# High level thermal-aware scheduling for multiprocessors

Jin, Cui

2012

JIn, C. (2012). High level thermal-aware scheduling for multiprocessors. Doctoral thesis, Nanyang Technological University, Singapore.

https://hdl.handle.net/10356/54996

https://doi.org/10.32657/10356/54996

NANYANG TECHNOLOGICAL UNIVERSITY

# High Level Thermal-Aware Scheduling for Multiprocessors

**Cui Jin**

**School of Computer Engineering**

A Thesis Submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy

**2012**

# Acknowledgements

First, I would like to express my deeply heartfelt thanks to my supervisor, Associate Professor Douglas L. Maskell, for his intellectual guidance, constructive and patient direction as well as his unceasing attention, support and encouragement. It is he who led me into a new wonderful palace of science, which made me enjoy the pursuit of the Ph. D. all the time. I wish to thank him for the open, interactive and relaxed academic environment he provided for me. In particular, I would like to thank him for all his kind consideration and care for me in my daily life and all the special time and effort he has spent to support me to fulfil the Ph. D. What I benefited from him has contributed much to my current career and I believe it will assist me more in my way ahead.

I would like to thank the staff and the students in the Centre for High Performance Embedded Systems for their kind help for my research work and all the happy days I have had with them.

I would like to acknowledge the School of Computer Engineering, Nanyang Technological University for providing the research facility and the research scholarship.

I wish to express my thanks in my deep heart ocean to my fiancée, Zheng Yuanyuan for her consistent care and support. Her words are always companying with me to shrink the tough time and double the happiness in my life.

Last but not least, I want to thank my family in China, for their constant love and encouragement. Particularly, my parents and grandma are always caring and loving me. They unceasingly input encouragement, comfort and support to me. I can do nothing to express my deep gratitude to them but dedicate the thesis to them.

# Table of Contents

# List of Figures

# List of Tables

# Abstract

Power and thermal issues are primary design constraints in both stationary and portable computing devices. Adverse thermal issues can impact microprocessor performance, including computational speed degradation, aging, and unreliable system behaviour. These situations are exaggerated in current state-of-the-art multiprocessors due to their high power density and the thermal coupling between cores. High level thermal-aware scheduling (TAS) is seen as one possible solution to optimize and control on-chip temperature. However, after performing an extensive review of the literature, a number of shortcomings in current high level TAS implementations have been identified. These include, the inaccuracy of thermal sensor readings, low computational efficiency of existing time-triggered thermal simulators, oversimplified thermal and leakage power models currently used at the system level, lack of appropriate thermal constraints used in scheduling analysis in hard real-time embedded systems and a lack of appropriate fine-grained dynamic TAS (DTAS). These shortcomings have provided suitable motivation for the work described in this thesis, which includes the following contributions:

- A fast event-driven look-up table (LUT) based thermal estimation approach is developed. We introduce the concept of power events which capture the significant power changes on-chip. These power events induce a temperature change which can be easily obtained using the pre-calculated LUTs (representing the thermal response of a unit power input). We show that these thermal responses, induced by individual power events, satisfy the superposition principle and can be accumulated to evaluate the thermal map when any event occurs. We also define the necessary optimizations and operations for the LUTs. Experimental results show our LUT method is accurate, producing thermal estimations of similar quality to an existing open-source thermal simulator (HotSpot), while providing 2 to 3 orders of magnitude reduction in computational complexity.
- We use our fast LUT approach to analyze the offline schedulability for a real-time task set on a simulated multiprocessor system under a strict (hard)

thermal constraint. This is very useful for reducing the risk of overheating in safety-critical embedded systems. Our schedulability test can also be used as a framework to optimize other goals (e.g. maximizing the performance and minimizing the peak temperature). We show that we are able to schedule large task sets (up to 50 tasks) in reasonable time (less than 12 minutes), which is 2 to 3 orders of magnitude faster than using scheduling with existing thermal simulation tools.

- For high power multiprocessor (or many-core) systems, it is not possible to ignore the temperature-leakage power dependence. Therefore, we modify the LUT-based approach to include a temperature-dependent leakage power model. The leakage power calibration enables us to accurately predict the near future thermal map without needing to resort to a computationally expensive iterative approach. Based on this prediction, we develop several heuristic policies for dynamic TAS on a simulated many-core system. We show that our proposed predictive policies are significantly better, in terms of minimizing average/peak temperature, reducing the dynamic thermal management overhead and improving other real-time features, than existing TAS schedulers, making them highly suitable for heuristically guiding thermal aware task allocation and scheduling.

# Chapter 1
# Introduction

Microprocessor performance has been one of the primary design goals, with performance scaling to follow Moore's law over the last few decades. To facilitate these performance advances, the transistor gate length and the dioxide thickness have been reduced year on year, thus allowing more and more complex functional units to be integrated into a much smaller area. This dimension shrinking in the modern integrated circuit (IC) manufacturing process brings a number of advantages such as smaller signal delay on shorter wires, lower core voltage, shorter signal edge, smaller chip area and more I/O pins on package. However, the function complexity and the number of transistors are becoming incredibly high, resulting in a number of disadvantages, including extremely high power density, increased design time and increased validation complexity. The high power density results in increased energy consumption and chip temperature.

Power/thermal issues in uniprocessor system have been intensively studied by both academia and industry [15][45][48][50][55][56][57][58]. The thermal effects introduced by the high power density on chip are unavoidable, with heat fluxes (or power densities) in state-of-the-art microprocessors currently being in the range of 200-300W/cm$^2$ [1], and are expected to continue to increase. Power/thermal effects can impact the following critical metrics:

- Computational Speed Degradation: A higher temperature may degrade the computational speed due to the current leakage induced by carrier migration at the physical level [5][53].
- Reliability: Operating the processor at above the thermal designed power (TDP) could induce a timing sequence error and even physical damage. A timing error is recoverable without destroying the processor, but physical damage, such as a transistor fusing, is permanent. Current commercial processors usually embed thermal sensors into the core to monitor the temperature and control the temperature to within a safe threshold [102][103].

- Accelerated Aging: High temperatures can accelerate aging and shorten the life span of the processor. The metal components and wires on die can ablate in a long-term hot environment. As such, both the reliability and lifespan are reduced [102].
- Cost: Better packaging techniques and external cooling devices for heat dissipation increase the system cost dramatically. The processors used in embedded systems generally depend on natural convection, using the heat spreader and package to passively remove excess heat, while desktop and server processors depend on active cooling devices [27][35].
- Power: Higher temperatures increase the leakage currents which contribute to higher static power consumption. The increased static power results in an increased temperature, resulting in a positive feedback relationship between static power and temperature, which can result in processor failure due to "thermal runaway" [77].
- User Perception: Higher temperatures can result in an uncomfortable feeling while using the devices. This is particularly the case with portable devices.

About a decade ago, processor designers realized that reducing the transistor size was not going to produce significant frequency gains, as occurred previously, due to adverse power/thermal effects. This launched the multi-core era, as the best way to efficiently utilize the available silicon was to duplicate multiple processors on a single die. However, this does not solve the thermal problems. In fact, the ITRS [1] have identified power and thermal issues as a major design constraint and bottleneck for current and future computing devices.

In the last decade, processor manufactures have introduced a number of multi-core processors for mainstream server and desktop market. For example, the modern Intel Xeon processor, targeting the server market, has up to 10 processing cores [113]. ITRS [1], reproduced here as Figure 1.1, predicts that the number of CPUs will hit 50 by the 2020s, if not sooner.

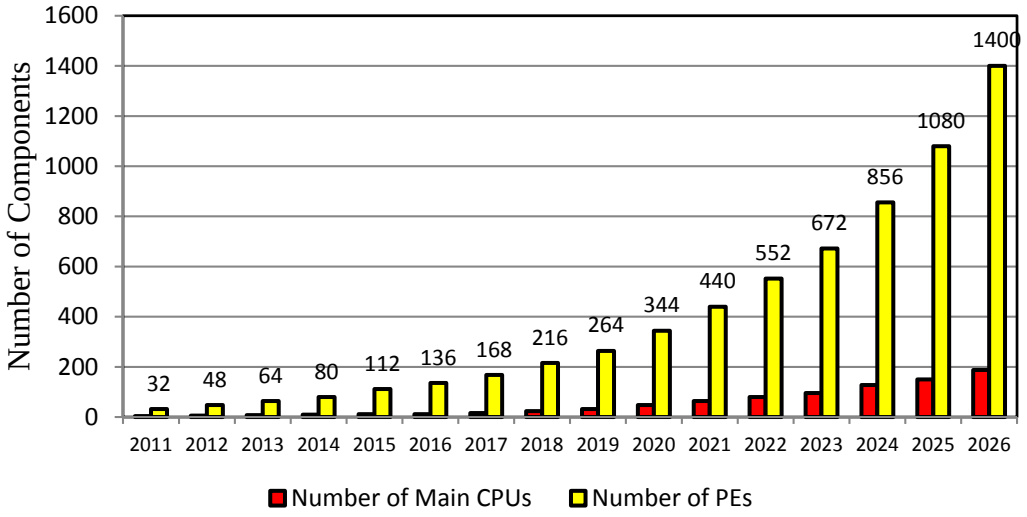



Figure 1.1: Consumer stationary design complexity trends [1]

## 1.1 Power and Thermal Optimization

Power and thermal optimization in microprocessors[1] has been extensively studied [2][3][4][5][6][7][8][9][10][11][12][13][14][15][16][17][18][19] and is applied at different levels in the design flow. Different approaches and techniques are applicable at each level. At the lower levels (e.g. the physical [2][3][4][7][8][9][10], logic and register transfer levels [15][17][18][19]), the techniques used are relatively mature [1] and include: new materials (e.g. High-K dielectric [5] or GeA compounds [1]) and new processes (such as, tri-gate [6], FinFET [53] and 3D-ICs [11]). The EDA tools used for design at these levels also provide support for power/thermal-aware synthesis, mapping [12][13][14], placement and routing [7][8][9].

Techniques for power/thermal management applicable at the micro-architecture and architecture level include: dynamic voltage and frequency scaling (DVFS) [21], clock gating [21], pipeline gating [21], stop & go policy [21], I-cache toggling [20][21]. There techniques are usually adopted on a global or per-module basis.

At the higher levels (e.g. the algorithmic and system level), compiler optimization techniques [22] and scheduling [21][69][72][81][82][88][89] can help to alleviate the power/thermal issues in a high-level (abstracted) way. In fact, according to the ITRS

[1] We use the term multiprocessor in our research to refer to multiple processors, integrated onto a single silicon die and tightly coupled to each other.

[1], system-level power/thermal management and optimization will be confronted with tremendous opportunities and challenges in the coming decades. The ITRS predicts that the higher design levels will play an increasingly important role in achieving the required levels of system power minimization, as shown in Figure 1.2.

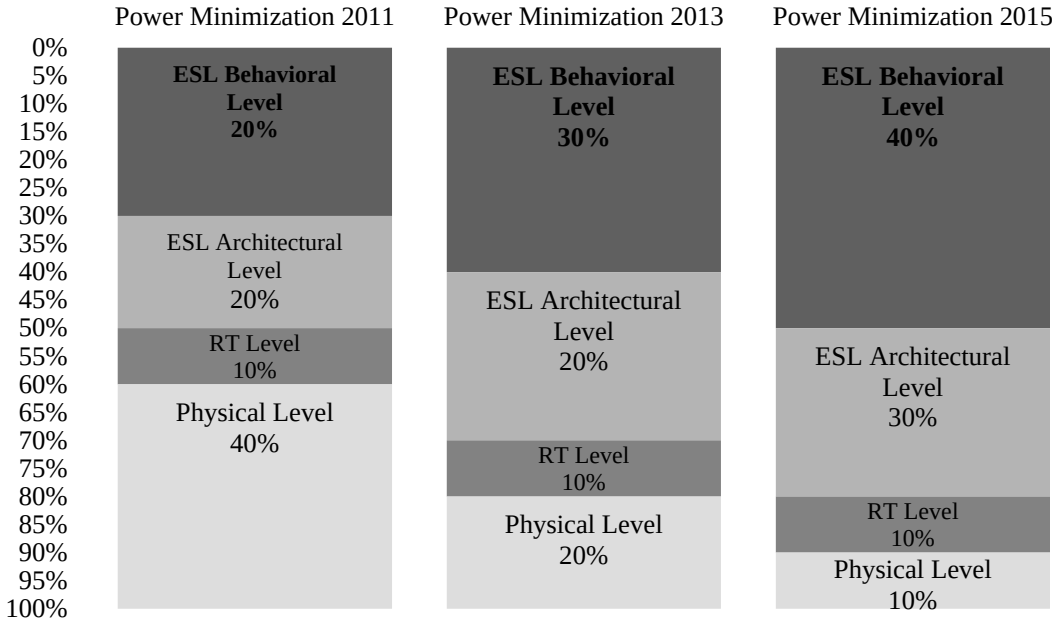



Figure 1.2: Evolving Roles of Different Design Levels in Overall System Power Minimization [1]

The increasing power densities associated with transistor scaling and the trend towards more and more processors on a single IC are likely to significantly impact on reliability and performance [1], while decreasing supply voltages worsen leakage currents and noise [1]. These trends will require power optimizations that simultaneously exploit techniques at all levels in the design process, as well as at the operating system and application software levels.

## 1.2 Motivation of the Research

Increasing the power/ thermal optimisation efforts at higher design levels will produce more efficient circuits, which when mapped to the lower level could obtain better results than are possible for optimizations at the lower levels only. Higher level design impacts the performance of the whole system, implying that an improvement at the high-level could lead to significant overall savings.

Thus, examining high-level power/thermal optimisation for multi-core and many-core systems would appear to be an important and timely endeavour. In particular, high-level thermal optimisation and management for delivering power/thermal improvements at the system level would provide an appropriate focus for my research. However there are a number of challenges, or research questions, that need to be addressed. These include:

- Current thermal management techniques (e.g. DVFS, power gating, etc.) are coarse grained (with a relatively long time response from an OS scheduling point of view) and they are reactive (in that the system responds to a system event, such as the thermal threshold being exceeded). *Will a fast proactive thermal-aware scheduling technique, such as thermal-aware scheduling (TAS), provide a better option for system-level thermal management?*
- Current TAS techniques fall into one of two general categories:
    a. Relatively fast with a very simple, but inaccurate thermal model due to the thermal coupling effect between cores [65][92].
    b. Relatively slow due to a thermal simulator with an accurate but much more complex thermal model.

    In a multi-core/many-core TAS scenario, and particularly for dynamic TAS (DTAS), both speed and accuracy are important. The scheduling analysis in static TAS (STAS), which is an NP-hard problem, would also benefit from an accurate but faster thermal estimator, particularly with the expended search space associated with multi-core/many-core systems. *Is it possible to propose a technique for thermal estimation which is able to run orders of magnitude faster than a thermal simulator (such as the widely used HotSpot simulator [29][35][72][43]) while providing similar accuracy?*

- Leakage power is an important consideration in current IC design, and will become more and more significant with transistor scaling. *Is it possible to extend a fast thermal estimator to include a leakage power model while still maintaining accuracy?*

If these questions can be answered, STAS and DTAS could be used to more effectively provide high-level thermal optimisation and management in multi-core

and many-core systems.

## 1.3 Thesis Organization

Chapter 2 presents a detailed literature review where we analyse the strengths and weaknesses of the existing research in the area of power/thermal-aware management and scheduling.

In Chapter 3, we develop a fast and accurate technique for thermal estimation, applicable to multi-core/many-core systems. The technique is based upon event driven power changes and uses a fast look-up table (LUT) to perform the thermal estimation. The LUTs are prebuilt and use the same non-leakage model as HotSpot, and as such, have the same steady-state accuracy, but with a significantly reduced computational overhead. A number of primary definitions and LUT operations are introduced.

Chapter 4 applies this LUT-based approach to static thermal-aware real-time task scheduling in low power multi-core/many-core systems. In this scenario, the effect of leakage power is less significant and to some extent can be ignored. A forward search, which gives the minimum-time schedule of the task set for a given thermal constraint, is introduced. A heuristic peak temperature minimization algorithm is also developed. Experiments using both real and synthetic real-time benchmarks show that we are able to schedule large task sets (up to 50 tasks) in reasonable time (less than 11 minutes), which is 2-3 orders of magnitude faster than using scheduling in conjunction with the Hotspot thermal simulator.

Chapter 5 examines DTAS. It firstly extends the simple LUT-based thermal estimator developed in chapter 3 to include a temperature dependant leakage power model. To minimize the overhead, while maintaining the estimation accuracy, prebuilt look-up-tables and predefined leakage calibration parameters are used to speed up the thermal solution. Based on this refined fast LUT-based thermal estimator, a number of heuristic DTAS algorithms are developed for high power multi-core/many-core systems. In this scenario, the effect of leakage power is much more significant and cannot be ignored. We show that our proposed DTAS policies are better able to

minimise the average/peak temperature than existing DTAS schedulers, making them highly suitable for heuristically guiding thermal aware task allocation and scheduling. We are able to reduce the dynamic thermal management overhead (by 3 orders of magnitude compared to using HotSpot) while maintaining comparable accuracy.

Lastly, in Chapter 6, we summarize and highlight the contributions of this work and discuss possible extensions and other future work.

# Chapter 2

# Power/Thermal-Aware Management and Scheduling

The current research examining power/thermal-aware management and scheduling in microprocessors is introduced and discussed in this chapter. Based on this review of the literature, we are able to summarize the high level thermal management and optimization process, as shown in Figure 2.1.

