Thermal-aware Scheduling in OpenMP

ARTUR PODOBAS

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:114
I would like to thank my supervisor Mats Brorsson for the freedom I had in this work and his time, help, advices and directions when I was stuck as well as his mentorship.

I also would like to offer thanks to the SICS multicore groups for useful feedback in the different stages of this work.

Simics and Gems forum users have my thanks for all the question solved by reading through them.

Finally my thanks to the guys at Barcelona Supercomputing Center for their support in the Nanos++ and Mercurium compiler.
Contents

I Abbreviations 11

II Introduction 12

III Background Study 13

1 Introduction 14

2 Effects 15
  2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  2.2 Electromigration . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  2.3 Stressmigration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
  2.4 Time-Dependent Dielectric Breakdown . . . . . . . . . . . . . . . 16
  2.5 Resistor-Capacitor-Delays / Propagation Delays . . . . . . . . . . 17
  2.6 Ohmic drop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
  2.7 Package and Lifetime Cost . . . . . . . . . . . . . . . . . . . . . . . 18

3 The transistor and the inverter 20
  3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
  3.2 The MOSFET Transistor . . . . . . . . . . . . . . . . . . . . . . . 20
  3.3 The Inverter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4 Power calculation 21
  4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
  4.2 Static Power . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
  4.3 Dynamic Power . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

5 Power management 24
  5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
  5.2 Voltage and frequency scaling . . . . . . . . . . . . . . . . . . . . . 24
  5.3 Clock Gating . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
  5.4 Power-efficient logic . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
  5.5 Thread, Task and Process migration . . . . . . . . . . . . . . . . . 26
  5.6 Shutting down modules . . . . . . . . . . . . . . . . . . . . . . . . . 27
  5.7 Cool-loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

6 Scheduling and Thermal modeling 27
  6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  6.2 Predictive Dynamic Thermal Management for Multicore Systems 28
    6.2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
    6.2.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
    6.2.3 Model: Application-Based-Thermal-Modeling . . . . . . . . . . 28

3
6.2.4 Model: Core-Based-Thermal-Modeling ........................................ 28
6.2.5 Scheduler: Predictive Dynamic Thermal Management .................... 29
6.3 Temperature-aware Scheduler Based on Thermal Behavior grouping in Multicore systems ......................................................... 29
  6.3.1 Introduction ........................................................................... 29
  6.3.2 Results .................................................................................. 30
  6.3.3 Model: Core-Based-Thermal-Modeling ..................................... 30
  6.3.4 Scheduler: Prediction Based Thermal Management .................... 31
6.4 Temperature Aware Task Scheduling in MPSoCs ................................. 32
  6.4.1 Introduction ........................................................................... 32
  6.4.2 Results .................................................................................. 32
  6.4.3 Scheduler: Temperature-aware Task Scheduling in MPSoCs ........... 33
6.5 Temperature-aware MPSoC scheduling for reducing hot spots and gradients .................................................................................. 33
  6.5.1 Introduction ........................................................................... 33
  6.5.2 Results .................................................................................. 34
  6.5.3 Model: Task-Graph optimum ILP model .................................... 34
  6.5.4 Scheduler: Hybrid Temperature-aware scheduler ....................... 35
6.6 Proactive temperature balancing for low cost thermal management in MPSoCs ................................................................. 36
  6.6.1 Introduction ........................................................................... 36
  6.6.2 Results .................................................................................. 36
  6.6.3 Model: ARMA model ................................................................. 36
  6.6.4 Scheduler: Proactive Temperature Balancing Scheduler ............... 36

IV Methodology ................................................................. 38

7 Introduction ............................................................. 39

8 System ................................................................. 39
  8.1 CPU ....................................................................................... 39
    8.1.1 Introduction ........................................................................ 39
    8.1.2 Simulator .......................................................................... 40
    8.1.3 Power consumption ............................................................ 41
  8.2 Caches ................................................................................... 42
    8.2.1 Introduction ........................................................................ 42
    8.2.2 MSI .................................................................................... 42
    8.2.3 Simulator .......................................................................... 43
    8.2.4 Power Consumptions .......................................................... 43
  8.3 Network and Routers .............................................................. 43
    8.3.1 Introduction ........................................................................ 43
    8.3.2 Dimension-Order routing ..................................................... 44
    8.3.3 Simulator .......................................................................... 45
    8.3.4 Power Consumption ............................................................ 45
  8.4 On-Chip Temperature Sensor ....................................................... 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.4.1</td>
<td>Introduction</td>
<td>45</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Simulator</td>
<td>46</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Power Consumption</td>
<td>46</td>
</tr>
<tr>
<td>8.5</td>
<td>Hardware Counters and Monitors</td>
<td>48</td>
</tr>
<tr>
<td>8.5.1</td>
<td>Introduction</td>
<td>48</td>
</tr>
<tr>
<td>8.5.2</td>
<td>List of Magic-Instructions</td>
<td>48</td>
</tr>
<tr>
<td>8.5.3</td>
<td>Using the hardware counters and monitors</td>
<td>48</td>
</tr>
<tr>
<td>9</td>
<td>Software</td>
<td>49</td>
</tr>
<tr>
<td>9.1</td>
<td>OpenMP</td>
<td>49</td>
</tr>
<tr>
<td>9.2</td>
<td>Mercurium and Nanos++</td>
<td>49</td>
</tr>
<tr>
<td>10</td>
<td>Verification</td>
<td>49</td>
</tr>
<tr>
<td>10.1</td>
<td>System temperature test</td>
<td>49</td>
</tr>
<tr>
<td>10.2</td>
<td>System hardware counters test</td>
<td>49</td>
</tr>
<tr>
<td>10.3</td>
<td>Cache coherence verification</td>
<td>51</td>
</tr>
<tr>
<td>V</td>
<td>Benchmarks</td>
<td>55</td>
</tr>
<tr>
<td>11</td>
<td>Introduction</td>
<td>56</td>
</tr>
<tr>
<td>12</td>
<td>Fibonacci</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>nQueens</td>
<td>58</td>
</tr>
<tr>
<td>14</td>
<td>Multisort</td>
<td>58</td>
</tr>
<tr>
<td>VI</td>
<td>Implementation</td>
<td>59</td>
</tr>
<tr>
<td>15</td>
<td>Introduction</td>
<td>60</td>
</tr>
<tr>
<td>16</td>
<td>Case Study: Task-descriptors in Mercurium and Nanos++</td>
<td>61</td>
</tr>
<tr>
<td>17</td>
<td>Case study: Cilk-scheduler in Mercurium and Nanos++</td>
<td>62</td>
</tr>
<tr>
<td>18</td>
<td>Breadth-first Scheduler</td>
<td>66</td>
</tr>
<tr>
<td>19</td>
<td>Work-first Scheduler</td>
<td>67</td>
</tr>
<tr>
<td>20</td>
<td>Temperature-aware Scheduler</td>
<td>68</td>
</tr>
<tr>
<td>VII</td>
<td>Results</td>
<td>72</td>
</tr>
<tr>
<td>20.1</td>
<td>Metrics</td>
<td>72</td>
</tr>
<tr>
<td>20.2</td>
<td>Benchmark results</td>
<td>72</td>
</tr>
<tr>
<td>20.3</td>
<td>Fibonacci</td>
<td>73</td>
</tr>
</tbody>
</table>
## List of Figures

1. Chip-interconnect worn down by EM. Picture from Sanya Semiconductor. ........................................ 15
2. Gate-oxide breakdown (shows as a void) shown on a circuit. Picture from Sanya Semiconductor. ................ 16
3. Showing the propagation delay of the signal between the output of inverter one and the input of inverter two. ................................. 17
4. Ohmic-drop across a system’s power-rails. ................................................................. 18
5. CMOS Inverter. .................................................................................................................. 21
6. Leakage parasitics shown on the CMOS inverter. ........................................................................... 22
7. Energy and Dynamic power consumption. a) Charging the output node. b) Discharging the output node. c) Short-cut between $V_{dd}$ and GND. .................................................................................. 23
8. Example DVFS implementation. .............................................................................................. 24
9. Power distribution of a CPU (Caches not included) ....................................................................... 25
10. Example gated Clock tree. ...................................................................................................... 26
11. Inductive voltage drop. .......................................................................................................... 27
12. Flow-graph of the Predictive Dynamic Thermal Management scheduler. ........................................ 30
13. Thermal group and pattern. ..................................................................................................... 31
14. Flow-graph of the Prediction Based Thermal Management scheduler. ........................................... 32
15. $T_{low}$ and $T_{thr}$ shown in the sliding window. Picture from [12]. ................................................ 33
16. Flow-graph of the Temperature-aware task scheduler. .......................................................... 34
17. Flow-graph of the Hybrid Temperature-aware task scheduler. ................................................. 35
18. ARMA model order impact on accuracy. .................................................................................... 37
19. Overview of the system. ........................................................................................................... 40
20. CPU instruction power weights ................................................................................................. 41
21. CPU power trace using two processors. .................................................................................... 42
22. MSI Cache state transitions. Picture taken from NCSU Library. ............................................... 43
23. Cache latencies and energy consumption. .................................................................................. 44
24. Dimension order X-Y routing .................................................................................................. 45
25. Example router power trace ..................................................................................................... 46
26. Floorplan for one tile .............................................................................................................. 47
27. Tile64 chip layout. ................................................................................................................... 47
28. Doing a magic call using C. ..................................................................................................... 48
29. Function for reading a processor’s temperature. ........................................................................ 48
30. System temperature test .......................................................................................................... 50
31. Temperature image of the test-program. .................................................................................. 50
32. Ruby reacting to magic instructions ......................................................................................... 51
33. Example tester statistic dump .................................................................................................. 52
34. Case-1 coherence scenario ....................................................................................................... 53
35. Case-2 coherence scenario ....................................................................................................... 53
36. Case-3 coherence scenario ....................................................................................................... 54
37. Source code for Fibonacci ...................................................................................................... 57
38 Fibonacci for n=4 ........................................... 57
39 Queens task graph for a 4x4 table. ............................... 58
40 Multisort using parallel sort and merge. .......................... 59
41 Task spawn and Taskwait in fibonacci ............................ 61
42 Translated source-code for spawning a task and syncing using MCC and Nanos++ ........................................... 62
43 Scheduler class template in Nanos++ ................................ 63
44 The function for queuing tasks onto threads inside the Cilk-scheduler ........................................... 64
45 atSubmit function in the Cilk-scheduler ............................ 64
46 atIdle() function in the Cilk-scheduler ............................. 65
47 queue() function in the breadth-first scheduler .................. 66
48 atSubmit() function in the breadth-first scheduler .............. 66
49 atIdle() function of the breadth-first scheduler ................ 66
50 Breadth-first execution of given task graph ....................... 67
51 pop function inside the work-first scheduler ..................... 67
52 atIdle() function in the work-first scheduler .................... 67
53 A processor cooling-down function ................................. 68
54 atSubmit() of the temperature-aware cilk scheduler ........... 69
55 atIdle() function in the temperature-aware cilk scheduler ...... 70
56 atSubmit() flow-graph for the Temperature-aware scheduler ... 71
57 atIdle() flow-graph for the Temperature-aware scheduler ...... 71
58 Impact factor and Speed above $T_{thr}$ for all three benchmarks ... 72
59 Average component distribution for the three schedulers running Fibonacci ........................................... 73
60 Cores spent above threshold temperature during the benchmark 74
61 Task re-scheduled using the temperature-aware scheduler ...... 74
62 Average core temperature with the different schedulers ........ 74
63 Execution time, power consumption and energy cost of Fibonacci 75
64 Warmest and Coolest cores running Fibonacci on the three schedulers ........................................... 75
65 Average component distribution for Cilk and the Temperature-aware scheduler running queens ............................... 76
66 Cores spent above $T_{thr}$ temperature during the queens benchmark 76
67 Task re-scheduled using the temperature-aware scheduler ...... 77
68 Average core temperature with the Cilk and temperature-aware scheduler running queens .................................. 78
69 Execution time, power consumption and energy cost of queens. 78
70 Warmest and Coolest cores running queens on the temperature-aware and cilk scheduler .................................. 78
71 Average power distribution when running multisort ............... 79
72 Time above the threshold ........................................... 79
73 Re-scheduled tasks using multisort ................................ 80
74 Average core temperature for multisort ............................ 81
75 Scheduler absolute time,power and energy cost .................. 81
76 Warmest and coolest cores in multisort ............................ 82
**List of Tables**

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CMOS inverter actions and reactions</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Network parameters</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>Level-2 cache address distribution</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>Memory controller address distribution</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>Fibonacci parameters</td>
<td>73</td>
</tr>
<tr>
<td>6</td>
<td>Queens parameters</td>
<td>76</td>
</tr>
<tr>
<td>7</td>
<td>Multisort parameters</td>
<td>77</td>
</tr>
</tbody>
</table>
Abstract

Multi-core computer systems have widely been accepted as the future standard in computer architecture. Many simple processing cores bundled together on a single die have been shown more successful in terms of power consumption and execution performance compared to previous large and heavily pipelined uni-core systems. As the chip dimensions decrease, temperature effects start to become prominent. Elevated temperature gradients and hotspot on processors put an upper limit on both the execution performance and the lifetime of the devices, leading to failures, slowdown and eventually malfunction.

