finishing time of the longest job.

TABLE 4. Latency overhead for HR and RR.

<table>
<thead>
<tr>
<th>ET sum(s)</th>
<th>Base</th>
<th>HR</th>
<th>Overhead</th>
<th>RR</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>H-H-H</td>
<td>1174</td>
<td>1065</td>
<td>−9.28</td>
<td>1113</td>
<td>−5.20</td>
</tr>
<tr>
<td>H-Wh-Wh</td>
<td>653</td>
<td>617</td>
<td>−5.51</td>
<td>678</td>
<td>3.83</td>
</tr>
<tr>
<td>H-Wh-Wc</td>
<td>2122</td>
<td>2123</td>
<td>0.05</td>
<td>2251</td>
<td>6.08</td>
</tr>
<tr>
<td>H-Wc-Wc</td>
<td>2545</td>
<td>2615</td>
<td>2.75</td>
<td>2613</td>
<td>2.67</td>
</tr>
<tr>
<td>H-Wh-C</td>
<td>547</td>
<td>563</td>
<td>2.93</td>
<td>583</td>
<td>6.58</td>
</tr>
<tr>
<td>H-Wc-C</td>
<td>944</td>
<td>961</td>
<td>1.80</td>
<td>1002</td>
<td>6.14</td>
</tr>
</tbody>
</table>

TABLE 5. Finish time and throughput increase for HR and RR.

<table>
<thead>
<tr>
<th>Finish time(s)</th>
<th>Base</th>
<th>HR</th>
<th>Tp Inc.</th>
<th>RR</th>
<th>Tp Inc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>H-H-H</td>
<td>533</td>
<td>564</td>
<td>−5.50</td>
<td>564</td>
<td>−5.50</td>
</tr>
<tr>
<td>H-Wh-Wh</td>
<td>234</td>
<td>235</td>
<td>−0.43</td>
<td>233</td>
<td>0.43</td>
</tr>
<tr>
<td>H-Wh-Wc</td>
<td>798</td>
<td>778</td>
<td>2.57</td>
<td>765</td>
<td>4.31</td>
</tr>
<tr>
<td>H-Wc-Wc</td>
<td>1048</td>
<td>1090</td>
<td>−3.85</td>
<td>1071</td>
<td>−2.15</td>
</tr>
<tr>
<td>H-Wh-C</td>
<td>212</td>
<td>209</td>
<td>1.44</td>
<td>202</td>
<td>4.95</td>
</tr>
<tr>
<td>H-Wc-C</td>
<td>355</td>
<td>355</td>
<td>0</td>
<td>343</td>
<td>3.50</td>
</tr>
</tbody>
</table>
We can see that RR has latency overhead ranging from 2.67 to 6.58%. This mainly comes from constant task migrations between the cores. However, the throughput of the chip is hardly affected. Except for H-H-H and H-Wc-Wc workloads, there are noticeable improvements in throughput, ranging from 0.43 to 4.95% w.r.t the baseline scheduler. This is because RR regularly rotates tasks to each core for an equal amount of time. Therefore, the amount of total core idle time is greatly reduced, compared with the baseline. Such a benefit cannot be achieved by HR because task migration occurs only under certain condition. The property of improving both the thermal condition and chip throughput of our RR scheduler suggest that it is suitable for future many-core processor architectures targeting throughput-oriented workloads such as server applications.

6.2.3. Discussions
In a CMP, heat production on one core will raise the temperature on its neighboring cores. The heat generated on a single chip cannot be handled simply through threshold-based migration scheduling such as HR. It is necessary to balance the heat among all cores to maintain a healthy working status. The concept of RR scheduling is to proactively perform task migration among cores, to dissipate heat more evenly among them. The current Linux scheduler lacks thermal-awareness and throughput-friendly scheduling. This can be compensated for with a simple improvement using our proposed RR algorithm.

7. CONCLUSIONS
Our experiments with a sufficient number of representative benchmarks show that classic scheduling algorithms cannot meet tight thermal constraints on multi-core processors, and thermal-aware scheduling algorithms such as HR are inferior to our simple RR algorithm. Most existing thermal-aware schedulers are implemented on a simulator or using a numeric model, while we implemented our scheduler in Linux on a real multi-core processor. Our design and implementation incorporate physical impacts of multiples cores on each other such as heat propagation among multiple cores, randomness of Linux OS, difference in physical performance of multiple cores possibly due to process variations and hardware interruptions on physical processors. We also illustrate with our extended RR scheduler that thermal-aware scheduling can be improved with proper combinations of benchmarks and better configuration of time slices.

FUNDING
This work was partially supported by The National High Technology Research and Development Program of China (863 Program) (No. 2009AA012201), Shanghai Pujiang Program (Talented Scholar 07p14061) of China, NSFC (No. 60804003), Hong Kong RGC GRF Grant No. 613506 and 613108, and National Important Science & Technology Specific Projects Grant No. 2009ZX01038-001.

REFERENCES
Implementing a Thermal-aware Scheduler in Linux Kernel on a Multi-core Processor