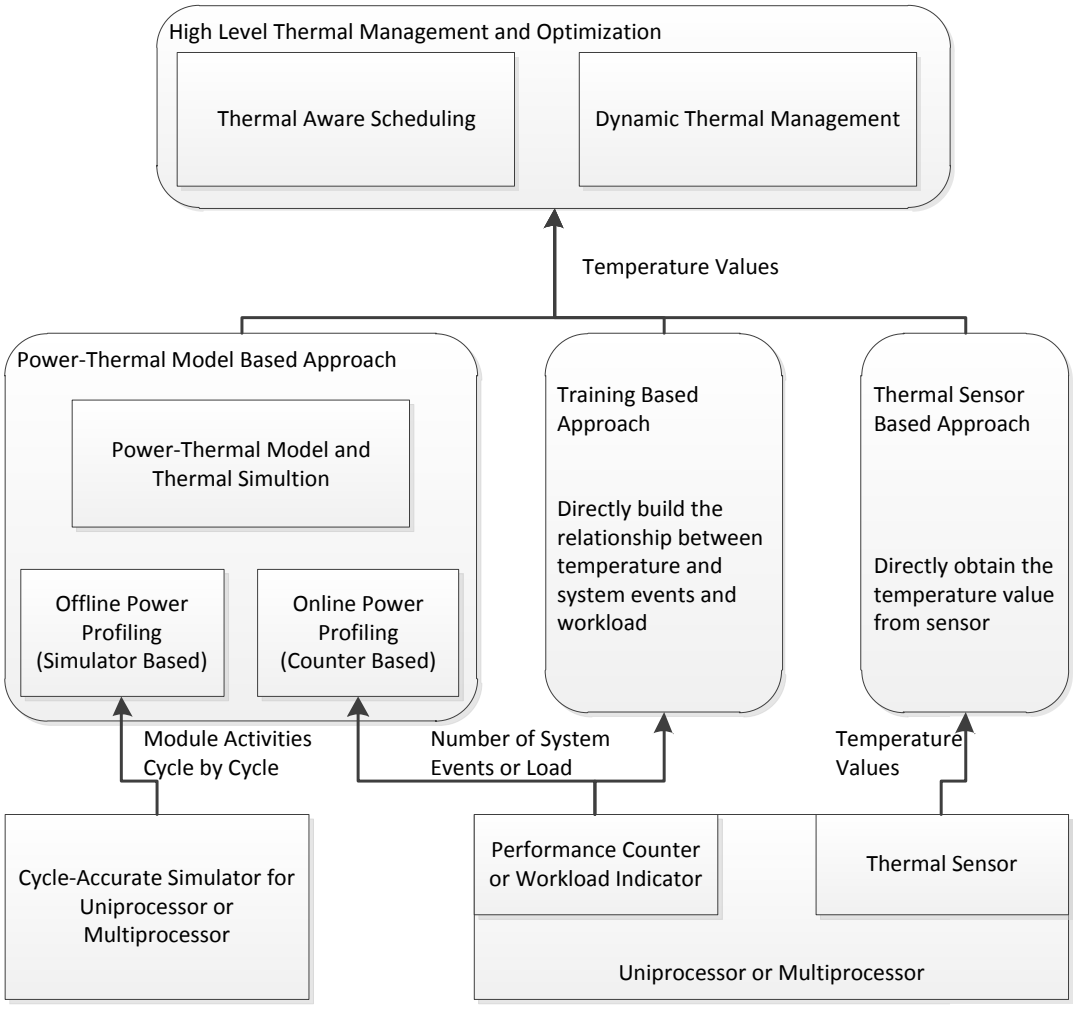



Figure 2.1: The high level management and optimization process

Thermal aware scheduling (TAS) and dynamic thermal management (DTM) are two common techniques used in high level thermal optimization. In general, TAS actively works with the OS scheduler, i.e. every time the scheduler is invoked (e.g. on the

occurrence of a timer tick interrupt in Round Robin scheduling, at the end of a system call or on the occurrence of an interrupt, etc.), to determine where to allocate (or migrate) a task according to some temperature criteria (e.g. allocate to the core with the lowest temperature or the lowest temperature gradient). Usually, TAS works in a relatively fine-grained time increment[2] while DTM is relatively coarse-grained and is passively triggered when the temperature exceeds a predefined threshold (resulting in the scaling down of the overheated core's frequency or putting the core into a low power sleep state).

Even though these two mechanisms are different, both of them can reduce or inhibit the occurrence of adverse thermal effects on-chip. Moreover, both techniques need to know the temperature distribution on-chip for proper operation. Therefore, capturing, monitoring or estimating the temperature is a vital step for high level thermal management and optimization.

In terms of TAS, there are two main categories in the research domain: static-TAS (STAS) and dynamic-TAS (DTAS). In STAS, thermal-aware scheduling is carried out offline, usually in design stage. STAS uses a pre-implemented power/thermal model and thermal simulator to schedule a set of tasks and simulate their corresponding thermal behaviours, without the need for real thermal sensors. DTAS, on the other hand, needs to track the temperature changes on-the-fly and schedule a task in, or near to, real time. Therefore, DTAS requires a fast thermal simulator, real-time thermal sensor information, or both, as the input. The various TAS techniques will be discussed in more detail in Section 2.4.

We classify the temperature measurement or estimation into three classes, as indicated in the middle part of Figure 2.1. These methods include: on-chip thermal sensor(s), training-based models that combine thermal sensor readings with the occurrence of system events; and temperature estimation based on a power profile measured using either a simulation based approach or a performance counter based approach. All of these methods can provide chip/processor/module temperature information to the higher level. However, each method has its own strengths and weaknesses, and will be discussed in more detail in Section 2.1 and 2.2.

[2] The typical TAS interval is usually that of two consecutive time ticks in the OS.

The procedure for high level thermal optimization and management can be summarized as in the following steps:

- STAS
    a. Perform static power profiling for tasks.
    b. Predetermine a schedule that meets the task constraints (e.g. deadline)
    c. Estimate the temperature induced by the task based on a power-thermal model and the results from the thermal simulator.
    d. Determine if the schedule meets the temperature constraints or if temperature optimal metrics are achieved
    e. Iterate steps b, c and d if the desired thermal threshold is exceeded.
- DTAS and DTM
    a. Perform static power profiling for tasks or dynamic power profiling using performance counters. This step is not necessary for a training-based approach or when using direct temperature sensor readings.
    b. Estimate the temperature based on a power-thermal model, a thermal simulator or a training-based approach, or direct temperature measurement from thermal sensors.
    c. Schedule the task based on the temperature estimation or thermal sensor reading.
    d. Repeat steps b and c at runtime.

In the following sections, we introduce and discuss the literature relating to DTM and TAS (including power profiling and thermal measurement/estimation). The strengths and weaknesses of the various techniques are highlighted and are then used to form the basis for the development of the research focus outlined in subsequent chapters.

## 2.1 Power Profiling

The power consumption of the processor is composed of two main parts: the dynamic power consumption induced by the charging and discharging of the capacitors during signal switching and the static power consumption due to the inherent gate leakage and sub-threshold leakage effects in the process technology. The proportion of the

static power relative to the total power consumption increases with decreasing transistor feature size. Currently, the static power represents a very significant proportion of the total power and cannot be ignored, and therefore, high level power modelling and analysis must represent both these aspects (Equation 2.1).

$$P = P_{dynamic} + P_{static} \tag{2.1}$$

For the dynamic power, an analytical expression for a single transistor is given by Equation 2.2:

$$P_{dynamic} = CV^2 f \tag{2.2}$$

where $C$ is the equivalent parasitic capacitance, $V$ is the supply voltage and $f$ denotes the frequency of signal switching. At the system/architectural level, a functional unit is composed of a large number of transistors with many different signal inputs. As such, simulating large scale circuit activity using the single transistor model will result in a significant computing overhead. As a result, most high-level power analysis and estimation techniques use a module-based approach that stores the different categories of power related parameters (e.g. the signal switching probability, determined by performing a number of simulations on a functional unit, and the equivalent parasitic capacitance, determined by performing circuit and gate level simulation, such as using HSPICE [37]) for each module at the architectural level. The total dynamic power can then be estimated based on the resource utilization and the power parameters. Since the power parameters are predetermined constants, the only variable is the switching frequency which can be significantly affected by different applications and their processed data. For instance, some computation-intensive tasks require more ALU or FPU resources, while some data-access-intensive applications occupy the load/store queue and memory controller more frequently.

On the other hand, the static power consumption is related to the inherent transistor parameters (e.g. gate length, leakage ratio, voltage etc.), and is not affected greatly by the application. Therefore, power profiling firstly examines the resource utilization for a task to estimate its dynamic power consumption, and then adds the static power to determine the total power consumption. Both online and offline power profiling can estimate the resource utilization for the dynamic power component.

### 2.1.1 Offline Power Profiling (Simulator-Based)

Offline power profiling, also called simulator-based profiling, uses a simulator to estimate the power. The simulator can model the different activity details of a specific piece of hardware. For example, a circuit simulator emulates the signal switching at gate and wire level; a register-transfer-level (RTL) simulator can track the input/output and register values for every logical component (e.g. adder, multiplexer, flip-flop and so on); and a micro-architecture level simulator can emulate the status (e.g. input/output values, idle or busy) for each functional unit on a cycle-by-cycle basis.

Different simulators can provide different details of the signal activities and capacitance estimation, as such, affecting the accuracy of power profiling. A low level simulator has the best accuracy, but also requires a huge computation overhead, particularly if simulating a complex module. For example, while it may be possible to carry out a circuit level simulation of an entire processor, the computational resources and time required make this infeasible. Even at the higher register-transfer-level, a simulation for a common application may last for several days on a high performance workstation. Such a huge overhead is not acceptable for higher level optimization, especially for the on-line case.

The highest level of simulation is referred to as behavioural or functional simulation. This kind of simulator only analyses the binary instructions one by one in the execution file, and simulates their logical (semantic) outcomes and records the register and memory status in logical (semantic) order. This level of simulation can only reflect the logical correctness of a section of code, and neglects all the activities related to the realistic implementation of the instructions in a processor. Therefore, it is applicable for debugging software or testing logical behaviour of a code section, and is often used in the early developmental stages when a real hardware platform is unavailable. It is not suitable for any complex architecture, such as out-of-order execution (superscalar), branch prediction and speculation. This level of simulation has a relatively low computational overhead, and is able to emulate a full-system with a fully functional OS. A behavioural simulator concentrates on the logic, rather than on the implementation, and is not able to obtain the necessary cycle-by-cycle information relating to module utilization to provide an accurate power profile.

QEMU [38] is a very popular open source full-system emulator that supports a range of platforms (e.g. X86 PC, ARM Cortex-A9 PandaBoard, etc.). Simics [39] is also widely used in academia for multicore functional simulation, but it only supports simple in-order pipeline architectures. These simulators also neglect the implementation of the real processor, and as a result are unsuitable for power estimation.

GEMS [40] is a well-known execution-driven multiprocessor simulator, which can provide full system simulation. GEMS integrates with Simics, an in-order processor behavioural simulator, and Ruby, a customized memory subsystem simulator. However, Simics emulates the instruction behaviours one by one without properly simulating the processor stalls caused by inter-instruction dependency (e.g. data hazards3), and it only captures the stalls caused by the memory requests4 (e.g. the memory instruction latency simulated by Ruby, such as L1 and L2 cache hit and miss). As such, GEMS is unable to provide an accurate power estimation which reflects the realistic power consumption. Subsequently, the GEM5 implementation integrated M5 [114] as the processor simulator. This supports out-of-order superscalar simulation, but doesn't support the shared-memory-based pthread library for multi-threaded applications or provide a power profiler.

A micro-architecture (MA) level simulator simulates the activities and status (e.g. input/output, idle/busy) for the functional units (e.g. load/store queue, instruction fetch and decoding unit, ALU, register renaming units and so on). It reflects how these functional units work together and intercommunicate (e.g. internal bus, data path and instruction path) with each other in each cycle. The most important information obtained from a MA simulator is whether one unit is busy or idle in each cycle. As such, the cycle-by-cycle resource utilization is profiled so as to estimate the power consumption of an entire functional unit as a simulation object, rather than the detailed signals and circuits inside the functional unit. As a result, the MA simulator can obtain the cycle-accurate information for the functional units, while keeping the simulation overhead as low as possible. Thus, it gives a good trade-off between efficiency and accuracy.

[3] This refers to data dependency among instructions.
[4] Memory requests include all memory access instructions (e.g. load and store, stack pop and push etc.) accessing the data cache and all instruction fetches to the instruction cache.

SimpleScalar [41] is one such MA simulator, used for architecture design and research. It originally supported the simulation of a single-core Alpha processor with out-of-order superscalar, and was subsequently extended to support the single-core PowerPC and ARM processor (referred to as SimpleScalar-ARM). However, SimpleScalar only simulates a uniprocessor without implementing the privileged instructions[5] used by the operating system. Moreover, it only supports simple system calls, with many of the critical system calls (e.g. process creation/switching (e.g. fork) and inter-process communication (IPC)) not being simulated. As such, multi-thread/process applications or operating systems cannot be simulated with SimpleScalar.

Brooks et al. [42] developed a toolset PowerTimer for use in early stage, micro-architecture level power and performance analysis of microprocessors. The main component of the toolset, Wattch [43], is an extension of SimpleScalar. Wattch is a set of parameterized power estimation functions (accumulating the resource utilization to get the power estimation) that can be integrated into SimpleScalar or any other cycle-accurate micro-architectural simulator. However, Wattch, is a uniprocessor power estimator and is not suitable for the multiprocessor case.

Eisley et al. [44] examined the high-level power analysis for CMP and MPSoC, which are both more complicated than the uniprocessor case as they must also consider the power consumption for intercommunication between cores. A power estimator called LUNA (link utilization for network power analysis) is used for estimating the power consumed by the NoC intercommunication network, which is faster than Orion (a bus simulator) and is able to maintain a good relative accuracy. In terms of the NoC power estimation, the utilizations of the key parts of the router (e.g. write/read buffer, crossbar and the (four directional) link) are recorded. However, only the utilization of the link is used (as a proxy for all the other parts), as ignoring the detailed power components improves the performance of the power estimator while still maintaining a good relative accuracy. Another contribution is that the network graph, used for tracking the utilization of the various functional units, allows a segment of code to be walked through for obtaining the functional unit utilizations. However, as the network graph cannot correctly analyse the instruction path for out-of-order-execution or for the speculation architecture, its power profiling is less accurate than that of a micro-

[5]Control register instructions (e.g. the switch between supervisor mode and user mode) and co-process instructions (e.g. control page table, MMU, TLB and cache).

architecture level simulator. Moreover, the leakage power in the power analysis is always treated as a constant or even ignored, which can be a source of error, as the leakage power heavily depends on temperature which inversely impacts power consumption.

In summary, even though several simulators are available for academic research, power estimation and profiling for multiprocessor is not mature. These simulators (e.g. Simics and GEMS) simplify the processor implementation and do not provide cycle-accurate information. The inter-core communication in many simulators (e.g. Ruby and LUNA) is based on a message passing interface and is not compatible with the shared memory architecture used by many current multi-threaded applications.

### 2.1.2 Online Power Profiling (Counter-based)

Online power profiling, also referred to as counter-based power profiling, is carried out directly on a real processor while an application is running. Since current processors do not provide runtime power information and no measuring devices are embedded into the package, direct online power profiling is not feasible. If a processor can dynamically provide the statistics of key functional units, the utilization of these functional units can be used for online power estimation. The performance counter in current state-of-the-art processors is a set of registers recording critical system events for different functional units, such as the number of cache hits/misses, the number of ALU accesses, and so on. For example, the performance counters in the older processors, such as the Compaq Alpha 21164 and the Intel Pentium II were able to count monitor 22 and 77 system events [45], respectively. More recent processors, such as the Intel Core-i7 [46] and the ARM Cortex-A9 [47] are able to count 97 and 58 system events, respectively. Based on the number of system events, the utilization of a functional unit in a certain period can be determined and hence the power can be estimated [48][49].

Russ et al. [45] proposed a heuristic utilization estimation based on the number of system events. As the performance counter can only measure a limited number of system events in the same time slot and cannot capture all system events in one cycle,

heuristic formulas are used to estimate the utilization. This work showed that accurate online power estimation using a counter-based approach was possible. In a similar way, Canturk et al. [48] used counter-based power profiling with heuristic estimation, on an Intel Pentium 4 processor. The results showed good correlation between the proposed counter-based profiling and the direct power measurement, and as such the procedure has been adopted by other researchers [49].

### 2.1.3 Leakage Power Estimation and Profiling

The online/offline power profiling mentioned in last two sections are only effective for estimating dynamic power consumption on-chip. However, we have emphasized the importance of leakage power consumption, which due to reductions in the transistor feature size can no longer be neglected in current research. The leakage power consumption is usually regarded as a constant in much of the literature [51][54][55]. However, this is not the case in practice as the leakage power consumption is heavily affected by temperature. In this section, we examine the literature relating the relationship between leakage power and temperature.

The leakage power, also called the static power, consists of two parts: the sub-threshold leakage and the gate leakage [52], as:

$$I_{leakage} = I_{subthreshold} + I_{gate} = A_s \frac{W}{L}\left(\frac{kT}{q}\right)^2 e^{\frac{q(V_{GS}-V_{th})}{nkT}} + I_{gate} \qquad (2.3)$$

where $k$ and $q$ are thermal voltage constants, $n$ is the sub-threshold swing coefficient for the transistor, and $A_s$ is a technology-dependent constant. $L$ and $W$ are the device effective channel length and width. $V_{GS}$ and $V_{th}$ are the gate-to-source voltage and threshold voltage respectively. $T$ is the temperature. $I_{gate}$ is primarily affected by the supply voltage and the dielectric thickness and is relatively insensitive to temperature [52][53]. Hence, $I_{gate}$ is usually considered a constant or even negligibl[6] (particularly in the high temperature case). This leakage model is accurate at the transistor level for both MOS and FINFET circuits [53], however is not suitable at the module or system

[6] If the temperature is greater than 50℃, the gate leakage is a small component of the total leakage, and the sub-threshold leakage is dominant.

level due to the computational overhead, and thus is unable to be applied to high level optimization and management.

Andrei et al. [54] consider the leakage power in their energy-aware scheduling process, but the leakage is assumed to be unaffected by temperature. Bao et al. [55] also use a similar assumption in their static energy-aware scheduling. However, in their later work [50], they improve the leakage calculation for dynamic scheduling scenarios. The following equation is used to evaluate the leakage power:

$$P_{leakage} = I_{sr} \cdot T^2 \cdot e^{\frac{\alpha \cdot V_{dd} + \beta \cdot V_{bs} + \gamma}{T}} \cdot V_{dd} + |V_{bs}| \cdot I_{ju} \tag{2.4}$$

where $I_{sr}$ is the reference leakage current at the reference temperature, $I_{ju}$ is the junction leakage current and $V_{bs}$ is the body bias voltage. $\alpha$, $\beta$ and $\gamma$ are technology dependent coefficients determined by curve fitting. These parameters are usually determined (or measured) prior to run-time, and only the temperature $T$ and the voltage $V_{dd}$ are changed dynamically.

The leakage power and temperature are interrelated, and hence determining the leakage power in a temperature-dependent scenario requires an iterative calculation to reach convergence between the leakage power and the temperature. This calculation usually requires several iterations, and is generally time consuming for an online algorithm. As a result, a reference table is built offline and is directly used for the online estimation. This leakage power model is used by a number of other researchers [52][53] at the micro-architecture level, however, the exponential component of the model increases the computational overhead, restricting its applicability for online purposes.

To reduce the exponential calculation, Liu et al. [52] use an efficient linear estimation for the leakage power. A piece-wise linear (PWL) function is used to map the relationship between the leakage power of a functional unit and its corresponding temperature. Experimental results, compared to those from HSPICE, showed that this technique was both accurate and reduced the computational overhead. An important theorem is developed [52], which states: “for all IC cooling configurations, as long as the total power input is constant, the sum of the IC area-temperature product in the active layer is also constant, if and only if, each power source has the same impact on

the average temperature of the active layer." Therefore, for CMP (where each core is regarded as a separate functional unit, e.g. a core level model for high level analysis) each core's leakage current $I_{leakage}$ can be evaluated as:

$$I_{leakage} = F_{tech} S_{tot} (M T_{avg} + N) \tag{2.5}$$

where $F_{tech}$ is the leakage current per unit area, and depends on the design style, the supply voltage, the manufacturing technology and the input pattern; $S_{tot}$ denotes the area of the core and $T_{avg}$ is the average temperature of the core; $M$ and $N$ are parameters obtained by curve fitting the piece-wise linear model. The accuracy of the estimation improves with an increased number of segments, with the 3-piecewise linear function being very close to the HSPICE results. In this work, this PWL is verified to have the capability that can replace the exponential part in most leakage power modelling and thus, is suitable for online thermal-aware scheduling due to its efficiency and accuracy.