Most work involved in software-based temperature management in multi-core systems have been in the kernel-space, hidden from the user. This work hopes to change this, and show that it is possible to make user-level schedulers account for temperature changes occurring in the system. OpenMP is the current standard in parallel programming and was used to implement a scheduling policy that uses hardware feedback to effectively try to eliminate elevated temperatures on the chip. The system itself was simulated and modeled using well-established simulators and models.

The results were promising, showing a decrease in time spent above the critical temperature with up to 140 times in some benchmarks and a decreased power consumption in all the benchmarks as compared to the Cilk and Breadth-first scheduler. This alone should encourage more research in this area, and hopefully give rise to a future standard of user-level temperature control in parallel based schedulers.
## Part I
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit.</td>
</tr>
<tr>
<td>ARMA</td>
<td>Auto Regressive-Moving Average.</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>CMP</td>
<td>Chip-Multicore-Processor.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DC</td>
<td>Direct current.</td>
</tr>
<tr>
<td>DFS</td>
<td>Dynamic Frequency Scaling.</td>
</tr>
<tr>
<td>DPM</td>
<td>Dynamic Power Management.</td>
</tr>
<tr>
<td>DVS</td>
<td>Dynamic Voltage Scaling.</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling.</td>
</tr>
<tr>
<td>EM</td>
<td>Electromigration.</td>
</tr>
<tr>
<td>FET</td>
<td>Field-Effect Transistor.</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array.</td>
</tr>
<tr>
<td>FIT</td>
<td>Failure-In-Time.</td>
</tr>
<tr>
<td>ILP</td>
<td>Integer Linear Programming or Instruction Level Parallelism.</td>
</tr>
<tr>
<td>MOS</td>
<td>Metal-Oxide Semiconductor.</td>
</tr>
<tr>
<td>MIPS</td>
<td>A RISC processor architecture.</td>
</tr>
<tr>
<td>MPSoC</td>
<td>MultiProcessor System-on-Chip.</td>
</tr>
<tr>
<td>MTTF</td>
<td>Mean-Time-To-Failure.</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip.</td>
</tr>
<tr>
<td>RLSM</td>
<td>Recursive Least Square Method</td>
</tr>
<tr>
<td>SLICC</td>
<td>Specification Language for Implementing Cache Coherence.</td>
</tr>
<tr>
<td>SM</td>
<td>Stressmigration.</td>
</tr>
<tr>
<td>SPRT</td>
<td>Sequential Probability Ration Test.</td>
</tr>
<tr>
<td>TDDB</td>
<td>Time-Dependent-Dielectric-Breakdown</td>
</tr>
</tbody>
</table>
Part II
Introduction

OpenMP is the current standard in task- and thread-based parallelism, and this unlikely to be changed anywhere in future. Current scheduler algorithms that exists in OpenMP, algorithms such as the Cilk or Breadth-first scheduler do not take chip temperature into account. Elevated temperatures on small areas, so called hotspots, degrade the chip which eventually leads to system failure. Previous work in this area focus on either hardware solutions, such as frequency scaling, or kernel-level scheduling.

This work has been about creating a simulation platform to evaluate different scheduling algorithms, as well as creating a temperature-aware scheduler to try to counter and give the programmer a way to control temperatures on the chip, and not blindly rely on the kernel to do so.

The simulation model, based on the system simulator Simics, was made to support a x86 and a SPARC based instruction set architecture. Together with the timing and interconnect simulator Ruby, this was the skeleton for the system. Inspired by Tilera processors family topology, the system used a mesh-type interconnect as means for communication, and the cache coherence was based on a distributed, banked L2 cache also serving as a directory. To this, a self-written power-module was added, serving both to calculate the power using programs such as HotSpot and Cacti, but also interrupt instructions and add power weight to them, and provide hardware counters and temperature sensors for the simulated system to use.

The OpenMP model used is based on BSC Nanos++ research framework. It was chosen for its simplistic nature of changing and creating new scheduling policies as well as the support. It is also one of the few OpenMP models available open-source.

Using this system and an OpenMP model, a scheduler was developed that has the awareness of the temperature around it.
Part III

Background Study
1 Introduction

Unwanted temperature increases has always limited the rated of which the execution performance of computer could be improved.

Some 40 years ago, the CMOS transistor was introduced, and it is today the most used transistor in the digital field. The shift to multi-core processors some years ago had a similar impact on the market; today all computer systems can be said to be multi-core systems. Both these shifts were driven by their abilities to cope with temperature increases, [17, 23].

However, these two shifts have not closed the issue of elevated temperatures. The semiconductor industry continues to improve existing technologies, and when creating integrated circuits in the deep sub-micron domain, some previously ignored effects cause problems. The carrier mobility, which is the drift the charge carriers have related to the applied electric field across them, decreases with increased temperature resulting in slower devices. Increased temperature also changes the layout of the chip; electron-migration changes the shape and resistance of the interconnects and stress-migration moves the interconnects around. The elevated chip temperature also increases the probability of transistor malfunction due to the dielectric breakdown. This probability has a larger than exponential dependency on temperature [30].

Perhaps the most noted effect that increased temperature bring to everyday life is the additional cost related to the chip package, cooling and degradation. Studies shown that temperature differences as low as 5-15 degrees may give a 50% increase in FIT [23]; this is one of the reasons that the cooling cost increases at a super-linear rate [12].

The voltage and clock frequency are two variables related to the power consumption of the system; controlling them proves to be the most effective way of lowering the power consumption and temperature of the device [31, 22]. The method used to control voltage and clock frequency is DVFS. The main premise for using DVFS is that a system only needs to use its maximum performance for small time fragments [26]. By lowering the voltage and/or clock frequency according to the computational needs, studies have shown that power consumption can be reduced by as much as 84% [25].

The clock-tree in a typical high-performance processor consumes 40-45% of all the power [31], disconnecting the clock from idle modules inside the processor would therefore save power. This technique is called clock-gating and can decrease power consumption by as much as 72% [16], depending on the design.

The two methods described above are the most effective and widely used methods for controlling temperature and power. However, due to the nature of multi-core processors, there is a possibility to schedule computational workloads onto the different cores with temperature in mind. Research on these scheduling techniques ([12, 10]) shows great potential in ability to lower temperature without limiting performance too much.
2 Effects

2.1 Introduction

This section gives the motivation for actually managing thermal hotspots on today’s electronic devices. Many of the explained effects affect the direct lifetime and durability of the devices, while others reduce the functionality or price.

2.2 Electromigration

Electromigration (EM) is a phenomenon that occurs when metal films are stressed with high current densities. Electrons move at a speed relative to the electric field affecting them. Together with high current densities, this causes the momentum to be transferred from the electrons, to the metallic conductor ions. This is called the 'electron wind’ force. Moving conductor ions causes voids and hillocks (lumps) in the interconnects; this will result in:

- Open-circuit (void)
- Short-circuit (hillocks)
- Change in the interconnect resistance.

Figure 1 shows an example of a wire worn down by EM.

The EM that is induced in the interconnect is increased exponentially with increased temperature. A formula for calculating the MTTF caused by EM was first given by Black [6] 10 years after EMs discovery by Fiks [32] in 1959.

$$MTTF = A * \omega * j^{-n} e^{\left(\frac{Q}{kT}\right)}$$
2.3 Stressmigration

Stressmigration (SM) is a phenomenon much like Electromigration; metal atoms in the interconnects migrate from one place to another. The reason is however different. Interconnects in the integrated circuits are formed by depositing metallic ions onto the chip in a certain patterns; the temperature with which the metallic ions are deposited is $T_0$. Later, when the chip is in use and the interconnect temperature changes due to use of the device, the interconnects will expand at different rate depending on how much the current temperature $T$ differs from the deposited temperature $T_0$. This can cause open circuits (voids) to form on the interconnects altering the function of the circuit.

The mean-time-to-failure caused by Stressmigration (SM) can be calculated [18] by:

$$MTTF = |T_0 - T|^n * e^{\frac{E_a}{kT}}$$

2.4 Time-Dependent Dielectric Breakdown

Time-Dependent Dielectric Breakdown (TDBB or gate-oxide breakdown) is a failure mechanism explaining the breakdown of the isolating layers between various nodes of the transistor (e.g. between the gate and the drain). Such breakdowns will result in a possible functional or parametric failure on the chip. Figure 2 shows an example of gate-oxide breakdown in a circuit.

The MTTF for TDBB has been formulated using experimental [15] data over different oxide-thickness, voltages and temperatures and is:

$$MTTF = \left(\frac{1}{V}\right)(a-bT) * e^{\frac{(X+\frac{1}{2} + Y)T}{kT}}$$

The model shows that the gate-oxide breakdown is highly dependent on the voltage and has a larger than exponential degradation due to the temperature [30].

Figure 2: Gate-oxide breakdown (shows as a void) shown on a circuit. Picture from Sanya Semiconductor.
2.5 Resistor-Capacitor-Delays / Propagation Delays

The propagation delay between two points in an electronic system is a function of the resistance ($R$) and the capacitance ($C$) between those two points. The propagation delay can be calculated using the Elmore delay expression \[ t_{pe} = 0.69 * R * C \]

Figure 3 shows a simplified motion-of-flow for propagating a signal from the output of one inverter, through the capacitors and resistors of the wire, to the input of the other inverter. Seen in (a) is the initial flow of current to charge up the capacitor ($C_{dr}$) through the driver-resistance ($R_{dr}$). (b) continues to charge up the load ($C_{L}$ which is the capacitance of the gate in the other inverter) through the sum of the resistance ($R_{DR} + R_{Line}$). (c) shows the the stable state where the voltage at the load is the same as the voltage at the source ($V_{C_L} = V_{C_{dr}}$).

The electrical resistivity is directly related to temperature by the Bloch-Gruneisen formula [3]:

\[ \rho(T) = \rho(0) (1 + \gamma T) \]
Figure 4: Ohmic-drop across a systems power-rails.

\[
\rho(T) = \rho(0) + A \left( \frac{T}{T_0} \right)^n \int \frac{e^{nx}}{(e^{x} - 1)(1-e^{x})} dx
\]

Increased resistance increases the propagation delay through the system, and therefore limits the maximal clock frequency that is available to the system.

2.6 Ohmic drop

Increased temperature increased the resistance; increased resistance increase the potential ohmic drop of the system. Voltage(U), Current(I) and Resistance(R) relates to each other by Ohm’s law: \( U = I \times R \); increased resistance also increases the voltage across that resistance. This can happen across the power-rails where large currents constantly passes to supply the system.

Shown in figure 4 is an example of ohmic drop across a system. The picture does only show parasitic resistances in the supply path although the return path does also contain them.

High temperature that increases the resistance can potentially cripple the whole system. Shown in figure 4, the voltage across the inverters are not the same; the second and last inverter may even receive low voltage levels that disable their functionality.

2.7 Package and Lifetime Cost

A ever-growing factor is the cost of the system. Systems that generate more heat naturally needs packages that are able to dissipate the heat; this increases
the package price. Every watt of power consumption above 35-40 Watt increases the cost by $1 [31]. This cost is most likely much higher than than it was in 1998 when the study was done. Giving an example on a larger scale, the total amount of energy used by server and data centers in the US year 2006 was 61 billion kWh (~1.5% of total US electricity consumption and $4.5 billion worth of costs) [28]; this was twice the usage compared to 2000.

Increased temperature also negatively impacts the lifetime of a chip as broken chips needs to be replaced. Temperature management is therefore important to increase the lifetime and decrease the cost of systems.
3 The transistor and the inverter

3.1 Introduction

Basic knowledge of the transistor and the most basic of all CMOS circuits, the inverter, is essential to understand how power is consumed in most of today’s digital circuits. This section will give information on how CMOS circuits work, how power is consumed and how to calculate it.

3.2 The MOSFET Transistor

The metal-oxide-semiconductor is certainly the workhorse of digital design [24]. Even though it is used within analog design, it’s main use is in the digital design where its switch-performance is very good and it introduces very few parasitic effects. Improving manufacturing techniques also allow the MOSFET transistor to be put together at extremely high volumes and small sizes.

There are two types of MOSFETs: the NMOS and PMOS transistor. The NMOS transistor works as a switch: applying a positive voltage corresponding to a logic ’1’ will switch the transistor on and allow current to flow from the voltage supply to the ground. A logic ’0’ switches the transistor off and there is no path between the supply voltage and the ground.

A PMOS transistor work in the direct opposite way. A logic ’1’ will switch the transistor off while a logic ’0’ switches it on, conducting current between supply and ground.

Combining these two properties give rise to the basic CMOS circuit: the inverter.

3.3 The Inverter

The inverter is the most fundamental of all digital circuits; understand it is critical for any digital designer. It is formed by inserting a PMOS and NMOS transistor between supply and ground, connecting the gates of these to a common input.

Figure 5 shows the CMOS Inverter. Note that the PMOS is being implemented as the pull-up network while the NMOS is the pull down network, giving the inverter the function:

\[ Out = 1 - I_n \]

<table>
<thead>
<tr>
<th>( V_{IN} )</th>
<th>( V_{Out} )</th>
<th>( C_L )</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Charged.</td>
<td>( C_L ) has been charged and is effectively connected to ( V_{DD} )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Discharged.</td>
<td>( C_L ) was discharged and the output is effectively connected to GND.</td>
</tr>
<tr>
<td>0\rightarrow1</td>
<td>1\rightarrow0</td>
<td>Discharging</td>
<td>( C_L ) is discharging through the active NMOS down to GND.</td>
</tr>
<tr>
<td>1\rightarrow0</td>
<td>0\rightarrow1</td>
<td>Charging</td>
<td>( C_L ) is being charged through the active PMOS to ( V_{DD} )</td>
</tr>
</tbody>
</table>

Table 1: CMOS inverter actions and reactions
Table 1 shows the actions and reactions of the CMOS inverter in terms of input voltage change.

4 Power calculation

4.1 Introduction

In this section, it will be shown how the simplest of CMOS gates, an inverter, give rise to the power that warms the chip up. This power model can be applied to any set of CMOS gates (AND, NAND, NOR etc.) since they work the same way, however, with many more capacitive nodes to take account for.

4.2 Static Power

Static power is when the CMOS inverter is not changing, but still consuming power. The static power (or leakage power) comes from the many different parasitic diodes that form between the doped areas on the transistor to the substrate as well as from capacitive nodes in the transistor; no capacitor is so good that it does not leak current.

Figure 6 shows an example inverter with the leakage current and capacitor parasitics included. The leakage currents for the diodes can be calculated using:

\[ I_S = \left( \frac{e k V}{q T} - 1 \right) \]

4.3 Dynamic Power

The dynamic power consumption is a result when charging up the internal capacitance’s, load capacitance’s, and the shortcut that form due to an transition in output voltage.

Figure 7 shows the steps involving in a logic state change from 0->1 in an inverter.
a) The input is steady at a logic '0' allowing the PMOS to conduct a current between Vdd and Vout, efficiently charging up $C_L$. Some of the power will however be dissipated inside the PMOS due to its drive-resistance.

b) The input is steady at a logic '1' allowing the NMOS to conduct thus discharging $C_L$ to GND. Some of the power will be dissipated inside the NMOS due to its driving-resistance.

c) The input is in the middle of switching between '0' and '1'. The NMOS and PMOS are both simultaneously conducting. This create a shortcut between $V_{DD}$ and GND efficiently discharging $V_{DD}$ to GND and dissipating heat through both the transistors resistance.

The dynamic power due to switching can be calculated as:

$$ P = C_L \cdot V_{dd}^2 \cdot f_0 \cdot f^{'1} $$

The power consumed is linked to the operating voltage of the circuit, the capacitances of the nodes inside the circuit and the clock frequency. Lowering any of these three will decrease the power consumption.
Figure 7: Energy and Dynamic power consumption. a) Charging the output node. b) Discharging the output node. c) Short-cut between $V_{dd}$ and GND.
5 Power management

5.1 Introduction

As seen from previous sections there’s a great need to control the temperature of the chip to reduce various lifetime and cost variables. The temperature is related to power densities on chip, and reducing the power in the area will effectively reduce the temperature. This section will describe some ways on how it is possible to reduce the chip-temperature using software and hardware solutions.

5.2 Voltage and frequency scaling

The power consumed by a CMOS digital-system is proportional to the voltage and the frequency the system uses. Thus, reducing the voltage is one of the most effective way to reduce the power [31], as is reducing the clock frequency.

A popular way of implementing this into a system is by the use of DVFS. The idea of DVFS works under the assumptions that a system only needs its peak performance during small time fragments, the rest of the time an average performance is sufficient [26].

An example of an implementation that uses DVFS [26] is given in figure 8. It includes a DC-to-DC converter, a programmable clock and a microprocessor that supports different operative voltages. When peak-performance is needed the maximum clock frequency and voltage supported by the microprocessor is used and in all other cases the voltage and clock is lowered according to the microprocessor load. Energy reduction varies but reports of up to 20%-82% less energy consumption are reported on different benchmarks and DVFS-scheduling algorithms [25].

There are however some issues with voltage scaling. To maintain performance, the threshold voltage \( V_T \) must be scaled as well. At low voltages (thus low \( V_T \)) leakage power start becoming a dominating factor [31] and with low enough \( V_T \) the leakage power actually starts ruining the benefits since many dynamic circuits (PLA’s, caches etc) becomes vulnerable when the leakage current rises.
Worth mentioning is that even though the chips today uses low $V_{dd}$ it is still beneficial to use voltage scaling. A 100nm process with a supply voltage of 1.2 V will have a $V_T$ of around 0.35 V and can potentially improve power consumption by up to 10 times [8].

5.3 Clock Gating

Figure 9 shows the distribution of power in an typical high-performance microprocessor [22]. The clock and its components consume roughly 40-45% of all the power. By eliminating certain paths in the clock-tree (and therefore the total capacitance that is needed to (dis-)charge during transition) the power consumed by the system can be reduced. This technique is called clock gating and is shown in figure 10. The figure shows an clock tree driven by inverters that drive all the clocked elements in the system. Each node has a capacitor and for each clock transition, all the capacitors must be (dis-)charged. This consumes a lot of power and by turning off paths of the clock-tree (shown in the figure as an AND-gate), one can control the amount of capacitors that need to be (dis-)charged and thus control the energy.

The difference in power consumption between a system using clock gating and a system that does not can be up to 72%, in favor for the system with clock-gating [16].

While clock gating provide a stable and consistent way of reducing the power it do have some problems. When frequencies reach 1 GHz and above a severe clock skew may occur and therefore the timing analysis may have to be reconsidered. Another problem is with nodes that have been switched off by the clock gates. The nodes may have recovery time to (dis-)charge the nodes and thus creating clock glitches in that sub-tree.
Another problem is the inductive voltage drop that might occur on the power rails. Since clock gating allows to potential disable the charge-up of large part of the clocktree, it indirectly allows different current densities to return back. Since power-pins can be seen as parasitic inductance's, the potential voltage drop across these inductance's can cause malfunction of the system. The voltage drop is given by:

$$V = \frac{\delta i(t)}{\delta t} \cdot L$$

This requires the power delivery system to handle additional noise, and thus increases the system and package cost. Figure 11 shows a picture with inductive drops on the rails.

### 5.4 Power-efficient logic

The hardware design choice of digital logic is an important factor to the power consumption, and therefore the temperature, of the chip. Besides actively implementing the control techniques above into the hardware, it is important to decide on the architecture of sub-units of the system. As an example, a comparison between two different hardware adders can be given. A normal ripple-carry adder consumes 1.17 W while the much faster carry select adder consumes up to 2.16 W [9]. This shows how important it is to chose and design hardware that accomplish the tasks given, but with power consumptions in consideration.

### 5.5 Thread, Task and Process migration

This software type of temperature management normally involves keeping track of the current temperature on the chip by the means of various feedback mechanism, and potentially move work from one processor core to another to reduce the current load and thus the temperature. The target core may be chosen
randomly, or more effectively moved to a core with the largest number of cold cores surrounding it [13].

5.6 Shutting down modules
If a sub-module, for instance a processor core, is not currently used it may prove effective to shut it down to reduce temperature dissipation from this core to others as well as the power consumption. This requires that the architecture supports such functionality.

5.7 Cool-loops
Cooling loops are software that causes the processor to decrease power consumption, yielding lower temperature. One of the potential advantages in using a cooling loop over hardware solutions is the potential decrease in hardware cost. A cooling loop can be introduced as a sequence of no-ops [10] instructions or power reducing operations to reduce potential fetch throttling, cache accesses and traffic.

6 Scheduling and Thermal modeling
6.1 Introduction
This section explains some approaches to scheduling and controlling the temperature on-chip with the goal of eliminating hotspots and temperature gradients. Most of the schedulers use SPEC-benchmarks to evaluate their methods, allowing two or more benchmarks to run in parallel of each other on the multicore
system. Also, some of the schedulers use off-line profiling to model the performance of the applications they will run for later predictions.

6.2 Predictive Dynamic Thermal Management for Multi-core Systems

6.2.1 Introduction

The Predictive Dynamic Thermal Management, devised by Yeo et al [34] is a scheduler based on two different thermal modeling schemes: the application-based-thermal-model (ABTM) and the core-based-thermal model (CBTM). ABTM predicts the temperature based on the application given and CBTM predicts based on the current core temperature. Yeo et al have evaluate their scheduler using the combination of some common benchmarks (libquantum + bzip2) running simultaneously or alone.

6.2.2 Results

The PDTM scheduler was compared against the standard Linux scheduler and two existing thermal-aware schedulers using two SPEC2006 benchmarks (libquantum and bzip2). The experimental platform was a Intel Quad-core with specific drivers to access the thermal sensor.

The PDTM was shown to reduce peak temperature by up to 5 degrees with only 1% loss in performance if compared with the standard Linux scheduler. When running both libquantum and bzip2 it lowered average temperature with up to 10%. The PDTM did also perform better than the two other thermal-aware scheduler in both prediction accuracy and lowered average temperature.