The above research [50][52][53] analyses the leakage power from a pure temperature aspect, which means they are only concerned with the circuit itself and its inherent properties. In other words, if the physical parameters of the IC and the current temperature are known, the leakage power can be estimated. However, other research [56] shows that the leakage power is also affected by the dynamic power, i.e. a higher dynamic power also increases the leakage power, even at a constant temperature. This is because the leakage power changes with the charge/discharge in a circuit.

Sharon et al. [56] gave an empirical equation that shows the ratio $r$ between the dynamic power and the static leakage power induced by the signal activities, as:

$$r = \frac{R_0}{V_0 T_0^2} e^{\frac{B_{tech}}{T_0}} \cdot VT^2 \cdot e^{\frac{-B_{tech}}{T}} \tag{2.6}$$

where $T_0$ is the ambient temperature; $R_0$ is the ratio between the dynamic power and the static leakage at $T_0$ and nominal voltage $V_0$; and $B_{tech}$ is a process technology constant that depends on the ratio between the threshold voltage and the sub-threshold slope, and is computed using the leakage current and saturation drive current values from ITRS 2001.

Unfortunately, the above literature does not consider the computational complexity of the calculations, and particularly the online calculation overhead associated with the power and temperature iterations required for convergence. We have identified this as one of our challenges for achieving DTAS, and we will address this problem in more detail in Chapter 5.

## 2.2 Temperature Estimation and Profiling

As seen earlier in Figure 2.1, there are 3 different techniques to get the temperature value needed for high level optimization purposes. These are: 1) estimating the temperature using a power/thermal model after profiling the power consumption; 2) directly estimating the temperature from the system events recorded in the performance counter; and 3) directly reading the digital thermal sensors (DTS) integrated on chip. These approaches are discussed in this section.

### 2.2.1 Power/Thermal Model

After the power profiling is obtained, by either a simulator-based or counter-based approach, the temperature can be estimated using a power/thermal model. In this section, several related thermal models are introduced, including the simple uniprocessor model, the thermal RC network, the finite element method (FEM) and other empirical models.

Zhang et al. [57] applied the thermal RC model to a uniprocessor. The thermal characteristics and the electrical characteristics have a duality, where the voltage, current, resistance and capacitance in an electrical circuit are equivalent to temperature, power input, thermal resistance (the reciprocal of thermal conductivity) and thermal capacitance. The processor is abstracted as one node connecting the ambient temperature (abstracted as the ground) via one compact thermal resistance and capacitance [7]. This simple model is widely used in other STAS research

[7] The thermal resistance denotes the heat dissipation rate between the core and the ambient temperature, while the thermal capacitance reflects the time interval of heating and cooling.

[58][51][59]. Since there is only one node with power input in entire thermal RC circuit, the core temperature can simply be expressed as:

$$\begin{aligned} T &= T_0 \cdot e^{-RC} + T_s(1 - e^{-RC}) \\ T_s &= T_{amb} + RP \end{aligned} \tag{2.7}$$

where $T_0$ is the initial temperature, $T_S$ is the steady-state temperature which is decided by the thermal resistance $R$ and the power input $P$ at the node, and $C$ is the equivalent thermal capacitance. However, this single node model is unable to give the detailed temperature for each functional unit in the uniprocessor, nor is it suitable for the multiprocessor scenario where each core is abstracted as a power input node and the inter-core heat transferring should be considered.

Dhodapkar et al. [60] presented a cycle-accurate, flexible and scalable tool and framework for power and performance analysis using SimpleScalar [41] and Wattch [43]. Both dynamic power and static power are taken into account. Since static power and temperature are tightly coupled (as stated in a previous section), an iterative thermal computation is implemented in the analysis. The two main highlights that are relevant to our work are:

- Both an empirical mode and an analytical mode are proposed, allowing the user to select between efficiency and accuracy.
- A temperature factor is introduced into the power/performance evaluation to study the thermal management and power-temperature interaction, since sub-threshold leakage and product reliability are exponentially related to temperature. However, this model regards the whole chip as a single uniform thermal resistance/capacitance, evaluating only the average temperature of the whole chip. Such a coarse-grained model may not be suitable to analyse the temporal and spatial thermal distribution on chip for different functional units at the micro-architecture level. Several useful equations (shown in Equation 2.8) are introduced for calculating the thermal distribution:

$$\begin{aligned} Heating: \Delta T^+ &= \left(T_{max} - T_{j-1}\right) \times [1 - exp(-1/\tau_{heat})] \\ Cooling: \Delta T^- &= T_{j-1} \times [1 - exp(-1/\tau_{cool})] \end{aligned} \tag{2.8}$$

  where $\Delta T^+$ is the temperature increment, $T_{max}$ is the maximum junction

temperature, $T_{j-1}$ is the temperature of the previous cycle, $\tau_{heat}$ is the thermal time constant in heating stage, $\Delta T^{-}$ is the temperature decrement, and $\tau_{cool}$ is the thermal time constant in the cooling stage. If instantaneous temperature generated by $T_a + R_t \cdot P_j$ is greater than $T_{j-1}$, then $T_j = T_{j-1} + \Delta T^{+}$, else $T_j = T_{j-1} - \Delta T^{-}$, where $T_a$ is the ambient temperature, $T_j$ is the present cycle temperature, $R_t$ is the equivalent thermal resistance, and $P_j$ is the present cycle power dissipation.

Skadron et al. [35] developed a compact thermal model for architecture-level thermal analysis and optimization. The architecture-level is able to take advantage of the runtime knowledge of the application behaviour and the current temperature information of different on-chip functional units to adjust the execution and distribute the workload in order to optimize thermal behaviour. For the same reason, CMP can also benefit from such a model since each core can be regarded as a single coarse-grained functional unit that can execute a single thread for applications. The allocation of the workload at runtime is essential for core-level and high level thermal management and optimization. At the architecture level, a reliable thermal model is needed to reflect the current and future temperature variation in both temporal and spatial scales for the different functional units on chip.

In terms of the architectural model of a chip, each functional unit can be abstracted into a piece of uniform material, with two adjacent units being connected via a thermal resistance R and a thermal mass (thermal capacitance) C, which are decided by the manufacturing process, the transistor density, the design complexity and the shared lateral area between the two adjacent functional units. Another contribution of [35] is the derivation of the equation to determine the lumped values of resistances and capacitances related to the material thickness and the area. The lumped R and C values can be obtained by experimentation or by using a low level simulator, e.g. HSPICE. Therefore, the temperature transient behaviour of the whole chip and its package can be modelled by a single integrated thermal model. This approach is suitable for architecture and higher level thermal estimation and analysis. Figure 2.2 shows the thermal circuit describing a 4×4 core CMP.

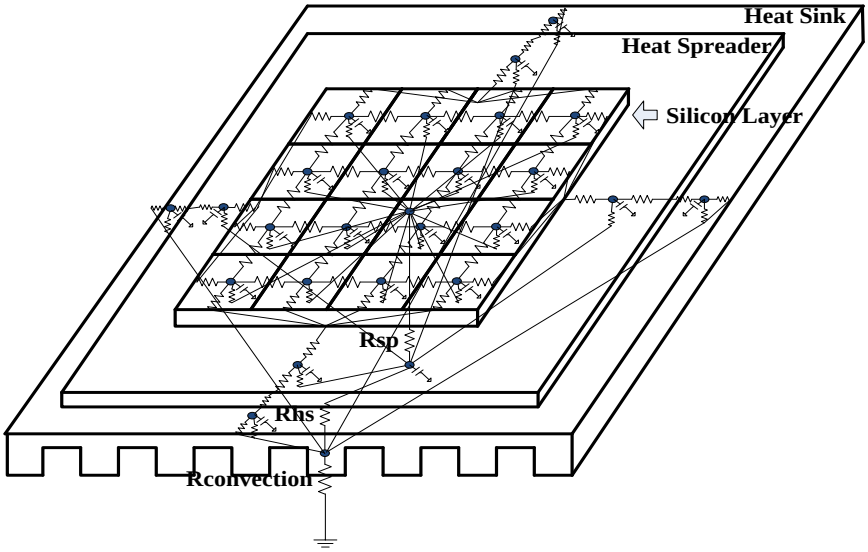



Figure 2.2: Thermal model for 4×4 CMP

Usually, such a model includes several important layers: the silicon layer (the silicon layer is the active layer as this is where the heat is generated, represented as a power source injected into the thermal circuit), the heat spreader layer and the heat sink layer. Power from each functional unit is injected into the circuit at the corresponding node, and then the temperature ($T$) transient of each node can be described and constrained by the differential equations of Equation 2.9:

$$\begin{cases} C_i \dfrac{dT_i}{dt} = p_i + \displaystyle\sum_{j \in Path} \frac{T_i - T_j}{R_j}, & \text{node i} \in \text{silicon layer} \\ C_i \dfrac{dT_i}{dt} = \displaystyle\sum_{j \in Path} \frac{T_i - T_j}{R_j}, & \text{node i} \not\in \text{silicon layer} \end{cases} \tag{2.9}$$

where $C_i$ is the thermal capacitance of node $i$, $T_i$ is node temperature as a function of the time $t$, $p_i$ denotes the instantaneous power input at node $i$, $Path$ is the set of all thermal conduction paths that connect with node $i$, $R_j$ denotes the resistance of each thermal conduction path in $Path$, and $T_j$ is the temperature of the adjacent node in each thermal conduction path. This equation indicates that the number of nodes determines the scale of the differential equation set, since each node represents one equation. Therefore, a large system with many functional units (or cores for core-level simulation) generates a large differential equation set, which requires a significant amount of computation to solve. For example, HotSpot [35] uses a 4$^{th}$-order Runge-Kutta solver to process this equation set.

HotSpot [35][56] models the heat dissipation at a much finer granularity for different functional units at the micro-architecture level. It also considers the lateral resistances and the packaging, as well as dividing the heat dissipation layer into heat sink, heat spreader and internal layer. This detailed modelling can dramatically increase the accuracy of the temperature transient with time. HotSpot and its corresponding thermal models, along with SimpleScalar and Wattch, provide a standard platform for thermal simulation which is widely used in academia. As such, Hotspot has become an important reference and highly used tool for experiments and comparisons related to thermal-aware design and optimization.

However, while HotSpot is accurate, it has a high computational overhead making it unsuitable for use in dynamic (online) thermal optimization scenarios. Two methods [61][62] have been proposed that use the same thermal model as HotSpot, but improve the computational performance. Liu et al. [62] proposed a method that treats the power trace as a piecewise constant power input, and then uses fast spectrum analysis and a moment matching algorithm in the frequency domain to determine the steady state temperature and the transient temperature respectively. This approach gave a 10x-100x speedup, compared to the traditional HotSpot solver. Chen et al. [61] used a global adaptive method to optimize the step size of the iterations in HotSpot, and achieved a 38x--138x speedup. Paci et al. [27] use two different thermal modelling approaches on a 16-core ARM7-based CMP that includes a NoC infrastructure for inter-core communication. One of the models is similar to the previous thermal RC circuit-based model, and the other exploits the advantages of the finite element method (FEM) and corresponding tools (e.g. ANSYS[63] and COSMOL[64]) to analyse the temperature variation across the chip. FEM can provide a very accurate thermal distribution and a detailed temperature transient analysis, but at the cost of a very long computation time. Due to its accuracy, it is ideal as a reference for other thermal models or thermal estimation methods. The significant contribution is the comparison between FEM and the thermal RC network, which shows that the temperature errors of the thermal RC network are small (in the range 0.1 to 0.6°C).

Stavrou and Trancoso [65] developed a fully parameterizable tool for CMP thermal scheduling simulations, called TSIC, which allows the testing of different

configurations, application characteristics and scheduling policies. As TSIC is designed for CMP systems, the thermal model must reflect the spatial and temporal thermal diversities over the chip, and take into account the inter-core and ambient heat transfer, as well as the lateral heat dissipation. Notably, their approach does not use the thermal RC network of [35] to obtain an accurate thermal distribution in each time interval, but rather an intuitive empirical model, described by Equation 2.10, is adopted for updating the temperature of each core.

$$\Delta T_i = \left[ \sum_{m=1, m \neq i}^{n} f_c(T_i - T_m) \right] - [f_a(T_i - T_{ambient})] - [f_p(Process_i, T_i)] \quad (2.10)$$

The first term in Equation 2.10, models the inter-core heat exchange, implying that such an exchange exists among any pair of cores on chip. For any pair of cores, $A$ and $B$, $f_c(T_A - T_B) = -f_c(T_B - T_A)$ must be satisfied. The function $f_c$ is dependent on the difference in temperature of a pair of cores and the location of these two cores. The second term in Equation 2.13 indicates the heat dissipation from a core to the external environment ($T_{ambient}$). The function $f_a$ also needs to consider the temperature difference and the core's location. For a core that is not on the chip edge, only the vertical thermal path is considered. The lateral thermal path is only considered for the outer cores. The last term of Equation 2.10 calculates the temperature change induced by executing the application (or process).

The model proposed in [65] improves the computation efficiency for temperature evaluation so as to be applicable for the online scenario. However, the model uses a time-driven approach which is only suitable for small-interval updates, using very fine granularity scheduling. This is because the temperature error becomes significant above 1 millisecond. Contrast this to the analytical model (thermal RC model), where any update interval is applicable.

Zhan et al. [66] proposed a novel approach to rapidly calculate the temperature distribution in VLSI chips by using the discrete cosine transform (DCT) and LUT. This approach improves on the previous temperature distribution algorithm that uses the Green's function [67] and the unrealistic assumption that the chip is infinitely large horizontally. Experimental results show that this approach is accurate, with a relative temperature error of less than 1%. The approach is suitable for a fine-grained

thermal analysis at the micro-architecture level since the computational overhead does not increase with the number of power sources (nodes on the active layer in the thermal RC model) and there is no need to solve a large set of differential equations. Unfortunately, the approach proposed in this paper only solves the steady state temperature distribution, and does not consider the transient profile as only the thermal resistance is taken into account while the thermal capacitance is ignored.

Michaud and Sazeides [68] used analytical methods to develop a temperature model, called ATMI. ATMI is based on an idealized microprocessor chip and its packaging. Partial differential equations are used to describe the physical model of ATMI (formed by two layers: the chip layer and the packaging layer) and its boundary conditions (the temperature and its first order differential on the cross-section of the adjacent layers and the lateral-section adjacent to the ambient). As ATMI is a linear model, and the multi-source power input can be solved by superposition of the individual power sources, the model produced accurate results, verified by the thermal sensors on chip. However, the convolution operation, used to obtain the temperature transient over time, is very time consuming, and thus is not computationally efficient enough for online thermal estimation and prediction in the OS, even when using the fast Fourier transform (FFT) to accelerate the convolution operation.

Computational efficiency and the accuracy of the power/thermal model are both critical in the thermal simulation, estimation and optimization topics mentioned above. The thermal models used in current research have several shortcomings: 1) Some thermal models (e.g. TEMPEST) are only used for uniprocessors, and cannot be applied to a multi-core scenario; 2) Some thermal models (e.g. TSIC) use inaccurate heuristic or empirical thermal modelling that is hard to adapt for different layouts and architectures; 3) Some thermal models [66][93][94] only consider the steady-state temperature and ignore the transient temperature because of the limitations in the solving algorithm (e.g. linear programming and DCT LUT), and thus the temperature error is large; 4) Some thermal models (e.g. HotSpot or the FEM approach) provide an accurate temperature estimation when used for offline thermal simulation, but require a computational overhead which makes them unsuitable for online thermal management; 5) Current thermal models (e.g. TSIC) are time triggered, and update the temperature in a fixed fine-grained time step, which adds to the computational

overhead; 6) Much of the research related to thermal-aware scheduling [90][92][57] assumes that the leakage power is invariant (that is a non-temperature-dependent leakage model is assumed). While these assumptions simplify the problem by skipping the power-temperature iteration, they can lead to unrealistic thermal results. Therefore, current thermal estimation and modelling does not sit well amongst the various competing factors: e.g. computational efficiency, estimation accuracy, updating period and leakage power.

### 2.2.2 Direct Readings from Digital Thermal Sensors (DTS)

The simplest and most intuitive way to get the chip/core temperature is by reading the integrated on-chip DTS directly. A number of practical DTM and DTAS techniques [29][70][71][72] take advantage of DTS. However, DTS has some inherent disadvantages which can affect the high level thermal management and optimization. For example, if the DTS reading is lower than the actual temperature, DTM would be triggered and activated later than desired, which may result in the degraded reliability of the processor since the temperature may exceed the predefined threshold; if the DTS reading is higher than actual one, the early activation of DTM can significantly reduce the performance and waste computation resources. Even though DTS is widely used and accepted, there are some issues, particularly relating to its accuracy. In this section, we will focus on those issues.

Zhang [73], Rotem [74] and Sharifi et al. [75] described that the accuracy of DTS readings is a primary limitation. Several reasons can affect the accuracy of DTS reading.

- Noisy behaviour: In reality, on-chip DTSs are affected by a range of noise sources. Some of these noise sources include fabrication randomness, power grid noise, cross coupling, non-linear dependence between temperatures and the circuit parameters and even A/D converter accuracy. Assuming ideal thermal sensor operation can either lead to failure to detect overheating or false alarms that result in costly and unnecessary responses from the thermal management unit [73].

- Placement error: Typically, sensors cannot be placed exactly at the locations where monitoring is critical, as these locations are high power density areas (e.g. they are accessed frequently or are high performance components) where silicon is at a premium. This means that the sensor cannot optimally capture the chip hot spot [75].
- Calibration error: Many chip manufacturers use un-calibrated DTS due to the high cost and overhead associated with the thermal sensor calibration [75]. This is particularly the case for systems featuring multiple sensors and can result in significant thermal errors [75].

Sharifi et al. [75] also indicated that the current trend is that the number of DTSs per core is decreasing as the number of cores increases. For example, Intel's Core 2 Duo and AMD's Quad-Core Opteron processor have multiple DTSs on each core, while Intel's 48-core SCC only has one DTS for each core. However, in many-core systems, these sensor based approaches are likely to be even less practical. The future many-core systems may even group the cores into clusters which share the DTS among cores [76]. The reason is that more DTS needs more channels for routing, thus increasing the silicon area and test cost.

In addition, the relative long thermal response latency is another inherent disadvantage that limits the usefulness of DTS in high level thermal optimization. Most state-of-the-art on-chip DTS need 30--150 milliseconds to reflect the temperature change at the measuring point [74]. This may be suitable for passively triggered DTM, but is not suitable for DTAS, since TAS needs to work with the scheduler in a fairly short time interval (less than that of the OS time tick), while DTM (e.g. DVFS or putting the core into sleep mode) requires a larger overhead (it requires a lot of cycles to complete these operations at the hardware level) and as such should not be triggered too frequently.