6.2.3 Model: Application-Based-Thermal-Modeling

The Application-Based-Thermal-Model (ABTM) aims to profile and mathematically describe a given application off-line using a RLSM, and later use this model as a weight for predicting future temperatures. The motivation for doing this is that the temperature difference can vary extremely rapidly in small intervals, even if the workload is static.

The model itself can be represented using the following formula:

\[ y = \theta_1 \ast f_1(u) + \theta_2 \ast f_2(u) + \ldots + \theta_n \ast f_n(u) \]

If all the functions \( f(0)...f(n) \) are known functions of \( u \) and \( \theta \) has been estimated using off-line profiling, the predicted temperature \( y \) can be calculated.

6.2.4 Model: Core-Based-Thermal-Modeling

The models relies on the fact that the temperature in a processor core changes proportionally to the difference between the current temperature and the steady-state temperature. The steady-state temperature is the temperature obtained when a core is continuously running a task until the temperature of the core doesn't change.
Thus, a formula for the temperature $T$, at a certain time $t$ can be estimating by:

$$\frac{dT}{dt} = b \times (T_{ss} - T) \quad (1)$$

($T_{ss}$ = Temperature steady-state. $b$ = constant).

Solving formula (1) with $T(0) = T_{init}$ and $T(\text{infinite}) = T_{ss}$ gives:

$$T(t) = T_{ss} - (T_{ss} - T_{init}) \times e^{-bt} \quad (2)$$

Obtaining $b$ and $T_{ss}$ is done by the following two steps:

1) Run the benchmark of interest on each core until the temperature of the core do not change. Record this temperature and call it the steady-state temperature ($T_{ss}$).

2) Substituting $T_{ss}$ and $T_{init}$ with the values acquired in earlier steps, we read the thermal sensor of the chip and calculate $b$ using formula (2).

6.2.5 Scheduler: Predictive Dynamic Thermal Management

Yeo et al. presents a algorithm that uses both the ABTM and CBTM models for providing a predictive temperature model. The motivation for using both is that ABTM provides a model for very rapid changes in temperature, while CBTM provides a model for slower changes (taking the lag of the temperature sensor in account). Thus, to provide robust predictability that do not ping-pong between values, the scheduler takes both models and add a weight to them. The total temperature prediction is:

$$T_{\text{predict}} = W_s \times T_{\text{app}} + W_l \times T_{\text{core}} \quad (1)$$

($T_{\text{predict}}$ = Predicted temperature. $T_{\text{app}}$ = ABTM model. $T_{\text{core}}$ = CBTM model. $W_s$ and $W_l$ = weights).

The study uses benchmarks which executes at 100% workload at all the time, and provides the recommended weights as: $W_s + W_l = 0.7 + 0.3 = 1(100\%)$. Should the workload be different, $W_s + W_l < 1$ could be chosen.

Figure 14 shows the flow-graph of the scheduler:

1) Calculate current temperature
2) If the current temperature is larger than a triggered threshold then (3, 4, 5)
3) Calculate when this occur?
4) Calculate all other cores temperature at the predicated time for this core.
5) Migrate the process to the coolest one.
6) If the predicted value is higher than a given priority trigger then decrease or increase the priority for this prediction.

6.3 Temperature-aware Scheduler Based on Thermal Behavior grouping in Multicore systems

6.3.1 Introduction

The Prediction Based Thermal Management scheduler devised by Yeo et al [20] is based on the CBTM (see subsection 6.2). The model is however used together with a method that is called K-means clustering; applications can be divided into thermal groups with the same temperature characteristics. The scheduler
was evaluated against several of the SPEC2006 benchmarks on Intel quad-core and dual quad-core machines.

6.3.2 Results

The PBTM scheduler was compared against the Linux default scheduler and two thermal-aware schedulers (amongst other the PDTM, see subsection 6.2). The system was an Intel quad-core and an Intel dual quad-core machine. The PBTM outperformed all the other schedulers in both temperature and efficiency. It reduced average temperature by up to 5 degrees with an 7.58% performance overhead compared to the Linux default scheduler on the dual quad-core machine and by 8 degrees and less than 12% performance overhead on the quad-core.

6.3.3 Model: Core-Based-Thermal-Modeling

The model relies on the fact that the temperature in a processor core changes proportionally to the difference between the current temperature and the steady-state temperature. The steady-state temperature is the temperature obtained when a core is continuously running a task until the temperature of the core doesn’t change.

Thus, a formula for the temperature temperature $T$, at a certain time $t$, can be estimating by:
\[
\frac{dT}{dt} = b \times (T_{ss} - T) \quad (1)
\]
\(T_{ss} = \) Temperature steady-state. \(b = \) constant.

Solving formula (1) with \(T(0) = T_{init}\) \(T(\infty) = T_{ss}\) gives:

\[
T(t) = T_{ss} - (T_{ss} - T_{init}) \times e^{-bt} \quad (2)
\]

Obtaining \(b\) and \(T_{init}\) is done by the following two steps:

1) Run the benchmark of interest on each core until the temperature of the core doesn’t change. Record this temperature and call it Steady-State temperature \((T_{ss})\).

2) Substituting \(T_{ss}\) and \(T_{init}\) with the values acquired in earlier steps, we read of the thermal sensor of the chip and calculate \(b\) using formula (2).

6.3.4 Scheduler: Prediction Based Thermal Management

Yeo et al. propose that application is to be clustered into \(k\)-different groups depending on their thermal behavior. Twelve different SPEC2006 benchmarks were executed and their thermal responses were recorded. It was found that five different groups would be the optimal to classify the twelve applications. These thermal groups are related to each other by their steady-state temperature \((T_{ss})\); applications in group A will have similar \(T_{ss}\). These thermal groups are profiled off-line.

When a new application (one not previously profiled) is executed there is a need to classify this application into one of the thermal \(k\)-groups for prediction. It was observed that the thermal patterns of an application can be divided into three regions (figure 13). Each of the regions slope affects the temperature differently. To calculate the slope of the operating temperature, the model uses:

\[
S_i = \frac{T(i+\delta t) - T(i)}{\delta t}
\]
(Si = slope for the i’th region. T(i) previous temperature. T(i + δt) is current temperature. 
δt occurs on predefined time intervals. Using the slope and the current temperature, an estimation can be derived based on how the current slope behaves in comparison with the groups.

14 shows a flow-graph of the scheduler:
1) Calculate the slope.
2) Fetch Tss.
3) Fetch b.
4) Check all cores: If the time predicted to reach maximum temperature is within 2 sec do 5.
5) If any core satisfy the above condition, then migrate the process to a target core with the longest predictive time to reach maximum temperature.

6.4 Temperature Aware Task Scheduling in MPSoCs
6.4.1 Introduction
Coskun et al [12] presents a scheduling algorithm based on probabilistic of sending works to a certain core. The scheduler was evaluated using a simulator for power recording and HotSpot thermal profiling on a UltraSPARC T1 Niagara system.

6.4.2 Results
The scheduler was implemented inside various other schedulers that did not support DPM before. The schedulers that include the temperature-aware scheduler
does perform better than without DPM and with negligible performance overhead.

6.4.3 Scheduler: Temperature-aware Task Scheduling in MPSoCs

The Temperature-aware task scheduler [12] is based on probabilities of sending a new task to a certain core. The formula for acquiring the new probabilities of a core is:

\[ P_n = P_0 + W \]  

This calculation is invoked when new tasks arrive. \( W \) is the weight and it is computed at regular intervals using an one second sliding window of temperature changes of the cores. The core that will receive the work is randomly selected. \( W_{\text{dec}} \) and \( W_{\text{inc}} \) are the weights that the probabilities of the cores are increased or decreased with. Two constants are present: \( T_{\text{Low}} \) and \( T_{\text{thr}} \). Figure 15 shows them in relation to an example temperature sliding window.

The sliding window is what determines the increase in probabilistic. If any of the temperatures inside the sliding window have been above \( T_{\text{thr}} \), then the probability of that core receiving new work is set to 0\%, else it is increased by \( W_{\text{inc}} \). If the temperature was below \( T_{\text{low}} \) at any point in the sliding window, \( W_{\text{inc}} \) is calculated as:

\[ W_{\text{inc}} = \beta/A_{V_{\text{Thr}}} \]  

\( A_{V_{\text{Thr}}} \) is the average temperature below \( T_{\text{Thr}} \) divided by \( T_{\text{Thr}} \). Cooler cores probability to receive work is increased much more than the probability for warmer cores.

Figure 15: \( T_{\text{low}} \) and \( T_{\text{thr}} \) shown in the sliding window. Picture from [12]

fig16 shows a flow-graph of this scheduler:
1) Check if work has arrived. If yes, continue with 2.
2) Calculate \( W_{\text{inc}} \) for all cores.
3) Calculate \( P_n \) for all cores.
4) Randomize a number and send work to that core.
5) Continuously update each core’s temperature sliding window.

6.5 Temperature-aware MPSoC scheduling for reducing hot spots and gradients

6.5.1 Introduction

This scheduler, devised by Coskun et al [13], is based on off-line profiling to calculate the optimum scheduling policy for a given workload. It is assumed
that the task-graph and it’s dependencies are known beforehand. The scheduler was evaluated using a UltraSPARC T1 Niagara system. Task-graph information were collecting using Sun’s Continuous System Telemetry Harness(CSTH) program.

6.5.2 Results
The scheduler was shown to minimize hotspot and gradients by comparing with different other schedulers. It was also shown that it performs much better than pure dynamical scheduling.

6.5.3 Model: Task-Graph optimum ILP model
This model uses a task-graph together with Integer Linear Programming(ILP) using a set of constraints to calculate an optimum scheduling. The input to this model is the number of cores available to the system $P_0...P_n$, the task-graph $T$ with all its tasks $T_0...T_n$, all the voltage levels available to the core $V_0...V_k$. Each voltage setting is also associated with a performance number (in terms of cycles/second).

Using this input and a set of constraints and rules for calculating energy and temperatures, an ILP model can find the optimum way to schedule according to these rules.

Figure 16: Flow-graph of the Temperature-aware task scheduler.
Figure 17: Flow-graph of the Hybrid Temperature-aware task scheduler.

Amongst the constraints are:

- Each task runs on only one CPU.
- Each task runs at only one voltage level.
- If two task’s are scheduled on the same core, one of them precedes the other.

Using these and more constraints, the ILP can calculate in order to:

- Minimize thermal hotspots and gradients
- Minimize and balance thermal hot spots
- Balance energy consumption.
- Minimize total energy consumption.

6.5.4 Scheduler: Hybrid Temperature-aware scheduler

The hybrid temperature-aware scheduler is hybrid between the ILP solution presented above, and function to deal with unknown tasks. It uses Coolest-FLP which is a method of migrate a thread from a hot core to a cooler core; the cooler core is the coolest core with the most idle cores surrounding it.

Figure 17 shows a flow-graph of this scheduler:

1) Fetch the new task.
2) Check if this task exist in the before-hand known task-graph.
3) If it does exist, then schedule it according to the ILP scheduling.
4) If it does not exist, then schedule all remaining and future tasks according to the Coolest-FLP. Send each new task to the coolest core that is surrounded by most idle neighbors to reduce temperature overlap between cores.
6.6 Proactive temperature balancing for low cost thermal management in MPSoCs

6.6.1 Introduction

This model is presented by Coskun et al [11] and uses an auto regressive moving average (ARMA) type of model to predict the future temperature. This is a proactive, on-line predictor. The scheduler was evaluated on the UltraSPARC T1 Niagara system using various standard benchmarks (gzip, gcc, mplayer, web etc).

6.6.2 Results

The model was verified both using a Hotspot floorplan and a real UltraSPARC T1 system connected to Hotspot running on another system (to eliminate overhead) that continuously monitors and calculates the temperature activity. The experiments yielded 60% reduction in hotspot occurrences, 80% reduction in spatial gradients and average 75% reduction in thermal cycles compared to other schedulers.

6.6.3 Model: ARMA model

The model uses ARMA modeling for predicting future temperature on the cores. The ARMA is a way used to estimate and create a model based on a time-data series. The model is actively used in economics to monitor and predict future changes and trends. In this case the time-data series is temperature $T$ at a certain time $t$. The formulation for the ARMA model is composed of two parts; the AR part (Auto-Regressive) and the MA (Moving Average) part. The complete formulation is:

$$y_t + \sum (a_i * y_{t-i}) = e_t + \sum (c_i * e_{t-i})$$

By rewriting the formula we can predict the temperature at the time $t$.

The steps involved in creating a ARMA model consist of identification (finding out the order needed for the model, figure 18 shows impact of different model order), estimation (computing the coefficients of the model for each order) and verification (checking the model). For identification, the authors used a trial-and-error way of establishing the order; starting out with first order and increasing until satisfaction. The satisfaction can be calculated using Final-Prediction-Error (FPE). Coefficients were calculated using MatLab and reported to take less than 150ms for a AR(p) model and less than 300ms for a ARMA(p,q) 5-th order.

6.6.4 Scheduler: Proactive Temperature Balancing Scheduler

The scheduler devised by Coskun et al [11] uses the ARMA modeling together with a SPRT to predict future temperature of cores. If the thermal model predicts correctly, then anomalies should fluctuate around 0. However, if residuals starts drifting away towards -M or +M, the ARMA model is updated to account for these anomalies. M is a reassign threshold. This is to ensure the
The ARMA model is continuously up-to-date and reflects the background for more correct prediction. The SPRT is formulated as:

$$SPRT = \frac{M}{2} \sum (T_c(i) - T'(i) - \frac{M}{2})$$

The scheduler itself deals with predicting possible thermal emergencies in cores. If a thermal emergency is predicted to arise, several methods for preventing it is used:

- Proactive Thread migration - Moving workload from cores.
- Proactive DVFS - Reducing Voltage/Frequency settings.
- Proactive Temperature Balancing - Follows the principle of locality (allocating threads on the same core)
Part IV
Methodology
7 Introduction

In order to explore different scheduling strategies, and their effect on temperature, a multicore system was created. It included 16 cores connected to each other and the memory controllers by a mesh type interconnect. A mesh interconnect topology is a so-called Network-on-Chip (NoC) which, unlike bus-based topologies, sends data using messages (so-called flits) instead of broadcasting. This rather uncommon topology was used because there are very few studies done on the temperature aspects of such a topology, and, to our best knowledge, none has investigated user-level scheduling. Possible alternatives would have been the UltraSPARC T1 Niagara processor which has been used in many papers concerning temperature scheduling as this system is open-source and synthesizable on a FPGA leads to more detailed studies.

Aside from the standard functions of the system, a temperature sensor was added to each core, and functionality was developed to access the temperature sensor from inside the simulation. To get as much data about the simulation as possible, support was added to dump the current power values of caches, routers and CPUs. This allowed for power-tracing and thus see average power distributions amongst cores. This feature also makes it a basic experimental platform for studying power consumption itself. The system was modeled to resemble the Tilera TILE64-family of processor which, to our knowledge, is the only commercially built multicore processor based on the mesh-type of interconnect.

8 System

An overview of the system can be seen in figure 19. As previously stated, the system is a 16-core chip, connected by a mesh interconnect. There are four memory controllers in the system, two at north and two at the south side of the system.

The memory controllers as well as the L2 caches are statically distributed depending on the address. The L2 is divided into 16 banks, and is shared across the chip. Each tile (or core) has its own private L1 cache, a L2 bank as well as a router and a temperature sensor. The tile also includes a CPU. The entire system (including the NoC) runs at 750 MHz, because the same clock frequency is used in the Tilera multicore family of processors.

8.1 CPU

8.1.1 Introduction

The CPU is the main workhorse of the system, and it executes code written by the programmer. The CPU includes a set of registers, that are used as temporary placeholders during computations, and an ALU that performs arithmetic
operations on the registers. Most CPUs also include, beside the mandatory ALU and registers, devices that improve the CPI of the processor. ILP, branch prediction, hypervideo, scouting are some common methods employed to improve the performance of the processor.

8.1.2 Simulator

The system is simulated using Simics [2]; a full system simulator capable of simulating a great number of processors and systems. Simics is a functional simulator which means it only simulates the functionality of the system and lacks architectural details; all instructions take the same time to execute, no matter what kind the instruction is and how long it takes to execute it in a real system. Simics also comes with the possibility to simulate cache and memory hierarchies however this feature was not used as Gems provides more robust and accurate cache simulations.

The CPU core itself is based on either the Pentium 4 (x86) or the SPARC v9 instruction set architecture. Both architecture can be used as the temperature-sensor supports trapping both ISAs.
8.1.3 Power consumption

The CPU power-model is very simple, but fully suitable for the kind of experiments done in this work. Ideally, a CPU power-model would include detailed information about the energy consumed by the different actions happening inside a processor, actions such as accessing the register file, branch prediction, pipelines and so forth. Although there exists such power simulators, e.g. Wattch [7] including such a model together with Simics, Ruby, Hotspot and Orion does not lie within the scope of this work.

The power-model is based on two things:

- Activity. The amount of instructions execute during the current timeslot
- Instruction type. The type of instructions executed during the current timeslot.

A breakdown of the instruction power weights that was used is shown in figure 20. Each instruction contains one or more of these weights, and different instruction therefore consumes different amount of power. The weight is attached to the maximal power of the CPU, allowing potentially up to 0.9W (using 1 CPI) or 1.8 W (using 0.5 CPI) of sporadic power although the average power is around 180-300 mW which is comparable to the ARM Cortex-R4 (which has roughly the same area and power consumption if scaled up to 750 MHz). Ten ADC instruction will consume more power than ten CLI instructions. NOP instructions have been given the special property to consume zero energy and are used to control the temperature of the processor (in the case of overheating). Other studies [10] show that NOP instructions do indeed consume less energy and can be used to counter overheating.

Figure 21 shows a snapshot of the power trace of two CPUs running a benchmark.
8.2 Caches

8.2.1 Introduction

The system contains private L1 caches, and a shared L2 cache. The L2 cache is split into 16 banks with each bank located inside a tile. The total on-chip L2 cache is 1 MB in size and each L1 cache is 8 kB. Cache sizes, associativity and line size was based on the Tilera TILE64 processor, taken from the manual. The coherence protocol is a MSI type protocol. The L1 is strictly inclusive to the L2 meaning that the L2 knows the current states of the L1 cache-lines.

8.2.2 MSI

The cache coherence protocol chosen is the MSI-protocol. There are three states: M (Modified), S (shared), I (Invalid) that each line can be in. The modified state means that the current processor has written to this cache-line. Should another processor also write to the same cache-line, the modified cache-line will be changed to invalid, indicating that this cache no longer holds a correct value of this cache-line. The shared state means that there are several copies of this data in various caches. Normally, a cache switches to a shared state when it reads a value not present in the cache or if another processor requests a value that is in the modified state in the current cache. The states for the MSI protocol on a broadcasting medium is shown in figure 22. The difference between the broadcasting medium and the system used in this work is that the directories is responsible for invalidating and keeping track of the state bits of cache-lines, while in a snooping based medium each cache is responsible for invalidating and keeping track of state bits.

![Example CPU power trace](image)

Figure 21: CPU power trace using two processors.
8.2.3 Simulator

Multifacet Gems [1] is a architecture and cache/network simulator that features OPAL, which is a cycle accurate SPARC v9 architecture simulator and Ruby which is a cache and interconnect simulator. Ruby also contains the Garnet [5] and Orion [33] interconnect power and simulator.

Ruby provides many different interconnect topologies (meshes, toruses, butterflies) as well as the possibility to describe unique custom topologies (e.g. ring-topologies). Various cache coherence protocols are supported and a SLICC is also supported giving the user ability to create custom cache protocols.

Gems is the only existing way to relatively easy create and simulate mesh-type interconnects and the associated coherence protocols within Simics. It also has a very good support forum.

8.2.4 Power Consumptions

The power consumptions of the L1 and L2 caches were modeled using Cacti [14], a cache and memory modeling tool. The model is simple, only the data-read and data-write are modeled. The instruction cache is not modeled at all due to a limitation in the Ruby protocol disabling instruction fetches for x86 simulations. Figure 23 shows the cache latencies and energy consumption.

8.3 Network and Routers

8.3.1 Introduction

In a NoC interconnect architecture, such as the mesh interconnect used in this study, the routers are responsible for receiving and redirecting traffic toward its destination. Messages can be of different sizes and are composed of a number of
flits; the messages start with a unique head-flit and end with a unique tail-flit, in between are flits containing the data to be transmitted. The parameters of the network are shown in table 2.

8.3.2 Dimension-Order routing

Garnet uses the dimension-order routing algorithm. In this system, a message first travels along the X-axis from the source, until it reaches the same horizontal location as the destination and then proceeds to travel along the Y-axis. This work by having a router-table inside each of the routers, with information where every tile is relative to this one. The routing-table and routing algorithm is generated from the interconnect model when Ruby starts up. To switch from X-Y routing to Y-X routing, one must make sure that the latencies on the vertical links are less than the latencies on the horizontal links. Figure 24 shows how a message from tile #4 to tile #10 travels using the routing algorithm.

Table 2: Network parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Topology</td>
<td>Mesh</td>
</tr>
<tr>
<td>Routing algorithm</td>
<td>Dimension-order X-Y routing</td>
</tr>
<tr>
<td>Virtual Channels</td>
<td>4</td>
</tr>
<tr>
<td>Flit size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Number of nodes</td>
<td>16</td>
</tr>
<tr>
<td>Link bandwidth</td>
<td>12-24 GB/s</td>
</tr>
</tbody>
</table>
8.3.3 Simulator

The network simulator is Garnet and is included in the Ruby model. For power consumption, the Orion (also included) power simulator was used.

8.3.4 Power Consumption

Power is automatically extracted from the Orion power simulator, which is based on 4 statistics:

- The number of messages read into this router's buffer.
- The number of messages written from this router's buffer.
- How many times the crossbar was used.
- Activity in the virtual channel arbiters.

Figure 25 shows an example power trace of the router located at node7.

8.4 On-Chip Temperature Sensor

8.4.1 Introduction

The power consumed in each core is ultimately calculated and converted into temperature, with the possibility for the simulated system to access this value. This feature is necessary as the system have no notion or idea about temperature without it. The components fed into the temperature sensor is all the power traces within the current timeslot. Since this is supposed to model a real temperature sensor the sample time should not (but could) be set to less than one tenth of a millisecond. The temperature sensor outputs the value calculated in various ways such as temperature images, power consumption graphs and temperature graphs. It is also the means for hardware monitoring and simulated feedback.
8.4.2 Simulator

The simulator used to calculate temperatures is the Hotspot [29] modeling tool. Hotspot takes a floorplan and a power trace as an input, and outputs the temperature relative to the duration of the power trace. The floorplan used in this project is estimated using parameters received from the Cacti cache modeling tool and the ARM Cortex-R4 area.

The floorplan for one tile (seen in 26) is the size of a TILE64 core. The TILE64 were derived visually and are shown in figure 27.

Other studies (e.g. [19]) also verifies the dimensions assumed for the tilera chip is reasonable. Hotspot provides two means of calculating the temperature: the Block model and the Grid model. The Block model treats each component as a block with one temperature value while the Grid model tries to estimate how intra-block temperature would be. Used within the temperature sensor is the block model, due to it simplistic natures and faster calculation.