The information from DTS is also limited in that it only reflects the temperature at a single point and at the current time instant. This single point on one core cannot be regarded as the average temperature or the worst-case temperature, and as such it is not possible to build a full thermal map or tell where the hot spots actually are. It is also more difficult to use DTS readings to predict the realistic future temperature of the core without the assistance of other complicated mathematical models or training

methods. DTS is a simple and direct way to get temperature information, but is not fast enough and may not have the required accuracy to guide high level thermal management and optimization [73][74][75]. Therefore, some researchers [73][74][75] have considered both direct and indirect measurements for determining chip temperature and use modelling methods to complement the DTS readings.

## 2.3 Dynamic Thermal Management at the Micro-Architecture Level

After obtaining the temperature information on-chip, high level thermal management and optimization can be carried out. We have already emphasized that DTM is mainly a passive technique triggered by some predefined temperature threshold. DTM techniques usually have a higher system overhead, thus are not suitable for high speed or frequent activation. As such, in this thesis, we refer to DTM as a coarse-grained thermal optimization technique. In this section, we will introduce the different DTM technologies appearing in the recent literature.

Benini et al. [77] investigated system-level dynamic power management (DPM). DPM involves selectively turning off (or reducing the performance of) system components when they are not used (or only partially used). Generally, the principle of DPM can be described by a state machine: RUN, IDLE and SLEEP consume different amounts of power and power saving can be achieved by the processor moving between states depending on the processing requirements. Different control mechanisms and implementations of DPM are analysed, and a number of examples of DPM techniques are presented. The Advanced Configuration and Power Interface (ACPI) standard, which is widely used in desktop PC and other embedded microprocessors, is described. The ACPI can monitor a set of thermal sensors and the CPU load indicator. According to these readings and a predefined power/thermal budget, the operating system invoking the ACPI interface can dynamically adjust the performance of the CPU, memory and other devices, making them switch between different execution states (e.g. working at high speed, sleep, standby or hibernation) to reduce the power consumption and heat dissipation.

Brooks et al. [20] examined dynamic DTM mechanisms in modern microprocessors.

These techniques refer to a range of possible hardware and software strategies which work at runtime to control the chip's operating temperature. DTM can reliably reduce the power and temperature by using inexpensive hardware or software solutions, while impacting as little as possible on the processor performance. This reduces the need for larger external physical devices, such as larger heat sinks, cooling fans, etc., and thus reduces cost. Five DTM mechanisms are discussed in [20]. These include: clock gating, DVFS, decode throttling, speculation control and I-cache toggling.

- Clock Gating and Stop & Go (Sleep): It is possible to stop driving idle modules, by adding an AND gate before the clock input of the module. If this technique is applied to a core instead of an individual functional unit, this technique is known as "Stop & Go" policy. This means that the processor would start to sleep when its temperature reaches the upper-limit threshold and then would resume running when the temperature drops to below the lower-limit threshold. Clock gating is different from the shutdown policy: in the case of clock gating, the states of the core are latched into registers similar to a freeze operation and only the dynamic power consumption is eliminated; however, in the case of a shutdown, the complete circuitry of the core is switched off after its state is saved to external memory.
- Voltage and Frequency Scaling: The supply voltage and clock frequency are adjusted dynamically. When the power consumption, and thus the temperature, is higher than desirable the supply voltage or the clock frequency is decreased without stopping the processor. The drawback is a noticeable loss in compute performance. However, if the application requirements are still satisfied then this loss in performance is not really an issue. Almost all current processors used in commercial desktop PCs take advantage of DVFS. However, the core voltage can usually only be assigned to a set of discrete values (that cannot be changed in a continuous range). This is usually done globally, and is referred to as global DVFS (distinguished from distributed DVFS that controls the voltage on a per-core basis).
- Decode Throttling: This technique restricts the flow of instructions from the I-Cache to the core when DTM is triggered. The fewer instructions that are sent to the decoding and execution stage, the less energy consumed.

- Speculation Control and Pipeline Gating: Speculation control is the technique of arbitrarily restricting the amount of speculation in the pipeline whenever a thermal trigger level is reached. This is implemented using a counter which is incremented whenever a branch instruction is decoded and is decremented when a branch is resolved. If the counter exceeds the limit, the decode stage stalls until enough branches have been resolved. Similarly, pipeline gating is based on speculation control, where several stages in the pipeline are shut-down when a stall occurs or some stage is idle.
- Unit Toggling: To alleviate temperature related problems, some units, like the ALU and register file, can be replicated into multiple copies. When one unit is overheated, the other can be started so as to allow the overheated unit to cool without interrupting execution. I-Cache toggling is a special case of unit toggling that stops the instruction fetch unit at the specified interval and uses the instruction queue to continuously feed the instructions.

The above DTM mechanisms can be classified by the range over which they are applied in the multiprocessor: global DTM means one global controller controls all the cores on chip in a uniform manner, while distributed DTM has multiple controllers for each core or a cluster of cores, which are controlled independently.

Donald and Martonosi [21] explored various thermal management techniques that exploit the distributed nature of multi-core. They used Turandot [78] and a HotSpot-based thermal simulator to simulate a variety of workloads under thermal stress on a 4-core PowerPC processor. A new high-level DTM designed for multi-core, known as “migration” is described. Migration can move the workload among the different processors in the system, like unit toggling (described above), except that a unit now refers to a single processor. It records the processor state and all the context of the running program into temporary memory and then restores this context to another processor and resumes execution of the stalled program. When the temperature of one core exceeds the threshold, migration can move the task on this core to another cooler idle core and put the overheated core into a sleep state. In fact, migration only introduces a relatively small overhead (only a context switch at the OS level) compared with other DTM methods. Therefore, migration can also be applied in fine-grained management and optimization, such as DTAS. However, the migration

described in [21] is confined to passive triggered mode, which is still a coarse-grained DTM technique which is only triggered when the temperature threshold is exceeded.

Chaparro et al. [79] demonstrated the thermal implications of multi-core and many-core architectures. They argued that dynamic thermal management for a multi-core processor is a relatively new area and will gain more and more visibility in the EDA arena. Two important categories for DTM were generalised: 1) Temperature can be decreased by reducing the power consumption at the cost of some computational speed degradation, so long as this degradation still allows the requirements of the application, such as real-time deadlines and computational throughput, to be met. DVFS is one such example as it achieves a quadratic energy reduction with only a linear speed reduction. 2) The temperature is controlled by distributing the processor activity over the chip area, which is similar to workload balancing in a distributed system. Many techniques belong to this category: toggling at the granularity of the functional unit, pipeline, cache, instruction decoder, etc., and thread migration (TM).

A significant contribution of [79] is the in-depth analysis of the parameters (e.g. number of cores, thermal parameters, the maximum allowed temperature, the overhead of different policies, temperature measurement interval, scheduling interval, etc.), especially relating the sensitivity of the different thermal control policies to the value of these parameters. A number of important observations are made:

- As the number of cores increases, the temperature of a core depends on the global heat dissipation rather than the local heat dissipation. This is because the heat spreader and the heat sink gather the heat generated by all the cores.
- Solving a thermal RC circuit with many nodes is very time consuming. The overhead can be decreased by reducing the internal nodes of a core. However, modelling different nodes inside a core is important as hot spots usually affect small regions in the core and not the whole core homogeneously.
- The lateral heat exchange between cores and from the core’s free edge to ambient should be modelled for accuracy. For the same reason, the multi-node heat spreader and heat sink should be modelled as well. Leakage power modelling is still implemented as a feedback loop and requires a large number of iterations for temperature-power convergence.

- In terms of global DVFS and distributed DVFS, it is better to perform distributed DVFS in most cases. However, distributed DVFS is significantly more difficult to implement. Global DVFS is still useful when used with passive thermal dissipation as most cores will trigger DTM shortly after one of the cores overheats. Therefore, global DVFS applied to the whole chip helps to achieve a faster cool-down time.
- TM (Thread Migration) only helps distributed DVFS in conjunction with active thermal dissipation. For systems with passive thermal dissipation, TM actually inhibits the performance of distributed DVFS.
- The performance of TM + global DVFS is very close to distributed DVFS and provides a good trade-off between the number of migrations and the average core frequency/voltage level.
- Different system parameters can lead to different optimal management schemes and/or scheme configurations.

Kumar et al. [69] proposed a hybrid DTM scheme: a hardware–software DTM technique using both proactive mechanisms, such as migration, to avert thermal stress, and reactive mechanisms, such as clock gating, to deal with overheating. Most importantly, they implemented this hybrid DTM, adopting a Pentium 4 as the experimental platform and modifying the Linux kernel to make the thermal-aware scheduler fit into the OS. Both uniprocessor and simultaneous multi-threading (SMT) [80] are analysed in this work. Another important contribution is the novel regression thermal model that provides a relatively fast and accurate prediction of the overall processor temperature, directly from the hardware performance counters. Unfortunately, this work only considers the uniprocessor and SMT cases, and is not applied to CMP.

Mulas et al. [72] proposed a passive migration method for stream computing on MPSoC. Generally, DTM is only triggered when the temperature exceeds some ceiling threshold, but in this paper, two thresholds are set: a higher and a lower threshold. Each time the temperature of a processor reaches the upper/lower threshold, task migration is triggered moving a task from one processor to a lower temperature processor. To reduce the amount of computation in selecting tasks to migrate, the migration process is restricted to two processors at a time, e.g. from one hot core to a

cold core. A two phase algorithm is also proposed to implement this policy is used to reduce the amount of computation in selecting tasks to migrate. The first phase chooses the candidate cores (pair of source and destination) that might need thermal balancing by evaluating their current temperatures, clock frequencies and power consumptions. After determining the pair of cores, the second phase is to determine the number of migrating tasks on the source core and the destination core by evaluating and minimizing the migration cost. This approach can effectively limit the number of migrating tasks so as to reduce the overhead introduced by the context switching. Another contribution in [72] is the middleware implementation of a thermal balancing policy in uC/Linux. A comparison is made between the energy balancing policies, the stop & go policy and their proposed policy using an FM radio benchmark. The results show their proposed policy has advantages in terms of deadline miss, migration cost and temperature deviation.

## 2.4 Thermal-Aware Scheduling

Thermal-aware scheduling is an active technique for optimizing the system level temperature. Unlike DTM, it is not triggered by a temperature threshold, but instead uses proactive thermal reduction measures even though the temperature has not yet exceeded the temperature threshold. For instance, the OS scheduler could allocate tasks to cooler cores when it is invoked. This can occur over a relatively fine-grained interval (e.g. at each timer tick, at a return to user space from an interrupt handler or from kernel space, when creating a new task, and so on). Sometimes, the workload/task schedule can be pre-determined at the design stage to achieve the desired thermal management, by using offline thermal simulation.

Thermal-Aware Scheduling can be classified into two categories, according to when they are applied, as: 1) Static TAS (STAS) and 2) Dynamic TAS (DTAS). With STAS, a thermal simulation is carried out offline (during the design stage) and as such does not require temperature information on-the-fly. This assumes that a task's properties (e.g. execution time, power consumption etc.) are known and can be used as input to a thermal simulator. The design stage task then reduces to searching for a suitable schedule which minimizes the temperature or reduces some other thermal effect. As

this thermal simulation is done during the design stage, the computational overhead is generally not a constraint. On the other hand, DTAS requires real-time temperature information from thermal sensors, thermal models or both. The overhead of thermal simulation or optimization is this case must be kept as low as possible so that it can be integrated with the task scheduling process.

### 2.4.1 Power/Energy-aware Scheduling

Power-aware and energy-aware policies attempt to minimize the power or energy of a system. They are somewhat related to TAS, and as such it is appropriate to examine and analyse them here, even though they are not a focus of this work.

Irani and Pruhs [81] surveyed a number of recent works on algorithmic problems related to power management. First, the formalization of speed scaling (i.e. dynamic frequency scaling) is reviewed as a scheduling problem that is combined with real-time task features. There are two goals for this problem: 1) minimizing the total energy used subject to the deadline feasibility constraints; and 2) minimizing the peak temperature subject to the deadline feasibility constraints. This work also details some open problems for power management, many of which are related to temperature-aware issues. Therefore, it is natural to examine these two concepts (e.g. energy-aware and thermal-aware) together.

Bansal and Pruhs [82] assumed that the rate of cooling of the device adheres to Fourier's Law, which states that the rate of cooling is proportional to the difference in temperature between the object and the environment. An approximation of the rate of change $T'$ of the temperature $T$ can be expressed as: $T' = ap - bT$, where $p$ is the supplied power, and $a$ and $b$ are constants. They also observed that there is a relationship between total energy and the maximum temperature, and thus were able to simplify the temperature calculation. They found that over an interval, if $b = 0$, then the temperature minimization problem is equivalent to the energy minimization problem; whereas, if $b = \infty$, then the temperature minimization problem is equivalent to the peak power minimization problem, or equivalently the peak performance minimization problem. However, similar to the power/thermal relationship described

in Section 2.2.1, the relationship between maximum temperature and total energy is relatively complex, and the simplified empirical relationship used in [82] is unrealistic, limiting the technique's practicality.

Energy minimization with deadline feasibility is introduced in [83][84][85][86]. An offline greedy algorithm, called YDS [83], is used to optimally solve this problem iteratively. During each iteration, tasks with the highest periodic frequency are scheduled using earliest deadline first (EDF) [83] at a speed equal to that frequency. A native implementation of YDS has $\Theta(n^3)$ time complexity. This can be improved to $\Theta(n^2)$ if the interval has a tree structure. It has been shown that calculating the minimum energy schedule for jobs with a fixed priority is NP-hard [84]. Irani and Pruhs [81] also give a fully polynomial time approximation scheme for the same problem, while [85] gives a polynomial time algorithm for the case of a processor with discrete speeds.

Most energy-aware scheduling regards the energy/power budget as the optimization goal [87][88][89]. However, energy-aware scheduling is different to TAS. While it is easy to put into practice, it is not as useful as the relationship between energy and temperature, particularly for CMP. We do not consider energy-aware scheduling further as the energy/power budget is not an explicit constraint limiting the processor operation, however temperature is. Energy-aware scheduling cannot achieve the required thermal characteristics in many cases [87], because minimizing energy does not consider the energy distribution on chip and energy has no explicit relationship with temperature. High energy/power is not equivalent to high temperature, as temperature is related with power density and thermal dissipation features. For example, a high energy/power task might be allocated to a core with large silicon area and a location with better heat dissipation (e.g. on the edge of the chip) or to a core with a low temperature, and as such, even though it may not meet the required energy budget it may still run below the temperature threshold.

### 2.4.2 Static TAS

Zhang and Chatha [57] addressed the problem of performance optimization (mainly

minimizing the execution latency of a task set) for a set of periodic tasks with discrete voltage/frequency states under thermal constraints. They examined performance maximization of offline thermal aware scheduling on a uniprocessor under a predefined thermal threshold. Both optimal and approximate scheduling algorithms for performance optimization were presented. The optimal solution, using restricted shortest path, is NP-hard, while for the approximate solution, a fully polynomial-time approximation scheme (FPTAS) is proposed to improve the efficiency of the scheduling algorithm within a designer specified approximation bound. The approximate solution is 24 times faster than the optimal algorithm. The thermal RC circuit is used to capture the thermal behaviour, however, the coarse-grained model, which treats the uniprocessor as a single node, does not accurately reflect the temperature variation and details at the micro-architecture level. Additionally, the uniprocessor model uses a very simple thermal model which is not applicable to the multi-core scenario.

Xie and Hung [90] proposed a heuristic static thermal-aware task allocation and scheduling method based on hardware/software co-synthesis. HotSpot is used to evaluate the maximum and average temperature of the whole chip in the next scheduling interval. As this is a static scheduling problem, the off-line overhead associated with thermal estimation using Hotspot is not important. This work uses a task graph to define the dependency of tasks in the task set, as the task properties (such as the average power consumptions and worst case execution times (WCET)) are known in advance in STAS. The temperature is then used as a metric to move the tasks among different processing elements to achieve load balancing and in calculating the migration schedule offline. However, this heuristic metric does not consider dynamic temperature (the temperature evaluation and schedule are pre-determined offline, and thus lack runtime flexibility), nor does it provide an optimal solution for the static scheduling problem.

Chrobak et al. [91] examined temperature-aware scheduling problems and formalized a number of algorithms used in both offline and online scenarios. This is one of the first papers to undertake a theoretical analysis of thermal aware scheduling on a microprocessor. The objective of offline scheduling is to get a schedule which maximizes the number of tasks that meet their deadline. There are several important

conclusions: 1) a real-time task set with properties (e.g. the release time, deadline and heat contribution) is suitable for studying scheduling problems in real-time embedded systems; 2) computing the optimal offline schedule is NP-hard, even when all tasks are released at the same time and have equal deadlines.

This paper was one of the first to formalize thermal-aware scheduling. The premise is that hardware thermal management can continuously monitor and control the temperature using a feed-back loop policy which simply stopped the processor when it overheated and resumed execution after the next fixed-length idle slot without any consideration of the overhead. This form of thermal management is even simpler than the stop-go policy [20], and is impractical due to an oversimplified model and some unrealistic assumptions. The main reasons are:

- Although a task set with real-time features is used for analysis and discussion, all tasks are oversimplified with the same execution time.
- A simple analytical thermal model is used which does not adequately describe the exponential temperature transient during the heating and cooling stages.

Murali et al. [93] take advantage of convex optimization to solve the optimal scheduling for minimizing the peak temperature subject to deadline feasibility. The algorithm is completed in two phases: an offline and an online phase. In the off-line phase, an optimal frequency assignment for the different processors in order to meet a particular workload constraint, while satisfying the thermal constraints, is determined. Then, the convex algorithm is used to solve the optimal frequency for different workload requirements and initial core temperature values. Lastly, the frequency assignments for different cores under different workloads and initial temperature combinations are stored in a table for online look-up. In the online phase, the thermal management unit tracks both the application workload and temperature on a core, and then finds the optimal frequency from the table and applies DFS periodically (they refer to this DFS as Pro-Temp). The main contributions are:

- Convex optimization is shown to be applicable for this frequency assignment in STAS as the frequency constraints are quadratic, rather than linear (if all the constraints and objectives are linear, then linear programming can solve the optimization problem).

- Pro-Temp is shown to be superior to basic DFS in terms of the task waiting time and the duration that the temperature exceeds the threshold.

However, the assumptions and the thermal model used are limiting factors:

- A core must have one thermal sensor to track the temperature for online use. This may limit applicability for future many-core systems.
- The thermal model only considers the thermal resistance (thermal capacitance is ignored). Thus, only a steady state temperature analysis can be used. This is not practical as it assumes that the temperature between any two consecutive DFS time points reaches steady state without the temperature transient.
- The power consumption per core is only decided by the core frequency irrespective of the task characteristics. This oversimplification is not realistic as the power consumption for a core also varies on a cycle by cycle (and task by task) basis.