8.4.3 Power Consumption

The temperature sensor groups all components within a tile (CPU, L1, L2 and Router) together and uses Hotspot to calculate the temperature of that block. The sampling time can be varied but is 250k cycles. Besides calculating the temperature, the temperature sensor is also responsible for dumping statistics to files during the simulation.
Figure 26: Floorplan for one tile

Figure 27: Tile64 chip layout.
8.5 Hardware Counters and Monitors

8.5.1 Introduction

Simics provides a way for the simulated system to communicate with the host system. This is done with the so-called Magic-Instructions. The Magic-Instruction itself is not a special instruction, but rather a instruction that is rarely used in the system, and minimally affects the execution. For instance, the Magic-Instruction in the x86 platform is: `xchg bx,bx`, which exchanges the value of register `bx` with itself effectively doing nothing save consuming the time needed to execute it (the “xchg” instruction do never modify any flags).

When a Magic-Instruction is used, an exception handler in Simics forwards the exception to Ruby. Inside Ruby, exception-handling functions exists and since the Simics x86-platform only allows for one-type of magic instruction, it accesses the value of the register “eax” to determine what actions to perform.

8.5.2 List of Magic-Instructions

8.5.3 Using the hardware counters and monitors

Figure 28 shows how to use Magic Instructions within a C function. The input to this function is based on the list of supported calls in subsection 8.5.2.

Figure 29 shows a function to check the temperature of a processor in the system. A return value holder, (i) is defined as volatile (to prevent compilers from modifying it). The value of “msg” is later put into the current processors
32-bit eax register, and marked as clobbered for the compiler to know. The pre-defined Magic-Instruction is called (iii) and the return value is put into the volatile return value holder ret (iv). The value is later returned (v).

9 Software

9.1 OpenMP

OpenMP is a programming model created by a group that was representing several major vendors of high-performance computing [27]. It uses compiler directives and library routines to expose parallelism. By adding these compiler directives to a sequential program, the users specifies what parts are to be executed in parallel and how. As of version 3.0, OpenMP supports constructs for creating tasks (previous versions focusing on data-parallelism). The parallel constructs for creating a task is “#pragma omp task” and the construct for synchronizing with all previously spawned tasks is “#pragma omp taskwait”.

The motivation for using OpenMP is due to it being the standard in representing task based parallelism and that the Nanos++ supports great customization features.

9.2 Mercurium and Nanos++

Mercurium is the source-to-source compiler used with the Nanos++ library created by the Barcelona Supercomputing Center. The aim with Nanos++ is to create a experimental research OpenMP platform to easily change and compare different task-based strategies in both homogeneous and heterogeneous computing. The motivation for using Nanos++ and Mercurium was in its customizable scheduling engine. See part VI for more information how Nanos++ was used.

10 Verification

10.1 System temperature test

A toy-program were assembled to create one core (core 4) with increased cache and CPU activity and one core (core 10) with increase CPU activity while all other remaining in low power mode. Figure 30 shows the power traces for both the CPU and the L1 cache.

As expected, the core with increase cache and CPU activity is also the warmest followed by the core with elevated CPU activity. Figure 30 shows the generated temperature image of the system; Core4 and Core10, are the warmest.

10.2 System hardware counters test

A program was created to try out the different hardware counter. Each thread randomly generated a few of the Magic Instruction and verified that Ruby
Figure 30: System temperature test

Figure 31: Temperature image of the test-program.
Figure 32: Ruby reacting to magic instructions

Caught them.

Seen in figure 32 is an example how Ruby reacts when Magic Instructions are used.

10.3 Cache coherence verification

Interleaving  As previously stated, the L2 cache is sub-divided into 16 banks, forming one global on-chip L2-cache out of many small local ones. The address space is distributed among the L2 banks in 128-byte segments, shown in table 3.

The address space is also distributed among the memory controllers, in a very similar way that they are in the L2 banks shown in table 4.

<table>
<thead>
<tr>
<th># Controller</th>
<th>Address-range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00-0x3F</td>
</tr>
<tr>
<td>1</td>
<td>0x40-0x7F</td>
</tr>
<tr>
<td>2</td>
<td>0x80-0xFB</td>
</tr>
<tr>
<td>3</td>
<td>0xC0-0xFF</td>
</tr>
</tbody>
</table>

Table 4: Memory controller address distribution
Verification To examine and verify the correctness and function of the MSI protocol in our system, Ruby provides a tester program, which can be used without any processor simulator to examine different protocols.

The protocol tester requires a trace of transactions made, specifying what kind of transaction it is, who initiated the transaction and when the transaction was initiated.

Trace

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPU0 writes to address 0x00380. This address is not present anywhere on the chip.</td>
</tr>
</tbody>
</table>

Result Figure 34 shows the statics for the trace.
Figure 34: Case-1 coherence scenario

```plaintext
1 0 -1 Seq Begin > [0x380, line 0x380] ST
4 0 0 L1Cache Store MP-L1_IMM 0x380, line 0x380
38 0 7 L2Cache L1_GETX L2_IMM-L2_IMM 0x380, line 0x380
96 0 2 Directory GETX_HotOwner MP-SM 0x380, line 0x380
154 0 7 L2Cache Data_ext_sok_0 L2_IMM-L2_MT 0x380, line 0x380
189 0 -1 Seq Done > [0x380, line 0x380] 188 cycles NULL ST Yes
189 0 0 L1Cache L1_Data L1_IMM-L1_IMM 0x380, line 0x380
```

Figure 35: Case-2 coherence scenario

**Explanation** CPU0 initiates the store and since this line is not present in the local L1 cache (NP), it changes the state bit to an intermediate state (L1_IMM) showing that it soon will have it modified. It forwards a L1_GETX message through the network to the L2 Bank #7 which is the home node for this line. The L2 Bank #7 proceeds to inform the directory (memory controller) that this line is now modified and later, when it receives the acknowledge from the directory, changes it’s own state to MT (Modified in local cache). Later the L1 Cache line changes from IM (intermediate modified) to Modified finishing the transaction off. Note how subsequent writes don’t affect the L2Cache nor the directory.

**Case 2**

**Description** CPU0 continuously reads from address 0x00380. This address is not present anywhere on the chip at the first read.

**Trace**

```
0 0x00380 0 LD
0 0x00380 1 LD
```

**Result** Figure 35 shows the statics for the trace.

**Explanation** The request arrives at the L1Cache controller which emits a L1_GETS message to the L2 Bank due to the address being not present in the L1 cache. Since the address is not present there either, the L2 Bank proceeds to request the address from the directory (Memory controller) which responds with the data. The L2 Bank changes its state bits to Shared and delivers the data to the L1 Cache, which now also switches to shared. Subsequent reads do not trigger any new activities.
Case 3

Description  CPU0 loads from address 0x00380 and CPU1 later writes to the same address (0x00380).

Trace

0 0x00380 0 LD
1 0x00380 1 ST

Result  Figure 36 shows the statistics for the trace.

Explanation  CPU0 starts the transaction and events similar to Case 2 are triggered. When CPU0's L1 cache receives the value requested, CPU1 starts its transaction trying to write to that address. The write triggers an invalidation message thrown from the L2 Bank to CPU0's L1 cache. The L2 Bank continues to update the directory and finally submit acknowledge to CPU1's L1 cache.
Part V
Benchmarks
Algorithm 1 Calculating Fibonacci number

\[ F_n = (F_{n-1}) + (F_{n-2}) \]

11 Introduction

Benchmarks are programs specifically designed to burden the experimental platform in certain ways to evaluate the performance of a certain metric on the platform. Some common examples of metrics are execution speed and power consumption. This work focuses on temperature as a metric and three different benchmarks have been trialed: Fibonacci, nQueens problem and Multisort. The three of these represent different focus points: Fibonacci involves very little calculation and data-related code, nQueens problem involves more calculation intensive code and some data-related code while Multisort sorts an a large array thus becoming very data-intensive. These three benchmarks are well-known and used within the parallel programming community. Fibonacci is usually done as a stress-test, considering spawning very many tasks with small granularity. Both multisort and nQueens are included in the BOTS [4] and various other studies performed to evaluate programming model performance(ex [27]).

12 Fibonacci

The Fibonacci series, named after Leonardo Pisano Bogollo\(^1\) is a number series calculate using the formula given in algorithm 1.

The program code for this is inherently extremely parallel, as there are no lock mechanisms involved and only task synchronization is used. The code is void of any dense computations or data-manipulations, and thus executes extremely fast. It is usually used as a first program to write when learning a new parallel programming model due to it’s simplicity. Figure 37\(^2\) shows the program code for calculating the Fibonacci sequence in parallel using the OpenMP tasking model.

The OpenMP header (i) is included and a function for fib, (ii), with a return value and the input argument “n” is declared. If n < 2 then we return n (iii) since that is already the Fibonacci number for n. Should n be larger than 1, we recursively call fib for calculating Fibonacci numbers (n-1) and (n-2) (v, vii). These calls are created as tasks (iv, vi) using the OpenMP syntax given. A #taskwait statement is inserted to ensure that the tasks have completed before returning the Fibonacci number.

Before calling the function fib(), a team of threads\(^3\) must be created. (x) and (xi) specifies that the next code-line will be executed by the main-thread.

Figure 38 shows how the task-spawn diagram looks for Fibonacci's series.

---

\(^1\)His nickname was Fibonacci

\(^2\)There is no depth-control included in the picture. Task-parallel programs usually have depth control to easily control exposed parallelism.

\(^3\)This is required by the current OpenMP syntax but not using Nanos++
```c
#include <stdio.h>
#include <omp.h>

int fib(int n) {
    if (n<2) return n;
    int x,y;

    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);

    #pragma omp taskwait
    return (x+y);
}

void main(void) {
    #pragma omp parallel
    #pragma omp single
    printf("Fib(35) = %d\n", fib(35));
}
```

Figure 37: Source code for Fibonacci

Fibonacci series n=4

![Fibonacci tree](image)

Figure 38: Fibonacci for n=4
13 nQueens

Given a NxN sized chessboard and N queens, in how many different ways can one place the queens on the chessboard so that they do not threaten each other? The problem was first defined by Max Bezzel [3]. The algorithm works by creating n arrays of size n, and in each array put a queen on position from [0..n] and checking if this queen threatens any other queen from the line preceding this. If it is not threatening any other queen, the function is recursively called with the new array as input and the procedure repeats. Figure 39 shows how the nQueens algorithm works. Code has been explicitly excluded from here.

14 Multisort

Multisort is a more data-oriented benchmark created originally for the cilk programming model. It takes an array of data as input and sorts it. It has been included amongst other in the BOTS for evaluation of programming models. It works by recursively dividing the array and its segments into four quarters, and then recursively sort them. When finished, each resulting array is merged in parallel with the others to produce the final sorted array. Code has been explicitly excluded from here but figure 40 how the multisort works.
Figure 40: Multisort using parallel sort and merge.

Part VI
Implementation
15 Introduction

A scheduler is basically a small and fast piece of code, that execute before, after, or in between task-execution. Before going further into how schedulers work, a introduction to what task is and how they it is treated in the parallel programming model.

A task is basically a chunk of code, be it large or small, that can include program control and usually can be executed independently on a thread. Even data-parallel code can be thought of as tasks, where each iteration of a data-parallel loop is a task scheduled on a thread. Tasks are an excellent way to expose parallelism due to their often recursive nature, and the workload can easily be adjusted by changing the granularity of the task. These properties (independence and granularity) enables a well-written scheduler to implement functions such as work-stealing, load-balancing, task-migration and so forth to achieve the performance goal aimed at (temperature, execution speed etc).

How is a task defined code-wise? A task is a structure (s.k. task-descriptor) which includes a pointer to the function that corresponds to the task, a return value holder and the parameters to the function. Additional information is usually added to help the scheduler, such as information about who the parent that spawned the task is or a task id number.

The input to the scheduler is a lot of task-descriptors and it is the schedulers job to decide how and who should execute them. In the simplest of cases there’s one queue where all the tasks are scheduled into, and all the cores running would fetch tasks from this queue and call the function within. A more complicated scenario would include multiple queues and work-stealing between idle cores and more activate ones. Or, more related to the temperature-aware scheduler, a scenario where each core has it’s own execution queue and tasks are scheduled to the coolest of cores. This chapter explains the implementation of different schedulers which this work’s temperature-aware scheduler was evaluated against, as well as a detailed description of the temperature-aware scheduler.

---

4These are called work-descriptors in Nanos++
To illustrate the different steps given in the introduction, task-creation and scheduling, a small case-study on how these steps are performed inside the Barcelona Mercurium [21] compiler and the Nanos++ library is given below. For simplistic reasons, the chosen program-segment will be from an OpenMP version of the classical fibonacci-program, a well known example in the academia.

The fibonacci program was one of the benchmarks trialed in this work and can be read about in section 12. The program-segment is found in figure 41.

Figure 42 shows how part of the OpenMP code is transformed to use the Nanos++ library functions.

(i) is the OpenMP syntax for spawning the task specified by the next line (ii). The shared(x) constructs insist that the variable x has been declared before and that it will use that one. (iii) stalls to ensure that the program will not continue beyond that point unless all the previously spawned tasks in this function have finish executing.

(1) shows the implementation of (ii) using a function with the difference that the passed argument is a pointer to where the arguments for the fib-call is stored and where to put the return value. This is the function that the task-descriptor will point to and call when executing this task. (2) creates a structure with a pointer to the function described in (1). This will later be used in the initialization of the task-descriptor. (3) Storage for the passed arguments and the return value is created. (4) This is where a pointer pointing to the function is created. (5) The task-descriptor is initialized and allocated. We supply the values of the address of the task-descriptor (&wd) , the pointer to the function (_ol_fib_0__devices) and a pointer to where the arguments are stored ((void **) &ol_args) along with some additional options (parent_wd, properties etc.) (6,
7) After the task-descriptor has been initialized, we setup pointers to where the arguments are. (8) The task is finally submitted and is ready for scheduling. More options can be specified here, such as dependencies. This is where the scheduling takes place, as the final call obtained by tracing this function will be: Scheduler::submit(). (9) This is the corresponding Nanos++ function for (iii), which basically waits until all tasks tied to the current task-descriptor (nanos_current wd()) are completed.

17 Case study: Cilk-scheduler in Mercurium and Nanos++

Submitting a task-descriptor ultimately calls the schedulers at submit() function for the current thread, where the final decision where and how the task will execute. Most schedulers aim to balance the load over the available resources, in order to maximize the execution speed using methods such as work-stealing or work-sharing: once a thread or core become idle, it tries to steal work from other cores surrounding it. Other, more simple schedulers, randomly distribute work to cores or don’t distribute at all, but simply put them in a common central queue.

As an example, we will go through one of the included schedulers inside of Nanos++: the Cilk scheduler. Before going into detail, a brief explanation is given in algorithm 2:

---

A thread and a core is in this case the same.
**Algorithm 2** Cilk-scheduler algorithm

- Each thread has its own task queue.
- Should no work be present in the local queue the thread will try steal its parent. If the parent cannot be stolen, the thread will proceed to try steal from another queue.
- Each thread focuses on depth-transversely meaning that each spawn will trigger an execution of that task.

```cpp
Class: CilkPolicy( public of type SchedulePolicy)

Private:
struct ThreadData : public ScheduleThreadData {
    WDeque _readyQueue;
    ThreadData () : _readyQueue() {}
    virtual ~ThreadData () {
    }
};

Public:
CilkPolicy() : SchedulePolicy ("Cilk") {}                  (1)
virtual ~CilkPolicy() {}                                  (2)
virtual size_t getTeamDataSize ();                        (3)
virtual size_t getThreadDataSize ();                      (4)
virtual ScheduleTeamData * createTeamData ( ScheduleTeamData *preAlloc ); (5)
virtual ScheduleThreadData * createThreadData ( ScheduleThreadData *preAlloc ); (6)
virtual void queue ( BaseThread *thread, WD &wd );      (7)
virtual WD * atSubmit ( BaseThread *thread, WD &newWD ); (8)
virtual WD * atIdle ( BaseThread *thread );              (9)
```

Figure 43: Scheduler class template in *Nanos++*
Figure 44: The function for queuing tasks onto threads inside the Cilk-scheduler.

```cpp
virtual void queue ( BaseThread *thread, WD &wd )
{
    ThreadData *data;
    if ( wd.isTied() ) {
        data = ( ThreadData * ) wd.isTiedTo()->getTeamData()->getScheduleData();
    } else {
        data = ( ThreadData * ) thread->getTeamData()->getScheduleData();
    }
    data->readyQueue.push_front ( &wd );
}
```

Figure 45: atSubmit function in the Cilk-scheduler

```cpp
virtual WD * atSubmit ( BaseThread *thread, WD &newWD )
{
    return &newWD;
}
```

The Cilk scheduler class template is seen in figure 43. (1) defines the task-descriptor queue, readyQueue, of type WDDeque\(^6\). This is the storage class used to push and pop work for each thread. There is an constructor and destructor(2 & 3) that can be used to initialize this scheduler at start-up. (4 & 5) simply returns the size of the private members.

(8) is responsible for enqueuing a task-descriptor into a thread’s queue. Since this follows the OpenMP standard it must take into consideration if the task-descriptor is tied to another thread\(^7\).

seen in figure 44 is the function for queuing tasks onto a threads workpool, line (iii) check if the task-descriptor is tied, and if so, check what thread it is tied to and points to that queue. If it’s not tied it points to the thread originally destined to get the task-descriptor (iv) and finally pushes the task-descriptor onto the pointer queue (v).

As previously explained in section 16, the last call in the task-creating chain is the scheduler call to atSubmit(). This function (9) in figure 43, is called each time a task is created and it is time to schedule it. The Cilk scheduler in Nanos++ works using a depth-first transverse, meaning that the first task that is spawned is immediately execute, seen in (ii). Figure 45 shows how this is done.

If a thread is not submitting tasks nor executing them it is idling. The final public member, (10) in figure 43, is called whenever a task is finished and the

---
\(^6\)The WDDeque is a queue which can push and pop values from both ends as well as find and remove elements from within the queue.

\(^7\)A tied task always executes by the same thread, while a untied task may execute by several threads.
thread has no work to do. Figure 46 shows the code for this function.

(ii & iii) creates a placeholder for a task-descriptor and a pointer to the threads local queue. If there is work inside the local queue (i.e., POP:ing the queue with a return value of something else than NULL) then return that task-descriptor and execute it.

Should there be no work in the local thread queue, it will try fetch work from the current task-descriptors parent (iv). If work is found it returns the task-descriptor and start executing it. Should there be no work at either the local queue, and the steal from the parent failed then a cycling loop through all the threads is performed with attempts to steal from them. (vi) ensures that none else stole it, or that it is not executing at this very moment.

(vii) gets the current threads id and (viii) finds the total numbers of threads. (ix) sets the task-descriptor to NULL. A long loop is started (x) which increases the target identifier (xi) modulo the amount of threads in order to cycle through all of them. (xiii) attempts to steal a task, and if it’s a successful steal then the loop will break and a task-descriptor containing the work will be returned (xiv & xv). If the loop have cycled through all the threads and no work has been found, NULL is returned an no work executed.
Breadth-first Scheduler

The breadth-first scheduler is one of four default schedulers included in the \textit{Nanos++} library. It is a scheduling policy which always executes task-descriptors from the front of the queue, and creates new descriptors at the back of the queue, making it execute all leaves on a certain task-depth before going deeper; hence the name breadth-first.

The relevant functions to the algorithm can be found within queue(), atSubmit() and atIdle(). Figure 47 shows the queue function. (i) creates a pointer to the current thread data and (ii) pushes the given task-descriptor to the back of the queue.

Figure 48 shows the atSubmit function, which is called every time the thread spawns a task. Unlike the Cilk scheduler explained in section 17, this actually enqueues the data instead of executing it immediately. New task’s are always ensured to be pushed to the back of the queue(i).

Figure 49 shows how tasks are executed using the breadth-first scheduler.
Figure 50: Breadth-first execution of given task graph.

As seen in the figure, each depth is fully completed before going deeper down the leaves.

19 Work-first Scheduler

The work-first scheduler is the last of the described schedulers included in the Nanos++ scheduler plug-in package. It uses a hybrid version work-stealer, similar to the Cilk scheduler, but with the versatility to pop the target queue from the back or from the front (LIFO or FIFO). Having this option enables strategies such as breadth-first on stealing while depth-transverse on local. This scheduler is by default locally depth-first. That is, the function atSubmit() immediately starts execution of the task.

To be able to chose from what end of a queue to fetch task-descriptors from, a self-written written pop procedure is written and is shown in figure 51. Depending on the input policy it will either pop from the front or from the back of the queue.

The atIdle() function in the work-first scheduler uses the pop() function to either receive tasks from the local queue using policy (i), or steal the parents

Figure 52: atIdle() function in the work-first scheduler
Figure 53: A processor cooling-down function.

Algorithm 3 Temperature-aware Scheduler algorithm

- A recently created task should be scheduled onto the coolest of cores if the current core is too hot.
- If a thread is idle then makes sure the thread is cool enough before stealing or executing new tasks.

using policy (ii). Work-stealing can also be enabled or disabled (iii).

20 Temperature-aware Scheduler

The temperature-aware scheduler is based on the Cilk scheduler explained in section 17, but with provided functionality for reading hardware temperatures supported by the simulation platform, and schedule according to them. To read the temperature of a CPU, a function CHECK() is called which interacts with the system using a Magic Instruction (see subsection 8.5 for more information) and receives the temperature of the requested CPU.

There is a cool-down mechanism implemented, to make sure that an overheated processor can cool itself down. This uses the property of the "nop", which consumes no power as described in subsection 8.1. The function for cooling a CPU down is shown in figure 53; a large loop of "nop" instructions with the intention to maximize the ratio between "nop" instructions and other instruction as much as possible.

Many of the changes made to the original Cilk scheduler lies inside the atSubmit() and atIdle() functions. The algorithm can be explained with two simple lines (seen in algorithm 3):

To implement the first condition of the algorithm, a modified version of atSubmit() is shown in 54. The algorithm is the following: Get this cores current temperature, that is, the core submitting a task-descriptor (i). Create a pointer to the currently best core for scheduling (ii & iii). Save the coolest temperature, which currently is this threads temperature (iii). If this threads temperature
virtual WD * atSubmit ( BaseThread *thread, WD &newWD )
{
    int my_temp,best_temp;
    my_temp = CHECK(thread->getTeamId());
    BaseThread *best_thread = thread;
    BaseThread *target;

    best_temp = my_temp;
    if (my_temp > Tthr)
    {
        int thid = thread->getTeamId();
        int whoami = thid;
        int size = thread->getTeam()->size();
        thid = (thid + 1) % size;
        while (thid != whoami)
        {
            target = &thread->getTeam()->getThread(thid);
            if ( CHECK( target->getTeamId() ) < best_temp)
            {
                best_temp = CHECK( target->getTeamId() );
                best_thread = target;
            }
            thid = (thid + 1) % size;
        }
        queue(best_thread, newWD);
        while ( CHECK(thread->getTeamId()) > Tthr ) { COOL_DOWN(); }
        return 0;
    }
    return &newWD;
}

Figure 54: atSubmit() of the temperature-aware cilk scheduler
is below a threshold temperature $T_{thr}$, then return the task-descriptor to the thread and start executing it (xiii). If however, the current threads temperature is warmer than $T_{thr}$ then start cycling through all the remaining cores (vi). Inside the loop, get the pointer to the next thread (vii) and check its temperature (viii). If the target threads temperature is cooler than the current temperature existing in the 'best_temp' variable, then make this threads the currently coolest threads and reflect this change in the variables (ix and x). After the loop has cycled through all the values then the task-descriptor will be enqueued (xi) at the thread with the lowest temperature (the best_thread with the temperature best_temp). Since the thread who originally received the submit call still is too warm to execute, it will cool itself down until the temperature is lower than the threshold (xii).