Chantem et al. [94] examined static thermal aware scheduling to minimize the peak temperature. Both an optimal solution and a heuristic solution were proposed. Similar to the work of [95] and [57], their optimal solution is obtained using mixed integer linear programming (MILP) for a non-preemptive task set based on a task graph. Some important observations from their work are:

- The core power consumption only changes at the beginning or end of task execution.
- The temperature of a core experiences a rapid change with a change in power.
- The leakage power is significant and cannot be ignored in the calculation due to its non-linear relationship with temperature. Leakage power can be approximated by a linear function, in the operating temperature ranges of integrated circuits, with roughly 5% error.
- The thermal model can be further refined by using multiple thermal elements for each core, where the finer granularity of each core could reflect the thermal effects at the micro-architecture level.

They explicitly state two limitations of their optimal solver: 1) the MILP formulation cannot be used to efficiently solve large problem instances and 2) the steady-state

analysis used in the MILP-based implementation (a linear optimization problem cannot deal with the non-linear transient temperature when thermal capacitance is considered) may overestimate the chip peak temperature when the task execution times are short compared to the RC thermal time constants of the cores.

Their heuristic solution uses a scheduling framework where either steady-state or transient thermal analysis can be used. This framework takes advantage of a binary-search based method to iteratively improve the solution. However, the iteration process uses HotSpot to carry out the transient temperature calculation, and thus the time complexity of the framework depends on the HotSpot overhead.

Coskun et al. [95] explored the benefits of thermal aware scheduling for MPSoC using two different categories: static and dynamic scheduling. Static scheduling is modelled using integer linear programming. Different goals (e.g. maximizing performance, minimizing thermal hot spots (temperature) and gradients, minimizing and balancing thermal hot spots, balancing energy consumption and minimizing total energy) are studied for solving the optimal solution. However, a temperature threshold is not considered as a constraint in their STAS implementation.

In terms of STAS, several problems have been identified in this section. These problems can be summarized as: 1) Much of the STAS research is only applicable to a uniprocessor scenario, and does not consider the complexity of the spatial thermal distribution introduced by multiprocessor systems; 2) Linear programming (convex optimization) [93][94][95] can be used for STAS, but its linear (quadratic) constraints cannot deal with a transient temperature model ; 3) Some of the proposed algorithms [95][93], which optimize performance, energy and temperature, allow the core temperature to go above the thermal threshold (that is, they do not adhere to a hard temperature threshold constraint and instead allow the processor overheat) and then only minimize the overheating duration); 4) To the best of our knowledge, there is no STAS schedulability test for multiprocessor in the relevant literature due to the high computational overhead of current thermal models (a schedulability test without any thermal consideration is an NP problem).

### 2.4.3 Dynamic TAS

Dynamic TAS (DTAS) is a relatively new research area. Current DTAS approaches in the literature fall into three categories, according to the way the temperature information is used: 1) Look-current scheduling is based on the temperature at current time instance; 2) Look-backward (historical) scheduling uses a historical temperature profile to guide scheduling; and 3) Look-forward (predictive) scheduling uses the predicted future temperature or predicted temperature trend to guide scheduling.

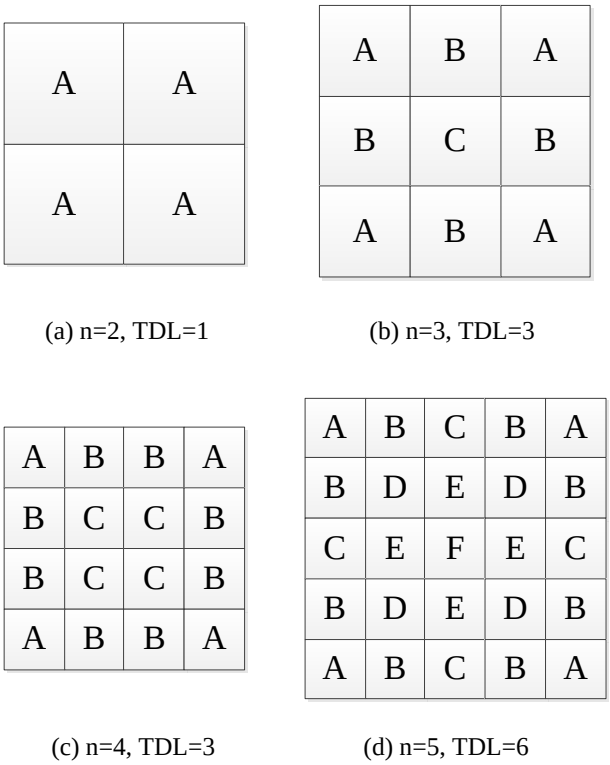
#### 2.4.3.1 Look-Current

Stavrou and Trancoso [92] examined TAS on CMP. With CMP, the increasing number of cores and the reducing feature size can lead to an extremely high power density and high thermal dissipation on chip. Additionally, the thermal distribution on chip can have dramatic differences between locations, resulting in a significant temperature gradient in both space and time. The contributions are the identification and clarification of the thermal issues that arise from these CMP (many-core) chips, as well as the proposal and evaluation of several heuristic dynamic thermal-aware scheduling policies. These include:

- The observation that an accurate and efficient thermal model is necessary for estimating and evaluating the temperature of each core on chip. It must consider the heat exchange between adjacent cores, the lateral heat dissipation on the cross-sectional area at the edge of the chip and the vertical heat emission from chip to ambient.
- The reliability problem of CMP is posed here to emphasize the fact that CMP has a greater failure rate compared to a single core due to thermal issues.

Stavrou and Trancoso also state that thermal-aware floorplanning is likely to be less efficient when applied to CMP, as core-level decisions are unlikely to be optimal when several cores are packed on the same chip due to the interaction among cores. The paper also introduces the concept of thermally different locations (TDL). The important implication of TDL is that if a task is allocated to a core, it will produce the same thermal effects and distribution as it would when allocated to another core of the

CMP with the same TDL. Figure 2.3 shows that for a CMP with $n^2$ cores, there will be $\lceil n/2 \rceil \cdot (\lceil n/2 \rceil + 1)/2$ different possible TDLs (each TDL has same letters on cores).




(a) n=2, TDL=1 (b) n=3, TDL=3

(c) n=4, TDL=3 (d) n=5, TDL=6

Figure 2.3: A CMP and its corresponding TDL

A thermal-aware scheduler for CMP, implemented in a real OS, can enable system-level (high level) thermal optimization without the need for micro-architecture changes. Several scheduling policies are mentioned in [92], including:

- Coolest: The ready task (which could be a process or a thread in the ready-to-run list) is assigned to the coolest idle core (shown as "C" in Figure 2.4). This is the simplest thermal-aware algorithm and the easiest to implement.
- Neighbour-Aware: For each available core, this algorithm calculates a cost function (given in Equation 2.11) and selects the core that has the minimum cost (as shown as "N" in Figure 2.4). This cost function takes into consideration the following: 1) The temperature of the candidate core ($T_c$); 2) The average temperature of direct neighbour cores ($\bar{T}_{DA}$); 3) The average temperature of diagonal neighbour cores ($\bar{T}_{dA}$); 4) The number of non-busy direct neighbour cores ($NB_{DA}$); 5) The number of "free" edges of the candidate

core ($N_{fe}$). The weight $a_i$ (in Equation 2.11) implies the importance of these different aspects and its value is determined statistically by experimentation to match the characteristics of the layout of a specific CMP. The rationale is that, the lower the temperature of the core's neighbourhood, the easier it will be to maintain its temperature at a low level due to inter-core heat exchange. Cores at the edge of the chip benefit due to the increased heat loss.

$$Cost = a_1 \cdot T_c + a_2 \cdot \bar{T}_{DA} + a_3 \cdot \bar{T}_{dA} + a_4 \cdot NB_{DA} + a_5 \cdot N_{fe} \quad (2.11)$$

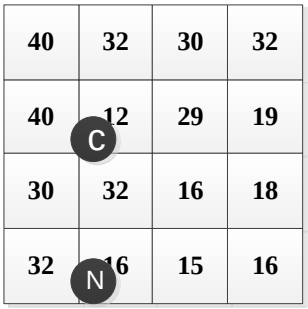



Figure 2.4: Example of TAS polices [48]

- Threshold neighbourhood: Uses the same cost function as neighbourhood, but schedules a task only if the cost function is lower than the pre-defined threshold. This means that a task should only be executed on a "thermally good" core, thus avoiding greedily choosing an idle, but thermally adverse, core. This approach would appear to affect the performance to a certain degree. However, in some cases the policy can improve the performance by reducing the frequency of DTM.
- Maximum Scheduling Temperature (MST): MST is not an algorithm by itself, but can be used with the previously mentioned algorithms. MST prohibits scheduling a task for running on idle cores when their temperature exceeds a predefined threshold. The temperature set by MST should be lower than the temperature which triggers DTM. The gap between these two critical temperature points needs to be carefully tuned by system designers. The rationale behind this is to avoid the triggering jitters of DTM, thus allowing the idle core to cool down for a sufficient interval.

The paper also gives some meaningful metrics for evaluating these heuristic policies

to verify if these algorithms produce better thermal effects and performance trade-offs. These metrics include:

- Average Temperature: This metric represents the average temperature of the cores on chip during the simulation period. The average temperature is given by Equation 2.12, where $T_i^t$ is the temperature of core i during the simulation interval $t$, $S_T$ is the total number of simulation intervals, and n is the number of cores on chip.

$$Average\ Temperature = \bar{T} = \sum_{t=0}^{S_T} [\frac{\sum_{i=0}^{n}(T_i^t)}{n \cdot S_T}] \tag{2.12}$$

- Average Spatial Diversity: The spatial diversity indicates the variation in the temperature among the cores at a given time. The average spatial diversity is the average of the spatial diversity during the whole simulation period and is given by Equation 2.13. The bigger the value, the worse the thermal effects are as a large temperature gradient could accelerate the aging of the chip and lead to an unreliable state. In the average spatial diversity equation, $\bar{T}^t = 1/n \cdot \sum_{i=0}^{n} T_i^t$, is the average chip temperature during the interval t. This metric has some shortcomings: it cannot reflect the local thermal difference and cannot reveal the distribution of hotter and cooler cores. For example, the scattered distribution of hotter cores is much better than putting them together, but if the temperature of each core is unchanged, this value remains the same.

$$Average\ Spatial\ Diversity = \sum_{t=0}^{S_T} [\frac{\sum_{i=0}^{n}|T_i^t - \bar{T}^t|}{n \cdot S_T}] \tag{2.13}$$

- Average Temporal Diversity: The average temporal diversity measures the variation of the average chip temperature, across all cores, over the whole simulation progress. It is defined by Equation 2.14.

$$Average\ Temporal\ Diversity = \sum_{i=0}^{S_T} [\frac{\sum_{j=0}^{S_T}|\bar{T}^t - \bar{T}|}{n \cdot S_T}] \tag{2.14}$$

- Efficiency: Efficiency is a metric relating to the performance that the CMP achieves under thermal constraints compared to its full potential. Efficiency is defined (Equation 2.15) as the ratio between the time required for the execution of the workload (Workload Execution Time) under thermal constraints and the execution time required if no thermal constraint existed (Potential Execution Time). The maximum value for the Efficiency metric is 1.

$$Efficiency = \frac{Potential\ Execution\ Time}{Workload\ Execution\ Time}$$
$$Potential\ Execution\ Time = \sum_{n=1}^{\#processes} \frac{Lifetime(Process_n)}{Number\ of\ Cores} \quad (2.15)$$

### 2.4.3.2 Look-Backwards

Coskun et al. [29] explored the benefits of thermal aware scheduling for MPSoC using two different categories: static and dynamic scheduling. The dynamic scheduling algorithm is the random policy with temperature-aware adaptation [26] (referred to as Adaptive-Random). The Adaptive-Random algorithm uses load balancing based on the historical and current temperature to reduce hot spots and temperature gradients, with minimal additional complexity in the scheduler. Two benefits of Adaptive-Random are highlighted: 1) A low computational overhead making the algorithm suitable for use in the operating system; 2) Better load balancing than that which is achievable by making a decision based solely on the instantaneous temperature.

Adaptive-Random updates workload core probabilities at each scheduling interval based on the recorded temperature history on the chip. For example, given two idle cores at the same temperature, the coolest policy (described in Section 2.4.3.1) would not differentiate between cores. However, Adaptive-Random would prefer the core which had a lower average temperature in its past history window. The rationale is that the lower average temperature in the history of one core suggests this core and its neighbours have been under lower thermal stress. The probability of allocating the task of each core is updated based on these history temperatures: if the temperature is

higher than the threshold in the past window, the probability is set to 0; if the temperature is a little lower than the threshold, the probability is not updated; and if the temperature is low enough (lower than a pre-defined value), the probability is increased by $\beta/T_{avg}$ , where $T_{avg}$ is the core's average temperature in the history window and $\beta$ is an empirical parameter. Cores with higher probabilities will then be randomly chosen for task allocation.

#### 2.4.3.3 Look-Forward

Coskun et al. [70] proposed an autoregressive moving average (ARMA) predictor to estimate the temperature in next time slot based on the current temperature reading from thermal sensors and the workloads of the running task. According to the estimated temperature in the next time slot, the thermal-aware scheduler can carry out the task allocation, migration and DTM on cores to reduce the bad thermal effects.

A number of observations and assumptions are made: 1) The workload characteristics are correlated during short time windows; 2) The temperature changes slowly due to the large thermal time constant; 3) The underlying data for the ARMA model is stationary. Therefore, a set of workload data with a fixed pattern can train the model by correlating the work load data in a short time window. Three steps are carried out to build the model by training: 1) order identification, 2) coefficient calculation and estimation and 3) model checking. Test showed that the prediction error of the ARMA model is less than 10%.

If the workload changes and the trained model no longer fit the runtime workload, it must be dynamically adapted. Therefore, a method that can detect the workload changes over time was developed by using statistical characteristics of the residual signals. As a result, this method can predict and detect the future workload changes so as to estimate the future temperature trends.

In addition, proactive thermal-aware scheduling, in both DVFS and migration scenarios, was implemented. This proactive scheduling was compared to several reactive policies without using prediction. The results (using both a simulator and an UltraSPARC T1 processor) demonstrate that their proactive temperature-aware task

allocation for MPSoC is able to significantly reduce the adverse thermal effects with a low calculation overhead.

However, there are some disadvantages. Firstly, this approach still relies on the thermal sensor on each core. Secondly, the predictive time interval is relatively long, usually 500ms, or more, ahead of the current time instant. A long interval is needed as building and adapting the ARMA model requires hundreds of milliseconds, and thus this technique is not feasible for fine-grained scheduling. Thirdly, the predicted temperature is still used to passively trigger the high overhead DTM (e.g. DVFS in this case). Lastly, the ARMA model heavily depends on the training data set. It may deal well with repeated or fixed workload patterns, but does not perform as well with arbitrary workloads.

Yeo et al. [71] also developed a predictive thermal model (called PDTM) that can be used to guide real-time scheduling. Their PDTM is composed of two parts: the application-based thermal model (ABTM) and the core-based thermal model (CBTM).

ABTM is a training model that uses the recursive least square method to estimate the coefficients of a linear polynomial which builds the relationship between workloads and temperature directly. ABTM is very similar to the linear regression training [69] and is used to estimate the temperature according to the system event patterns of the training application set. After training, this polynomial can predict the short interval temperature $T_{app}$ according to task (application) workloads profiled offline. CBTM is a simple thermal model that assumes an exponential temperature transient on each core in the multiprocessor. This simplified thermal model (given in Equation 2.16) is similar to Equation 2.8 and is only applicable to uniprocessors:

$$T_{core}(t) = T_{ss} - (T_{ss} - T_{init}) \times e^{-bt} \tag{2.16}$$

where $T_{core}(t)$, $T_{ss}$ and $T_{init}$ are the transient temperature, steady temperature and initial temperature, respectively. $b$ is the temperature time constant.

ABTM and CBTM are then combined together to predict the temperature in the near future using a simple heuristic accumulation, as:

$$T_{predict} = w_s T_{app} + w_l T_{core} \quad w_s + w_l = 1 \tag{2.17}$$

where $w_s$ and $w_l$ are the empirical weights to balance between ABTM and CBTM. After obtaining the predictive temperature in next time slot, the scheduler can actively move a task from a hotter core to a cooler core, or decrease the task's priority on the hot core while increasing the task's priority on the cold core. It is shown that PDTM outperforms the original DTM in reducing the average temperature by about 7%, the performance overhead by 0.15%, and the peak temperature by about 3ºC.

This algorithm has similar disadvantages with [70]: Firstly, it still relies on thermal sensor readings and the long predictive time interval is only applicable in a very coarse-grained way. Secondly, the accuracy of ABTM heavily depends on the training task set. Lastly, CBTM is not accurate as the simplified exponential thermal model cannot be applied in a multi-core scenario due to inter-core thermal coupling [92][65].

DTAS is a relatively new research topic, with little in common with the more mature DTM policies. Each of the DTAS techniques in the three categories have some weaknesses: 1) Look-current techniques (e.g. coolest and neighbour-aware) only consider the present temperature information, and do not try to use future core temperature; 2) Look-backwards is based on intuitive and empirical conclusions (that a lower temperature in the thermal history would be better for task allocation) and can result in an inferior thermal profile compared to Look-current [92]; 3) Current look-forward implementations (e.g. ARMA and PDTM) require thermal sensors and have a long predictive interval making them unsuitable for fine-grained scheduling. The look forward prediction relies on the training workload which can impact on accuracy. Additionally, the thermal model used in PDTM is oversimplified and is not accurate for multiprocessor systems.

## 2.5 Discussion

After investigating the related literature on high level thermal optimization and management, we have identified several areas that could be further developed.

Firstly, the computational efficiency and accuracy of the power/thermal model is very critical in all the thermal simulation, estimation and optimization topics mentioned above. The thermal models used in current research have several shortcomings: 1) Some thermal models (e.g. TEMPEST) are only used for uniprocessors, and cannot be applied to a multi-core scenario; 2) Some thermal models (e.g. TSIC) use inaccurate heuristic or empirical thermal modelling that is very hard to adapt for different layouts and architectures; 3) Some thermal models [93][94] used in STAS only consider the steady-state temperature and ignore the transient temperature because of the limitations in the solving algorithm (e.g. linear programming), and thus the temperature error is large, even when using a fine-grained time step; 4) Some thermal models (e.g. HotSpot or the FEM approach) provide an accurate temperature estimation when used for offline thermal simulation, but require a computational overhead which makes them completely unsuitable for online simulation (e.g. with DTAS); 5) Current thermal models update the temperature in a time-triggered manner (that is, with a fixed time step) and thus introduce additional (and possibly unnecessary) computational overhead; 6) Much of the research related to thermal-aware scheduling [90][92][57] assumes that the leakage power is invariant (that is a non-temperature-dependent leakage model is assumed). While these assumptions simplify the problem by skipping the power-temperature iteration, they can lead to unrealistic thermal results. Therefore, current thermal estimation and modelling does not sit well amongst the various competing factors: e.g. computational efficiency, estimation accuracy, updating period and leakage power.

Secondly, STAS, while being more mature than DTAS, still suffers from the use of inaccurate or overly computationally complex thermal models. The use of inaccurate models severely limits STAS's practicality, while the use of computationally complex models limits the size and the types of problems that can be addressed. For example, to the best of our knowledge, there is no STAS schedulability test for multiprocessor due to the high computational overhead of current thermal models (a schedulability test without any thermal consideration is already an NP problem). Simplifying the thermal model may allow this to be better addressed.

Lastly, to be effective, DTAS requires fine grained real-time temperature information from either thermal sensors or thermal models. As discussed previously, the relatively

long thermal response latency associated with DTS makes them unlikely candidates for directly driving DTAS. Thermal models (or thermal models calibrated by DTS) may be more appropriate. However, the overhead of these thermal simulators must be kept as low as possible so that they can be integrated with the task scheduling process.

Having reviewed the current research, it is fairly obvious that the main limiting factor in high level thermal optimization and management is the thermal feedback. If our aim is to improve high level thermal optimization and management, then it is important that we address this issue.

Specifically, in the subsequent chapters, this thesis further develops and addresses the following research topics:

- To improve the performance and accuracy of the power/thermal model, we propose a fast event-driven thermal estimator which can be used in both online and offline scenarios. This thermal estimator will need to have a very low computational overhead, with an acceptable accuracy, to be suitable for fined-grained DTAS. As leakage power should not be ignored, a temperature-dependent leakage model should be integrated into the fast thermal estimator to achieve better thermal accuracy. Moreover, this fast thermal model should be able to be combined and calibrated by direct thermal sensor readings to eliminate any long term temperature drift.
- For STAS, a schedulability test for multiprocessor is investigated by using our fast thermal estimator with a full transient temperature analysis. This analysis will optimize for different goals (e.g. performance, peak temperature) under a strict temperature threshold in a real-time embedded system scenario.
- For DTAS, we adapt our fast thermal estimator using the temperature-dependent leakage model into a fine-grained scheduler. This would then allow us to quickly predict the future core temperature, and thus enable us to propose several look-forward policies to achieve better thermal effects.

# Chapter 3

# A Fast Event-Driven Thermal Model



As discussed in Chapter 2, the thermal models used in recent research have a number of disadvantages that limit how the model can be applied at the system level. These include:

- An oversimplified model: The uniprocessor model [57] and its simple exponential thermal function [71] are assumed to be correct for the multiprocessor case. Thermal coupling between cores is ignored which can introduce significant errors [92][65][35].
- No transient temperature calculation: Many high level DTM and TAS mechanisms only use the steady-state temperature calculation for guidance, constraints and goals, and avoid the complexity of a transient temperature analysis [57][93][94][95][102]. Ignoring the thermal capacitance in the RC model, particularly when the temperature time constant is large and the task duration is relatively small will introduce errors [35].
- Training-based and empirical analysis: The accuracy of these techniques heavily depends on the training data set [70][71] and the empirical parameters [92], and generally have a significant overhead.
- Large overhead for thermal simulation: High accuracy thermal simulation requires significant computing resources and compute time. This overhead makes the fine-grained online scheduling necessary for both DTAS and STAS difficult to implement.

To solve these problems, we propose an event-based approach that uses a pre-built LUT to evaluate the temperature more efficiently. We show that this approach can provide similar accuracy to that of a widely-used high accuracy thermal simulator, HotSpot [35]. In the next subsections we introduce some of the simplifying

assumptions which can further reduce the complexity of our proposed event-based approach. We start by introducing the fundamental model of the LUT for non-leakage-dependent power (dynamic power). We then examine using the LUT-based temperature estimator for fast real-time scheduling (in Chapter 4), and then we formalise and extend it into a DTAS scenario involving leakage-dependent power (in Chapter 5).

## 3.1 Methodology and Metrics

We use a combination of both real applications and synthetic task sets to test our fast thermal simulator, and in the TAS experiments in following chapters. The real applications are first translated to a continuous power profile, and then to atomic power events. In all cases, we start with ANSI C source code. This is then compiled to the target architecture using GCC (version 2.95.3 for the ARM architecture used in the low power scenario in Chapter 4, and version 3.4.6 for the Alpha architecture used in the high power scenarios in Chapter 5. These were chosen as they are supported by SimpleScalar version 4.0 [41]. The binary image produced by GCC is executed on SimpleScalar and Wattch [43] to obtain a continuous power profile. In Chapter 5, we use our modified SimpleScalar-ARM for multiple ARM cores as described in Appendix A, while in Chapter 6 we use SimpleScalar version 4.0 and Wattch version 1.2 for the Alpha architecture.

In Chapter 3.2, we use the continuous power profiles generated above as the power input to HotSpot (version 5.0). We then use the highly accurate core-level temperature profile produced by HotSpot to compare with our fast LUT-based thermal estimator. As our estimator uses atomic power events (rather than a continuous power profile) as input, we need to convert the continuous power profile to an atomic power profile.

The synthetic task sets (used in Chapter 4 and 5) are randomly generated atomic power events. These synthetic power profiles are used as input to both HotSpot and our LUT-based thermal estimator for comparison purposes. The synthetic task sets are generated using Matlab code.

To evaluate the accuracy and overhead of our LUT-based approach, we use the

following metrics: average temperature error (defined as the sum of the temperature differences divided by the number of compared samples), maximum temperature error, overall runtime (between other simulators and ours), and the average overhead of a thermal map update (this gives an indication of the task schedule overhead). To evaluate TAS performance, we use the following metrics: peak temperature, average temperature, and average spatial/temporal diversity, which are defined in [92].

## 3.2 Power Events and their Profiling

In much of the literature relating to TAS [90][92][91][93][94], the average power consumption of a task, or the section of a task, is obtained by simulation and is stored for later use by the allocation and scheduling algorithms. Based on our power profiling observations of a number of different applications, we observe that:

- The power consumed by a task varies in the short term (it can even be different between clock cycles), but rapid power variance in the short term (less than several microseconds) is not able to introduce a large change in the core temperature due to the thermal mass of the core (usually the time constant of a typical CPU core is larger than 10 seconds[8]. The thermal mass of the core means that the core level power consumption of an entire task can be treated as a series of averaged power values, delineated by the significant power changes.
- The power consumption changes dramatically only after the occurrence of some system event (i.e. allocation, deallocation, context switch, preemption, memory access, DVFS, interrupt blocking, etc.), and the temperature of the core experiences a temperature change following these significant power changes. The same observation is also introduced in [94]. Therefore, it can be concluded that the average power of a task, separated by the sudden changes in power, may be a good abstraction and assumption to use for high level TAS [90][94].

[8] The duration from initial temperature change to steady-state temperature is usually long. Some chips (e.g. Intel Core i7 or Nvidia Fermi 100) take 35-75 seconds. However, the time constant does not means that the chip temperature is unchanged during this interval.

As a result, a continuous power profile may be able to be simplified to a sequence of constant power events. The definition of a power event used in this thesis is:

**Definition 2 (Power Event)**: *A power event is associated with the (relatively instantaneous) increase or decrease in power generated by a core. The power profile is described by, and converted to, a sequence of these power events.*

We refer to these significant power changes, which can be captured and maintained by the OS kernel, as high level power events. The high level power events include: task allocation, deallocation, context-switch, migration, preemption, stop-go and DVFS. Any high level event can be converted into one or more power events eventually. For instance, task allocation and dealloction can be seen as power events, with an instantaneous power change at the beginning and end of the task; preemption can also be treated as a decrease in power followed by a power increase, while DVFS can be regarded as a power increase when the clock frequency or voltage rises and a power decrease when they are scaled down. Thus, instead of using a time driven methodology which updates the core temperature at fixed time intervals, like HotSpot or TSIC, we use an event driven thermal model to update the core temperature only when an event occurs. These events are managed dynamically by the scheduler.

Power events can be determined by profiling the power consumption of an application (or task). In Chapter 2, we discussed both the simulator-based and counter-based approaches to obtain the power profile of a task. In a conventional simulator, the sampling resolution for determining power consumption can be customized by user. Usually, the power sampling rate is around 100us to 1ms, as a very fine-grained power profile is not necessary for high level optimization due to the thermal mass of the system. As we are using an event driven approach, we use the average power consumption between consecutive power events. The power events are then the critical points where large power variations can induce significant temperature changes in the core. We define a threshold value $P_{AE}$, to determine power events, with a smaller $P_{AE}$ resulting in a finer-grained power profile. For example, a more coarse grained power profile is likely to be suitable for DTAS, while a finer granularity would be more appropriate for thermal simulation. Figure 3.1 shows two different power profiles decomposed into atomic events for different $P_{AE}$ values. The dashed lines represent the original power profile and the solid lines show the power profile

approximated as power events.

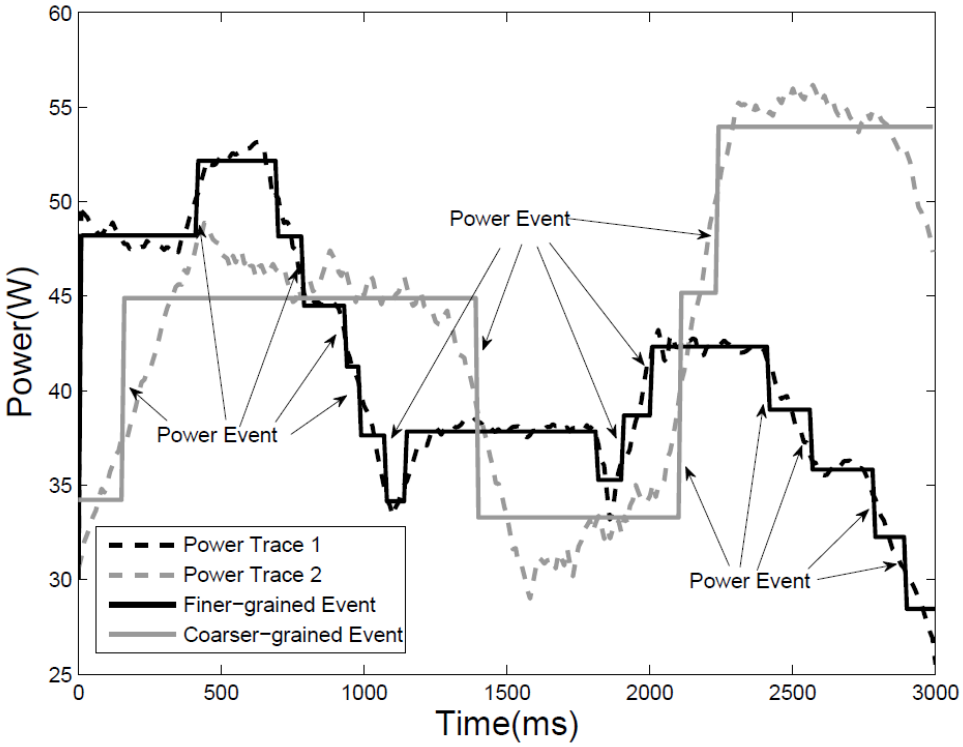



Figure 3.1: Power event profiling

Algorithm 3.1 shows how to use $P_{AE}$ to segment a continuous power profile. Every new sample is compared to the average power, $avgP$. That is, the average power value from the last event, $lastAE$, to the current sample point. If the difference between the current power sample value $P_t$ and average power value $avgP$ is larger than $P_{AE}$, a new power event is generated and recorded, as indicated in Figure 3.1. This algorithm is relatively simple and can be easily integrated into either an offline or online power profiling application, with no need to traverse the whole sample sequence. Therefore, the time complexity of Algorithm 3.1 for offline profiling is $\Theta(n)$, where $n$ is the number of sampling points, while for the online case, the time complexity is $\Theta(1)$. In Algorithm 3.1, one power event is denoted by $ae(t, P, loc)$, where $ae.t$ is the time point of event occurrence, $ae.P$ is the power increment or decrement (a positive number indicates an increment, while a negative number indicates a decrement) and $ae.loc$ indicates the location of the power event (usually, it represents a core ID or node ID).

**Algorithm 3.1: Power Event Profiling($P_{0\ldots t}, P_{AE}$)**

**Input**: $P_{0\ldots t}$ is the power sampling value from start to current time $t$, $P_{AE}$ is the threshold
**Output**: an array of power events, element is $ae(time, power, location)$

$lastAE = 0$;
$avgP = P_0$;
$sumP = 0$;
**WHILE** $t \geq 0$
  **IF** $|P_t - avgP| \geq P_{AE}$ **THEN**
    Generate a power event and record $ae(t, P_t, loc)$ in power event array;
    $avgP = P_t;\ lastAE = t; sumP = 0$;
  **ELSE**
    $sumP = sumP + P_t$;
    $avgP = sumP/(t - lastAE)$;
  **END IF**
  $t = t + 1$;
  Sample the next power consumption to $t$;
**END WHILE**

We next examine the effect of using atomic power events on the thermal characteristics on a single processor core. To validate that the average atomic power profile produces a temperature effect with similar accuracy to that of the original power profile, two experiments are carried out: 1) the synthetic continuous power input from Figure 3.1 is converted to atomic power events ($P_{AE} = 3W$) using Algorithm 3.1, and 2) an MPEG2 decoder power profile, using SimpleScalar and Wattch for Alpha 21264 [35], is generated and converted to atomic power events ($P_{AE} = 1.2W$). Both power profiles are input to HotSpot [35] and the corresponding temperature profile is obtained. In both cases, the temperature/leakage power dependence is not considered. HotSpot uses the default thermal parameter settings, and the die area and heat sink area are 256mm$^2$ (16mm×16mm) and 900mm$^2$ (30mm×30mm), respectively. Both the original continuous power input and the atomic power profile (obtained from Algorithm 3.1) are plotted as the lower two curves in Figure 3.2 and Figure 3.3. The two upper curves in Figure 3.2 and Figure 3.3 show the temperature traces obtained from the original continuous power profile and the atomic power profile. Examining the temperature traces of both figures, we see that there are instantaneous differences in temperature, but the long term characteristic is similar in both cases. The average temperature and worst case temperature errors for the first experiment are 0.021°C and 0.08°C, respectively. While the average temperature and worst case temperature errors for the second experiment are 0.0085°C and 0.081°C, respectively. These two experiments show that converting a power profile, with an appropriate granularity ($P_{AE}$) produces short term inaccuracies, but in the longer term is accurate enough to represent the original

continuous power profile.

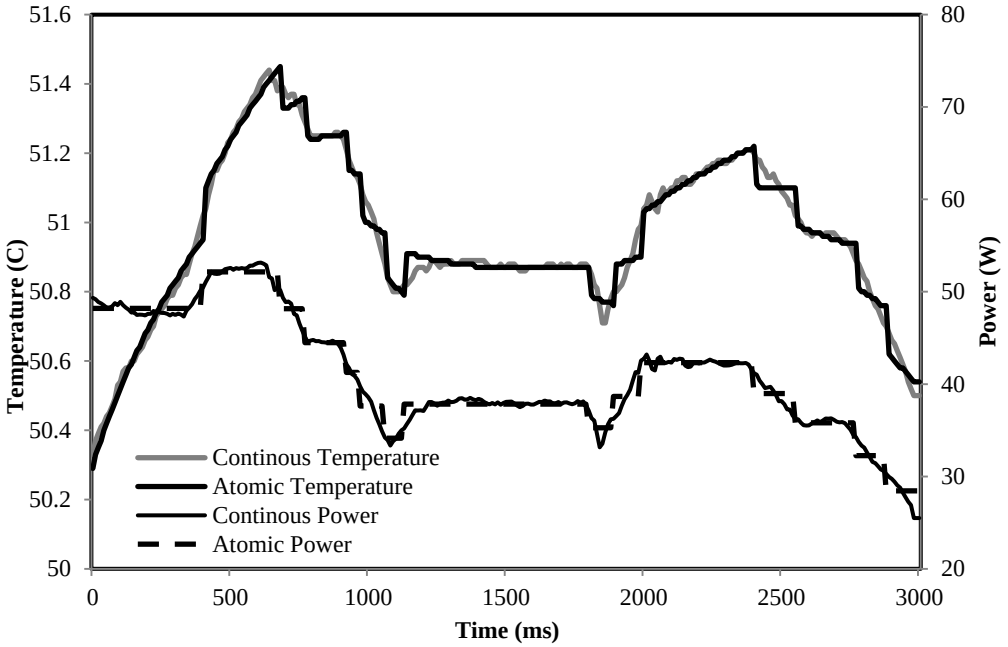



Figure 3.2: Temperature Trace Comparison for Synthetic Power Input

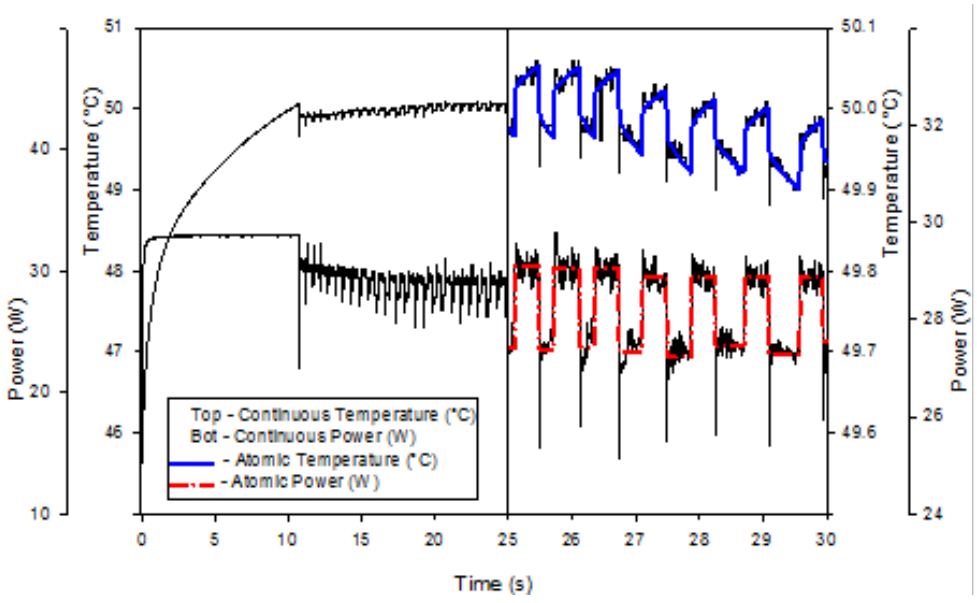



Figure 3.3: Temperature Trace Comparison for MPEG2 decoder power profile

## 3.3 Thermally Different Location

A reasonable abstraction suitable for high level thermal-aware optimization, is to treat

the multiprocessor chip as several rectangular regions, where each rectangular region represents a complete processor (with cache, local memory and an inter-communication block such as a network-on-chip router), as shown in Figure 3.4 [57][94][92][90]. Usually, most chip layouts have a symmetrical geometric pattern. This is particularly the case for many-core processors. We explicitly define the concept of thermally different location (TDL) which is proposed in [92], but has no formal definition.