To implement the second condition, the atIdle() function is modified. The code is shown in figure 55. Every time a core is idle, it will call this function looking for work. And to ensure that the temperature of the core is low enough to actually execute work, (i) it will cool the CPU down until the temperature condition is satisfied ($T_{cur} < T_{thr}$). Once this is fulfilled, the remaining part is identical to the original Cilk scheduler described in section 17. The flowgraphs for the Temperature-aware schedulers atIdle() and atSubmit() functions are shown in figure 56 57.
Figure 56: atSubmit() flow-graph for the Temperature-aware scheduler

Figure 57: atIdle() flow-graph for the Temperature-aware scheduler
Part VII

Results

20.1 Metrics

The two metrics used to compare the scheduling performance in terms of temperatures are based on the time each core spent above $T_{Thr}$.

- Total amount of time $t_{crit,i}$ a core spends above $T_{Thr}$
- The sum $T_{impact,i}$ of all the time a core spends above $(T_i - T_{Thr})$

Metric one can be calculated as $t_{crit} = \sum t_{crit,i}$ and metric two as $T_{impact} = \sum T_{impact,i}$. The impact factor is squared to reflect the fact that most negative effects occurring due to temperature are exponential, and therefore higher temperatures are more dangerous.

20.2 Benchmark results

The results of all three benchmarks can be seen in figure 58. The overall view is positive in that the temperature-aware scheduler does improve a lot compared to the Cilk and Breadth-first scheduler in all but the multisort benchmark.

Why the improvements are close to none in the multisort benchmark can be attributed to the relative small parameters being used, as well as the allowed tree depth. Another parameter that has a perhaps more significant impact on how the programs execute is the simulation checkpoint. In order to be able to deterministically simulate systems, Simics allow users to create checkpoints of a certain state of the system. These checkpoints can later be reloaded to allow determinism in simulation. Simulations done in this work have been executed using only one checkpoint, and this checkpoint might have put some of the program at disadvantage. An disadvantage could be for instance that before the benchmark initializes Ruby, the benchmarks first starts and binds all threads to the available cores and then proceeds to interrupt the simulation, load Ruby, and resume execution at the first task-create instance. However, in some cases,
<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Fibonacci number calculated</th>
<th>Maximum tree depth</th>
<th>$T_{Thr}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first</td>
<td>n = 17</td>
<td>no cutoff</td>
<td>n/a</td>
</tr>
<tr>
<td>Cilk</td>
<td>n = 17</td>
<td>no cutoff</td>
<td>n/a</td>
</tr>
<tr>
<td>Temperature-aware</td>
<td>n = 17</td>
<td>no cutoff</td>
<td>69</td>
</tr>
</tbody>
</table>

Table 5: Fibonacci parameters.

Figure 59: Average component distribution for the three schedulers running Fibonacci.

The main thread manages to reach the break point used to load Ruby before all threads have been activated and thus the threads that are not binded will take a long time to bind due to the extra latency added by the Ruby timing model.

For more simulation details, see subsections 20.3, 20.4 and 20.5.

### 20.3 Fibonacci

Calculating the Fibonacci’s series is generally a very lightweight task; not much calculation is done giving low CPU power usage and close to none data manipulation is done, thus reducing caches. The parameters that the Fibonacci program was run using are shown in table 5.

The average power-breakdown of the individual components can be seen in figure 59 for the different schedulers.

Figure 60 shows the difference between the temperature-aware scheduler and the others. The temperature-aware scheduler only allows for short spikes above the $T_{Thr}$, after which it immediately starts re-scheduling of tasks to cooler threads and begins the self-cooling loop.

Figure 61 shows how many re-schedules were performed to prevent overheating and, if compared with the average temperature of all the cores in figure 62, one can clearly see a correlation between the average temperature across the cores and the number of re-schedules done. In this particular case, core #6 performs the most re-scheduling. Core #6 is also the core that spends the most amount of time above $T_{Thr}$.

In figure 63 the execution time, the average power consumed and the total energy for this benchmark running on the different schedulers is shown. Re-
Figure 60: Cores spent above threshold temperature during the benchmark

Figure 61: Task re-scheduled using the temperature-aware scheduler

Figure 62: Average core temperature with the different schedulers.
Figure 63: Execution time, power consumption and energy cost of Fibonacci

Figure 64: Warmest and Coolest cores running Fibonacci on the three schedulers.

Remarkably, the Temperature-aware scheduler outperforms the Cilk scheduler in all three aspects. The Breadth-first scheduler consumes the least amount of power, but given the slow execution time, the total energy to complete Fibonacci is higher compared to the other schedulers.

Figure 64 shows the warmest and coolest core of all the schedulers throughout the benchmark. The difference between the warmest and the coolest core is the least in the Temperature-aware scheduler, giving the results which was sought after: to decrease hotspot and gradients. Here it is also clearly shown how equal the Cilk and the Temperature-aware scheduling temperature pattern are. The Breadth-first scheduler lacks any kind of re-scheduling or stealing mechanism and seem to have a much less “bursty” temperature characteristic.
<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Board size</th>
<th>Maximum tree depth</th>
<th>$T_{Thr}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Cilk</td>
<td>7x7</td>
<td>6 stages</td>
<td>n/a</td>
</tr>
<tr>
<td>Temperature-aware</td>
<td>7x7</td>
<td>6 stages</td>
<td>69</td>
</tr>
</tbody>
</table>

Table 6: Queens parameters

Figure 65: Average component distribution for Cilk and the Temperature-aware scheduler running queens

### 20.4 Queens

Queens benchmarks includes more computations and more data-manipulation compared to Fibonacci. The parameters are shown in table 6.

Fig 65 shows the average power distribution of each of the components of the Temperature-aware and the Cilk scheduler. They have roughly the same distribution between the four components.

Figure 66 shows the time spent above $T_{thr}$ in the different schedulers. Again the Temperature-aware schedulers impact upon temperature is clearly seen. Once the hotspots are detected, the Temperature-aware scheduler immediately cools-off decreasing the current cores temperature while the Cilk scheduler, unaware and unknowing of the temperature continues to increase.

Figure 66: Cores spent above $T_{thr}$ temperature during the queens benchmark.
Figure 67: Task re-scheduled using the temperature-aware scheduler

Figure 67 shows the effort done by the temperature-aware scheduler to minimize hotspots. Again we compare this to the average temperature of each core in figure 68 and see a correlation between the two.

Figure 69 shows the execution time, the power consumption and the energy the benchmark consumed. The difference in execution time is negligible but this leads nonetheless to an improved power consumption and energy drain in favor to the Temperature-aware scheduler.

The two warmest and coolest cores for each scheduler are shown in figure 70. The Temperature-aware scheduler has more balanced temperature-difference between the cores compared to the Cilk scheduler. This is due to the temperature-schedulers ability to keep the temperature below $T_{thr}$.

20.5 Multisort

Multisort is a data-intensive benchmark, sorting an array in parallel. The parameters passed to this benchmark are seen in table 7.

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Array size(bytes)</th>
<th>Serial cutoff size</th>
<th>$T_{thr}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first</td>
<td>65536</td>
<td>1024</td>
<td>n/a</td>
</tr>
<tr>
<td>Cilk</td>
<td>65536</td>
<td>1024</td>
<td>n/a</td>
</tr>
<tr>
<td>Temperature-aware</td>
<td>65536</td>
<td>1024</td>
<td>68</td>
</tr>
</tbody>
</table>

Table 7: Multisort parameters

Figure 71 shows the typical distribution of component power consumption in all the three schedulers. When running Multisort, the Breadth-first schedulers
Figure 68: Average core temperature with the Cilk and temperature-aware scheduler running queens.

Figure 69: Execution time, power consumption and energy cost of queens.

Figure 70: Warmest and Coolest cores running queens on the temperature-aware and cilk scheduler
L1 Cache activity corresponds to half of all power consumed. This is most likely due to no work stealing mechanism giving more cache hits.

Figure 71: Average power distribution when running multisort

Unlike the previous benchmarks, the temperature does not improve against the other two scheduler. This is most likely attributed to the fact that the checkpoint used when simulating puts the Temperature-aware simulator at disadvantage in one way or the other. The spikes above $T_{Thr}$ are shown in figure 72 and the tasks that were re-scheduled to avoid overheating are shown in figure 73; compared with Fibonacci, the number of tasks rescheduled is less. This might happen when the task-depth (before starting serial execution) is too low since there are no OpenMP constructs that are called within the sequential region and no temperature control is being done.

Figure 72: Time above the threshold

Figure 74 shows the average temperature across all cores running the benchmark. Note how core number 5,6 and 9,10 always have higher temperature. This is because those cores are the middles cores, where much of the power dissipation from the surrounding cores are spread through.

The time, power consumption and energy cost relative to the temperature-aware scheduler of the benchmark is show in figure 75. The only positive effect the Temperature-aware scheduler have is the power consumption, which is lower compared to all the other schedulers.
Figure 73: Re-scheduled tasks using multisort

The difference between the warmest and the coolest core is shown in figure 76. The breadth-first manages to keep all the cores reasonable close to each other while the Temperature-aware schedule do the worse job in smoothing them out.
Figure 74: Average core temperature for multisort

Figure 75: Scheduler absolute time, power and energy cost
A system capable of calculating, tracing and returning temperature have been constructed and verified functional. Using this system, a primitive temperature-aware scheduler was created and trialed against existing schedulers. The results were positive in respect to temperature and energy consumption. In two of the trials the Temperature-aware schedulers improved on temperature against the competing schedulers. This is significant in terms of possible future temperature-aware exploration using user-level parallel programming models, such as the OpenMP model used here. Future schedulers could perhaps include a construct specifying the maximum allowed temperature for a give region on the chip; together with other constraints, such as locality aspects or deadlines, this would create an extremely versatile system capable of working on a range of different systems with different restrictions or operation conditions. Extending this idea to not only include core-level chip control, but also be aware of other functional units of the system would create a whole new dimension for programmers to explore, and hopefully the life-time and chip cost could potentially decrease.
Part IX
Future work

20.6 Simulation platform
While the simulation platform created for this study fulfills the role to evaluate different scheduling policies in terms of temperature, it also suffers from many flaws that could be improved.

CPU power consumption should be modeled in more detail to find out exactly how the power flow goes through the CPU. This is a task not easily accomplished using a functional simulator, so other methods should be considered. Information about the CPU power consumption details are scarce, and most existing methods involve FPGA prototyping or RTL simulation for acquiring power values.

The cache power system should be more detailed as well, covering power consumed by directories which the current system does not account for.

While routers are very accurately modeled here using the Orion simulator, which has been evaluated against existing industry solutions, links aren't taken into account due to them being located outside the the tile's area. Links in a system can consume respective amount of energy due to them being long wires across the chip.

Assuming all mentioned is taken into account, a more correct floor-plan should be used to provide the hotspot modeling tool the best available information on the location of these blocks.

The cache coherence protocol should have an impact on the power consumed in the system, and thus on the temperature. Future studies would include comparison of different protocol and their impacts and differences.

20.7 Power regulation control
This work has implemented power regulation control in the form of that the instruction 'nop' doesn't use up any power. Future platforms should include more versatile control such as frequency scaling or halting cores temporarily, and also make this reflect performance of the system.

20.8 Scheduler
The scheduler trialed here is an extremely simple, reactive temperature aware scheduler. It has been shown to work but with performance overheads, as seen in the multisort scenario. A future scheduler would include some kind of prediction scheme, where future values could be predicted from current ones, such as a curve-fit model or an auto-regressive model. Location of threads and data should also be taken into account to minimize link traffic.

Scheduler should also be able to migrate threads to other cores. This was unexplored in this work but is a well-known strategy for power control in other...
studies.
References


[20] Ichoon Yeo Eun Jung Kim. Temperature-aware scheduler based on thermal behaviour grouping in multicore systems.