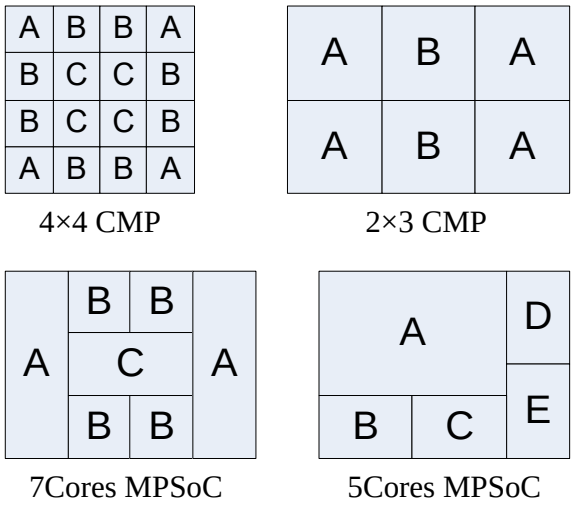



Figure 3.4: Core/module level abstraction

**Definition 1 (Thermally Different Location)**: *A thermally different location (TDL) is defined for sets of homogeneous cores which are symmetrical in their relative location, and thus have similar thermal characteristics. The important implication of TDL is that if a task is allocated to a core in a TDL, it will produce the same thermal effects and thermal distribution as it would when allocated to another core of the CMP (or MPSoC) with the same TDL.*

The TDLs for a number of abstracted CMP and MPSoC systems are given in Figure 3.4. Here, the different characters in the core regions denote the TDLs, with the cores having the same TDL (same character) being both symmetrical in layout and having identical architectures. We will use this concept when we build the LUT, as the number of TDLs in the layout decides the number of LUTs that need to be built. It should be noted that in an MPSoC with zero core (PE) regularity, there will be no common thermal locations. However this technique is meant to be more general and as such, TDLs are a useful means for reducing computational overhead in more

regular structures.

## 3.4 Thermal Model

The abstracted set of rectangular regions representing the individual cores in a many-core system can be represented as a thermal RC network, described by a set of differential equations. This thermal RC model is a good choice for high level estimation for both 2D and 3D ICs due to the accuracy and efficiency of thermal estimation [11][35]. HotSpot solves a thermal RC model, at a fixed simulation interval, resulting in a large computational overhead. We have already described the detailed thermal model (Figure 2.2) and its corresponding differential equation set (Equation 2.9) in Chapter 2.2.1. In this work, we use the same thermal RC network for building the LUTs.

Here, we give a simplified example of a 2×2 CMP thermal RC network and its equation set, shown in Figure 3.5 and Equation 3.1. In this simplified model, there are only 4 power input nodes (black dots in Figure 3.5 representing the cores) in the silicon layer which in this case is directly connected to the ambient, without any heat sink or heat spreader. Equation 3.1 is easily obtained by Kirchhoff's circuit laws, and for any node, the algebraic sum of the currents flowing in or out is zero. $T_i'$ is the first order derivative of the temperature versus time, $P_i$ is the power input.

$$\begin{cases} P_1 + \dfrac{T_2 - T_1}{R_{1,2}} + \dfrac{T_3 - T_1}{R_{1,3}} + \dfrac{T_{amb} - T_1}{R_1} + C_1 T_1' = 0 \\ P_2 + \dfrac{T_2 - T_1}{R_{1,2}} + \dfrac{T_4 - T_2}{R_{2,4}} + \dfrac{T_{amb} - T_2}{R_2} + C_2 T_2' = 0 \\ P_3 + \dfrac{T_3 - T_1}{R_{1,3}} + \dfrac{T_4 - T_3}{R_{3,4}} + \dfrac{T_{amb} - T_3}{R_3} + C_3 T_3' = 0 \\ P_4 + \dfrac{T_4 - T_3}{R_{3,4}} + \dfrac{T_4 - T_2}{R_{2,4}} + \dfrac{T_{amb} - T_4}{R_4} + C_4 T_4' = 0 \end{cases} \tag{3.1}$$

In the common case, the ambient temperature $T_{amb}$ is a constant, and the inter-core thermal resistances $R_{i,j}$ for a homogenous layout with identical material are also identical, as are the thermal capacitances $C_i$ and the thermal resistances between core and ambient, That is, $R_{1,2} = R_{2,3} = R_{3,4} = R_{4,5}$, $C_1 = C_2 = C_3 = C_4$ and $R_1 = R_2 =$

$R_3 = R_4$. This is identical to the TDL concept in Section 3.3, in that an identical power injected into any of the cores will produce the same thermal effect due to the symmetrical layout and identical material[9].

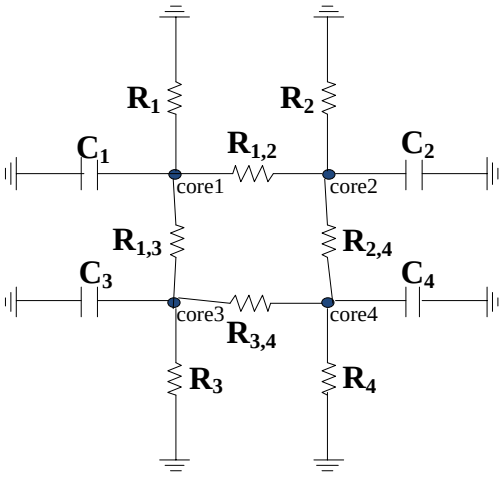



Figure 3.5: Thermal RC network for a 2×2 CMP

If only a steady-state analysis is required, without considering the transient temperature, the thermal capacitances of Equation 3.1 can be removed, and the equation set becomes a purely linear equation set. However, in most cases, due to the large thermal mass, the thermal capacitances in Equation 3.1 cannot be ignored [11]. Such an equation set is referred to as a linear ordinary differential equation (LODE) set. We introduce the following theorem to clarify our thermal model.

**Theorem 1 (LODE)**: *If the variance of the thermal resistances and capacitances are ignored (that is the resistances and capacitances are considered constant[10]), the thermal RC model for a multiprocessor at the core-level is a linear model that can be described by the linear ordinary differential equations (LODE) given in Equation 2.9.*

This means that we can apply the same techniques for LODEs, such as the superposition principle, to the thermal RC network. However, the time taken to solving a complex LODE makes this thermal simulation very inefficient, and as a result we examine an alternative technique for evaluating the thermal model.

[9] Chip manufacturing variation is not considered in this example, but the LODE model can deal with these variations which effect the thermal parameters (e.g. R and C values of individual cores). If the variation is large, it may no longer be appropriate to use TDLs.
[10] When the chip temperature varies over less than 50℃ (for instance, 50℃--100℃), the assumption that the resistance is constant is reasonable [11]. This assumption is also used in Hotspot [11].

At this point, it should be noted that while chip and manufacturing variations are important practical issues, this work focuses on using an existing accurate thermal model and accelerating the temperature evaluation in a multi-core scenario with minimal loss of accuracy. If chip/manufacturing variations were significant, it would be a relatively simple matter to perform a simple experiment to calibrate the model.

## 3.5 Prebuilding the LUT

To enable rapid temperature calculation, several LUTs are pre-built offline. For a given multicore layout, the equivalent RC network is determined by analysing the relationships between adjacent core regions, as presented in the last section. The thermal resistances and capacitances are obtained from research experiments or technical documents [27][35]. These physical parameters are independent of our methodology.

To build the LUTs, we assume that 1 Watt of power is injected into a core of a TDL, with no power being applied to the other cores, and determine the temperature transient at each core by solving Equation 2.9 for a fixed time interval (say 10 milliseconds). A LUT then reflects the temperature increment of each core, relative to the initial temperature, after applying 1Watt of power to any core of an individual TDL. In fact, the LUT can be regarded as the system response to a unit-step (1 Watt Power injection) stimulus. However, one LUT only reflects the response of the stimuli on a certain core. If we want to know the full system response, the number of generated LUTs should be equal to the number of distinct cores. As a result, a 2×2 CMP needs four LUTs. Obviously, as the number of cores increases, it would require a lot of memory space to store these tables. Thanks to the TDL, the number of generated LUTs can be greatly reduced since the same power stimulus on any core in a TDL has the same thermal response. The number of distinct TDLs determines the number of LUTs that need to be generated. For example, a 2×2 CMP needs only one LUT because there is only one TDL associated with this layout, while a 2×3 CMP requires two LUTs for the two different TDLs (corresponding to TDLs A and B in Figure 3.4). The computation of the LUTs is carried out off-line, using tools such as Matlab or Maple, and as such, the efficiency of building the LUTs is not important.

Table 3.1 gives an example of the LUT for TDL A in a 2×2 CMP (1 Watt of power injected at Core1). The entire heating stage after a 1Watt power input is recorded row by row, with the last row of the table indicating the steady temperature of each core. In each time interval (10ms in Table 3.1) the relative temperature increment of each core is recorded as a row of the LUT. $Steady$ is the time point when the temperature increment reaches the steady state and stops increasing, and depends on the thermal time constant[11]. It should be noted that the LUT only represents a unit power input, the values in the LUT should be multiplied by the actual power input to reflect the correct temperature increment.

| TDL A | Core1 | Core2 | Core3 | Core4 |
|---|---|---|---|---|
| 0ms | 0 | 0 | 0 | 0 |
| 10ms | 0.1553 | 0.0007 | 0.0007 | 0.0000 |
| 20ms | 0.1788 | 0.0012 | 0.0012 | 0.0000 |
| 30ms | 0.1844 | 0.0016 | 0.0016 | 0.0001 |
| 40ms | 0.1880 | 0.0019 | 0.0019 | 0.0001 |
| 50ms | 0.1912 | 0.0021 | 0.0021 | 0.0001 |
| 60ms | 0.1943 | 0.0024 | 0.0024 | 0.0001 |
| 70ms | 0.1971 | 0.0028 | 0.0028 | 0.0002 |
| … | … | … | … | … |
| 500ms | 0.2013 | 0.0648 | 0.0648 | 0.0446 |
| 520ms | 0.2014 | 0.0649 | 0.0649 | 0.0447 |
| 540ms | 0.2015 | 0.0650 | 0.0650 | 0.0448 |
| … | … | … | … | … |
| 1000ms | 0.3233 | 0.0854 | 0.0854 | 0.0639 |
| 1050ms | 0.3235 | 0.0856 | 0.0856 | 0.0640 |
| … | … | … | … | … |
| 2000ms | 0.3504 | 0.1116 | 0.1116 | 0.0891 |
| 2100ms | 0.3506 | 0.1118 | 0.1118 | 0.0893 |
| … | … | … | … | … |
| Steady | 0.3819 | 0.1428 | 0.1428 | 0.1200 |

Table 3.1: Look-Up Table for a 2×2 CMP

If we use a uniform time interval (e.g. 10ms) to record the temperature increment in the LUT, a single LUT will require a significant amount of memory, particularly if $Steady$ is large. To reduce the storage requirements for the prebuilt LUTs, Table 3.1 uses a non-uniform time interval. A smaller time step (we use 10ms)[12] is used initially, which is increased (e.g. a 20ms time step at 500ms, a 100ms time step at 2000ms, etc.) as the temperature gets closer to the steady-state value. This is because the

[11] In a multi-node thermal RC model, the time constant for one core is not as simple as the product of the single thermal resistance and capacitance.

[12] A reasonable time interval used to build the LUT is the minimal time interval between two consecutive scheduling rounds of the OS kernel, i.e. an OS timer tick.

temperature profile of the heating/cooling stage shows a more rapid initial change, with the rate of change in temperature slowing with time. By using this storage optimization, a large number of duplicated rows can be removed from table. Using a non-uniform time step can reduce the amount of storage required, compared with the original LUT size. This memory saving is shown in Section 3.8.

It is then likely that the time instant associated with the occurrence of a power event will lie between two consecutive rows (e.g. 35ms or 1020ms). Although it is possible to use linear interpolation or some other interpolation to get the temperature between time points, we show in Section 3.8 that the results obtained from just rounding to the nearest temperature value are sufficient (i.e. the temperature increment at 22ms could be rounded to the values in the 20ms row, 37ms rounded to 40ms, etc.). However, if a more coarse grained time interval is used, the accuracy will degrade and interpolation may need to be considered.

So far, only a step power increase has been considered. However, due to the symmetrical nature of the LODE, an increment and a decrement are treated equally and their solutions have an equal magnitude but with the opposite sign. As a result, a temperature decrement can simply be considered a negative increment. This is also verified in the literature [79]. Thus, just a single LUT can record both the temperature increment and decrement.

**Theorem 2 (Symmetrical Temperature Increment and Decrement)**: *The thermal responses induced by a power increase and an identical power decrease are symmetrical, and the resulting temperature increment and decrement have the same absolute values but with the opposite sign.*

## 3.6 Mapping Power Input to the Correct Core

We have used both TDL and optimized non-uniform interval to save on the storage needed for the pre-built LUTs. TDL is used to reduce the number of LUTs, and the non-uniform interval is used to reduce the number of rows in a LUT. One LUT describes the thermal response for a power injection at a single core. However, the power input can be injected into any core (node) in silicon layer. Therefore, we need

to map the corresponding thermal response according to power input location [35].

The TDL's symmetrical layout can help this mapping by using some transformation operation on it. A transformation of the temperature increment values is needed to cater for the relative location between the actual power input location (denoted by $ae.loc$) and the 1 Watt power input location (used when building the LUT) associated with the same TDL. We define a function $trans(LUT, ae.loc)$ to do this mapping operation, where $LUT$ is decided by the TDL which $ae.loc$ belongs to.

Function $trans()$ is relatively simple with eight possible coordinate transformations, being: 0: no change; 1: mid-x mirroring; 2: mid-y mirroring; 3: principal diagonal mirroring; 4: secondary diagonal mirroring; 5: centre point mirroring; 6: clockwise rotation; and 7: counter-clockwise rotation; as shown in Figure 3.6.

Figure 3.6 also shows an example using a 4×4 CMP. The 4×4 CMP requires 3 LUTs for TDL A (1Watt of Power at Core(0,0)), TDL B (1Watt of Power at Core(0,1)) and TDL C (1Watt of Power at Core(1,1)). In Figure 3.6, an event, $ae(10, 28, Core(0,2))$, occurs at Core(0,2). Because the temperature increment values from LUT B reflect the case when 1Watt of power is injected at Core(0,1), but $ae(10, 28, Core(0,2))$ needs the power increment at Core(0,2). Therefore, the function $trans()$ uses mid-y mirroring to map the correct thermal increment, as shown in Figure 3.6. In fact, the transformation only rearranges the order of the columns in a LUT. For example, swapping the column Core1 and Core4 (i.e. secondary diagonal mirroring) in Table 3.1 would obtain the thermal response for a power injection at Core4. The transformation can also be applied for MPSoC with symmetrical cores, such as the 7-Core MPSoC in Figure 3.4.

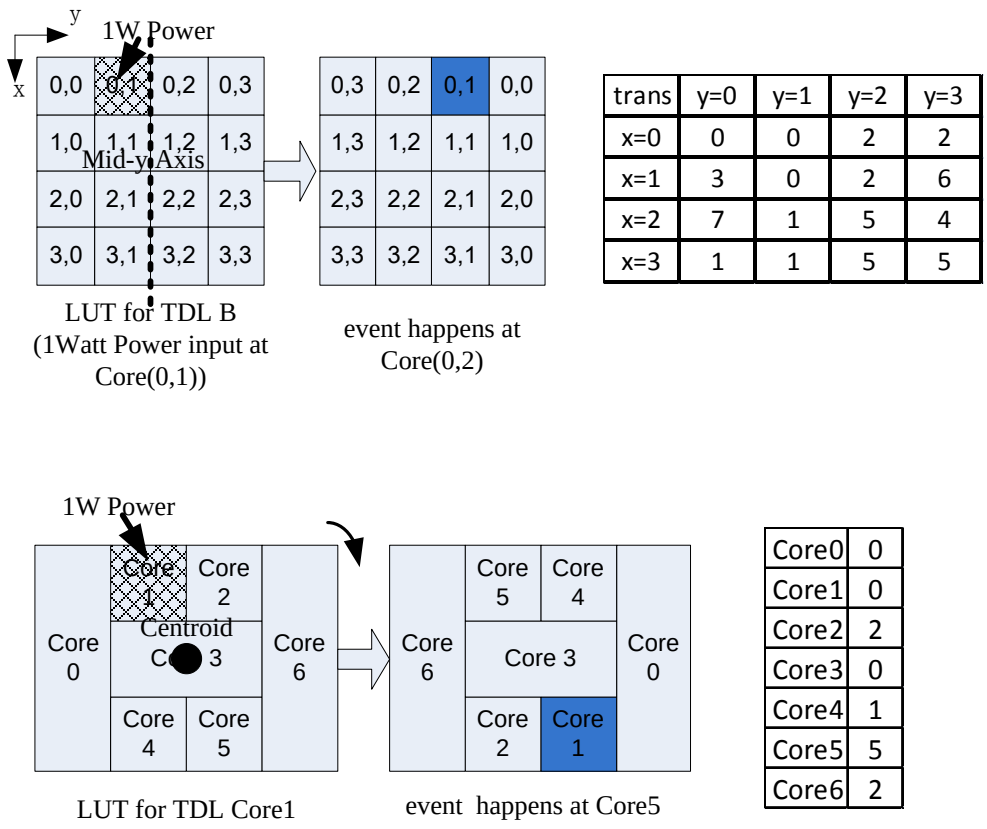



Figure 3.6: Transformation for Power Input Mapping

After obtaining the pre-built LUTs and defining the necessary transformation operations, any thermal response induced by a single power event can be easily determined. Figure 3.7 shows the procedure to get the resultant LUT (the thermal response) for any single power event. The obvious question is: if multiple power events can occur at any core and at any time, is it possible to determine the entire chip's thermal response by adding the individual thermal responses together? We will address this issue in the next section.

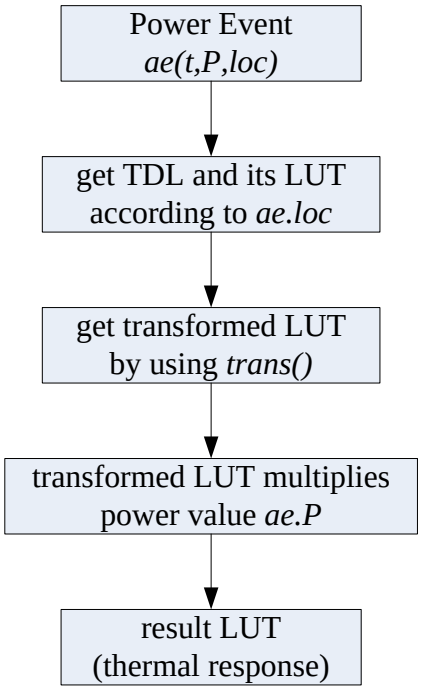



Figure 3.7: Thermal response for any single power event

## 3.7 Superposition Principle of the Thermal Response (LUT)

We have stated in Theorem 1 in Section 3.4 that a thermal RC network can be formalized as a LODE. The entire prebuilt LUT is a solution of LODE with only a single non-zero power, $P_i = 1$, and all other power sources are zero, i.e. $P_j = 0, j \neq i$, and where each row in the LUT represents a specific solution to the LODE at the indicated time instant. Regardless of the initial condition of the LODE, any solution of the LODE satisfies the superposition principle, and hence the multiple thermal response represented by the LUT also satisfies the superposition principle.

**Theorem 3 (Superposition Principle of LUT)**: *The entire thermal RC network's thermal response can be treated as the accumulation of every individual response induced by each power event. The entire LUT and its rows adhere to the superposition principle when power input is not affected by temperature*[13].

**Proof**: Only atomic power events can induce a thermal response. Firstly, the temperature in each row of a LUT must be a solution of Equation 2.9 since it is

[13] Leakage power is assumed as a constant. Later we will extend this to include varying leakage power.

obtained by solving Equation 2.9 at a specific time interval. Given a row $\mathfrak{R}$ in the prebuilt LUT, its transformed row $\mathfrak{R}^*$ is also a solution of Equation 2.9, and thus also satisfies the superposition principle. Then, given any two rows, $\mathfrak{R}_1^*$ and $\mathfrak{R}_2^*$, in the prebuilt LUTs, it follows that $\mathfrak{R}^* = P_1\mathfrak{R}_1^* \pm P_2\mathfrak{R}_2^*$, where $P_1$ and $P_2$ are constants (such as $ae.P$), must also be a solution of this LODE. This can be expressed more generally as: $\mathfrak{R}^* = \sum_i P_i\mathfrak{R}_i^*$, where $\mathfrak{R}_i^*$ can be seen either as a row or as an entire table.

According to Theorem 3, a temperature change is induced by the thermal response caused by a power event. Therefore, if the leakage power is assumed to be a constant, the temperature for each core can be updated when a power event occurs. This event-driven temperature estimation is different from the conventional time-driven approach that updates the temperature at a fixed time step. It is then possible to perform a number of different operations on these prebuilt LUTs.

- Adding several LUTs together ($\mathfrak{R}_1^* + \mathfrak{R}_2^*$): the LUTs can be aligned to the time points of power events, and can then be added together on a row-to-row basis. The result of the addition can show the overall thermal increment and the entire temperature transient induced by all these power events over time. This is referred to as a "table operation" in later chapters. Since the table for one power event represents its temperature transient, the tables need to be aligned to the absolute time instants before adding them together. If one event occurs at $t_1$, and another event occurs at $t_2$, the first rows of the two tables (representing their thermal response) should be aligned at the time points $t_1$ and $t_2$, respectively, on the time axis, and then they are added row by row. The resultant table (whose length is larger than the individual ones) shows the accumulated thermal response induced by the two events. Due to the larger memory space needed by the table alignment, this operation cannot be applied for long term temperature evaluation. The table operation can be applied where the full on-chip thermal distribution is needed, and is more suitable for a STAS scenario to check when and where overheating occurs. The detailed description of a table operation is formalized in Chapter 4 where we apply it to a STAS scenario.
- Subtracting two rows in one LUT ($\mathfrak{R}_1^* - \mathfrak{R}_2^*$): the result of the subtraction can show the temperature increment induced by one power event during the time

interval between two corresponding rows. This is referred to as “row operation” in later chapters. For an individual power event occurring at time point $t$, if the last thermal map update occurs at $t + 20ms$ and the temperature evaluation point is set to $t + 50ms$, then the two rows, the 20ms-row and the 50ms-row, can be fetched from the LUT and subtracted to obtain the temperature increment during the time interval from $t + 20ms$ to $t + 50ms$. Lastly, all of these temperature increments induced by the individual power events can be accumulated to update the entire thermal map relative to the last updated thermal map. This operation has nothing to do with absolute time instances, and it is easy to determine the temperature contribution from one power event between any two relative time points (e.g. consecutive update time points). Thus, it is suitable for continuous temperature evaluation. Moreover, its low computational overhead enables online (e.g. dynamic) temperature estimation which is ideal for fine grained DTAS. The detailed description of the row operation is formalized in Chapter 5 where we apply it to a DTAS scenario.

Both these operations have a low calculation overhead compared to any of the time-driven approaches, and thus are ideally suited to STAS and DTAS. While we next proceed to examine the accuracy of the LUT-based thermal estimation, the validation of these two operations will be carried out after we formalize and detail them in their respective STAS and DTAS scenarios.

## 3.8 Validation of the Generated LUT

Before we formally introduce our LUT-based thermal estimator in STAS and DTAS scenarios, we perform a number of experiments to show how effective our event-driven approach is compared to HotSpot.

Firstly, as we have claimed that existing simulator based estimators are too slow for TAS scenarios we examine the runtime for our proposed estimator compared to that of Hotspot. In this experiment we use a 4×4 CMP layout and a power task set with an average power consumption in the range [15W:30W] using a power threshold $P_{AE} = 5W$, and with execution times in the range [10ms:500ms]. To reduce the required

simulation time, we assume that the core has already been heated above ambient temperature. That is, we assume that the processor has been operational for some time and that the core temperature is at an initial temperature of 45°C ($T_{t=0ms}$= 45°C). The HotSpot iteration time interval is set to 10ms and we use a non-uniform 10ms minimum interval LUT. This experiment is identical to the one conducted in Chapter 5.7.2, except that the temperature-dependent leakage power is not taken into account.

The thermal estimation based on atomic power events, along with the Hotspot thermal estimation is shown in Figure 3.8. There is good agreement between the two estimators with a maximum error of less than 0.4°C. More importantly, there is a significant reduction in runtime, from 1.46s for Hotspot to 728μs for the proposed LUT based thermal estimator, shown in Table 3.2. This is partially because there are only 36 events in the atomic simulation, compared to 500 events in the Hotspot simulation for the 5s period, and partially because of the two orders of magnitude reduction in the overhead per iteration update, from 2922.4μs per update for Hotspot to 20.2μs per update for the proposed method. This represents a 2000 times improvement in the overall runtime.

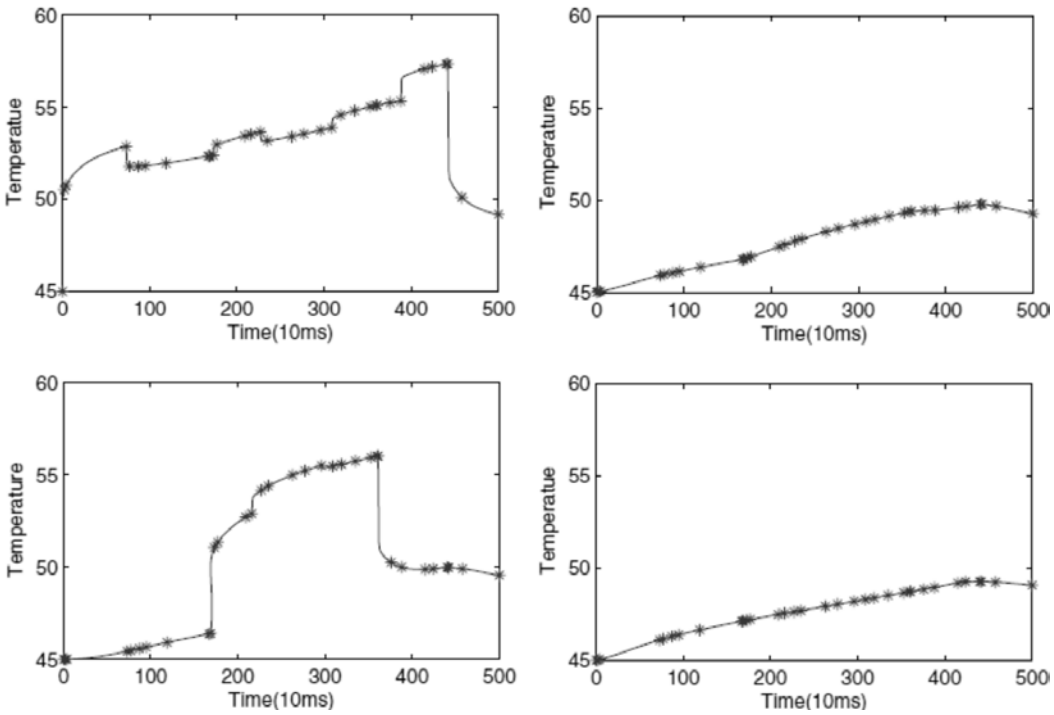



Figure 3.8: Event-driven temperature estimation comparing with HotSpot

| Thermal Simulation | LUT based Simulator | HotSpot Simulator |
|---|---|---|
| Overall Runtime | 728 μs | 1461215 μs |
| Number of Iteration Events | 36 | 500 |
| Average overhead per update | 20.2 μs | 2922.4 μs |

Table 3.2: Event-driven calculation overhead and runtime comparing with HotSpot

Next, we compare the temperature from our non-uniform interval LUT-based thermal model with the temperature obtained from HotSpot [35] on a 4×4 CMP layout to verify that the LUT method is accurate. As the row and table based operations (to be used in later chapters) rely on the fundamental accuracy of the LUTs, if the LUT itself is inaccurate, the results from the combination of multiple operations will significantly affect the overall temperature estimations in both STAS and DTAS.

In HotSpot, we set a 1ms time interval for transient temperature analysis and inject a 1Watt power input to Core(1,1) (belonging to TDL C). We use 4 different resolutions to generate the non-uniform interval LUTs to test their accuracy. Table 3.3 gives the interval details for the non-uniform interval LUTs. In all cases, steady state is approximately 12 seconds. The memory for a non-uniform interval LUT is dramatically reduced compared to that of a uniform interval LUT.

| | 1ms minimum interval | 5ms minimum interval | 10ms minimum interval | 50ms minimum interval |
|---|---|---|---|---|
| 1ms | 0ms—10ms | - | - | - |
| 2ms | 10ms—20ms | - | - | - |
| 5ms | 20ms—30ms | 0ms—30ms | - | - |
| 10ms | 30ms—50ms | 30ms—50ms | 0ms—50ms | - |
| 20ms | 50ms—110ms | 50ms—110ms | 50ms—110ms | - |
| 50ms | 110ms—260ms | 110ms—260ms | 110ms—260ms | 110ms—260ms |
| 100ms | 260ms—460ms | 260ms—460ms | 260ms—460ms | 260ms—460ms |
| 200ms | 460ms—860ms | 460ms—860ms | 460ms—860ms | 460ms—860ms |
| 500ms | 860ms—1360ms | 860ms—1360ms | 860ms—1360ms | 860ms—1360ms |
| 1000ms | 1360ms—4360ms | 1360ms—4360ms | 1360ms—4360ms | 1360ms—4360ms |
| 2000ms | 4360ms—Steady | 4360ms—Steady | 4360ms—Steady | 4360ms—Steady |
| Memory Utilisation (Uniform Interval) | 2.7KBytes (450KBytes) | 2.1KBytes (90KBytes) | 1.7KBytes (45KBytes) | 1.1Kbytes (9KBytes) |
| Memory Reduction | 99.4% | 97.7% | 96.3% | 87.8% |

Table 3.3: Non-uniform interval applied for different resolutions of LUTs

We use both (a) rounding to the nearest row and (b) linear interpolation to estimate the transient temperature at any given time instant. The Figure 3.9 shows the errors in our LUT-based approach, compared with HotSpot, over a 10 seconds period. The maximum error, average error and steady state (SS) error for the two estimation approaches used in the LUT-based method are listed in Table 3.4. The errors are

calculated by comparing the continuous thermal profile generated by HotSpot and the estimated values (by rounding or interpolating) between the two consecutive rows of the LUT.

| | **Rounding to nearest row** | | | | **Linear interpolation** | | | |
|---|---|---|---|---|---|---|---|---|
| Minimum Interval | 1ms | 5ms | 10ms | 50ms | 1ms | 5ms | 10ms | 50ms |
| Maximum Error(°C) | 0.02 | 0.12 | 0.16 | 0.19 | 0.016 | 0.053 | 0.086 | 0.142 |
| Average Error (°C) | -0.0021 | -0.0022 | -0.0023 | -0.0027 | -0.0014 | -0.0016 | -0.0016 | -0.002 |
| SS Error (°C) | 0.00001 | 0.00001 | 0.00001 | 0.00001 | 0.00001 | 0.00001 | 0.00001 | 0.00001 |

Table 3.4: Errors in the LUT due to different LUT row resolutions

Firstly, examining Table 3.4 shows that the steady state error between the LUT-based method and Hotspot is relatively low ($10^{-5}$°C). This is not unexpected, as there should be no real difference as both Hotspot and the LUT-based method solve an identical set of equations. We can also see from Figure 3.9 and Table 3.4 that linear interpolation has a higher accuracy compared to the rounding approach, but the average errors induced by both methods are similar and small enough to be ignored. As such, while it is possible to use either rounding or linear interpolation, we adopt the rounding approach for further use as it is simple and has an acceptable average error. Additionally, the average errors do not increase much for the different resolutions. This is because the maximum error occurs in the first iteration, the larger the interval the larger the error. As we are using an event based approach, with a time horizon of one OS timer tick, it is unnecessary to consider very small increments at the beginning of task execution. As a result, the 10ms-interval LUT is more suitable for practical use after balancing between the accuracy and required memory space. If needed, we could minimize the maximum error of the 10ms-interval LUT by adding additional rows in the 0ms to 10ms interval where the maximum error occurs.

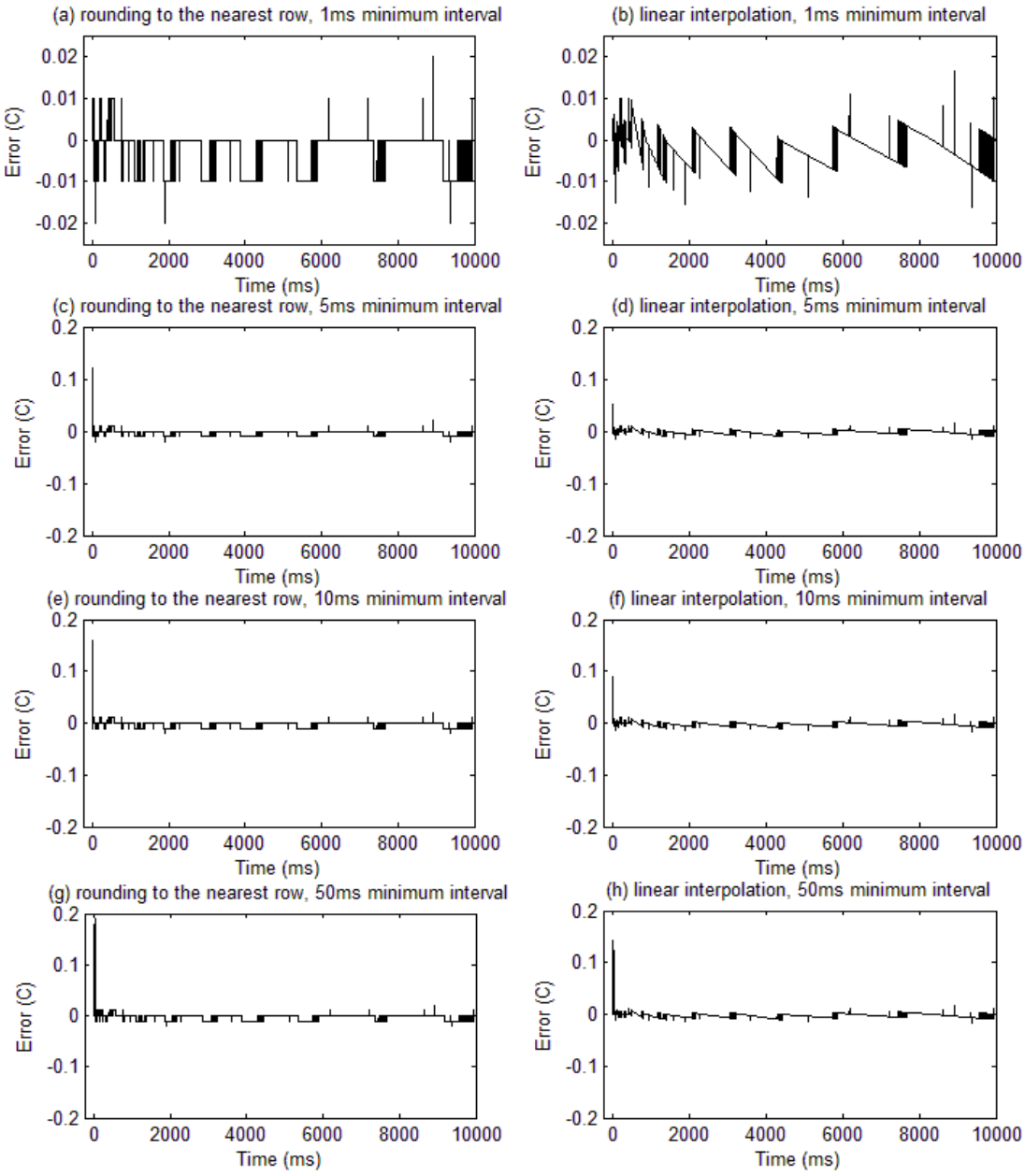



Figure 3.9: Errors due to rounding to nearest row, and linear interpolation vs. time for different resolutions

## 3.9 Summary

In this chapter, a novel event-driven thermal estimation method was proposed to update the temperature only when a power event occurs. Firstly, we show how to capture the power events from a power profile, and then how a number of LUTs, representing the temperature increment of a core for a unit power applied to that core, are prebuilt offline. After obtaining the LUT, the TDL concept and a non-uniform time

interval are applied to reduce the memory requirements. A transformation function is introduced to map the thermal response according to the actual location of the power input. Lastly, we show that the thermal response recorded in the LUTs satisfies the superposition principle, and as such can be used to accumulate all the individual thermal responses induced by the power events.

We showed by experimentation that using an atomic power profile only resulted in a small temperature error, less than 0.1°C. We also examined the accuracy between our LUT-based thermal estimator and the temperature profile from HotSpot. These results showed that the 10ms interval LUT provided an appropriate balance between temperature errors (less than 0.16°C) and storage requirements (1.7KBytes). This leads us to conclude that our proposed LUT-based thermal estimator, using a power profile converted into atomic power events, is both accurate enough and simple enough to be considered for use in high level TAS. We shall examine this hypothesis in subsequent chapters.

# Chapter 4

# Schedulability with Thermal Constraints in a Static Thermal-Aware Scheduling Scenario

In Chapter 2, we outlined the lack of a suitable schedulability analysis for multiprocessor systems which includes a fixed thermal threshold. In this chapter, we attempt to address this problem by considering the temperature threshold (junction temperature) as a strict hard scheduling constraint. This is particularly important in safety-critical embedded systems where accelerated aging and other thermal related stresses can impact on system reliability [103]. With the hard temperature constraint, performing an analysis and verification to determine if a set of real-time tasks running on a multiprocessor can meet the different timing constraint requirements in a mission-critical system becomes even more challenging. This chapter focuses on the impact of thermal constraints on real-time task sets running on a multi-core platform. The following points, summarized from the existing research literature, provide the motivation for our research in STAS.

- STAS on multiprocessors has been extensively studied [59][93][94][90][104] [95][105][93] and is important for analysing real-time systems as well as for high-level optimization. In terms of real-time task sets, this work can be classified as: 1) selecting the voltage and frequency to minimize energy consumption or temperature [59][94][93]; 2) maximizing or improving the performance under a given set of power/thermal constraints[104][105], and; 3) finding an optimal schedule to achieve the required thermal performance bounds [90][95]. Much of this work uses similar optimization techniques, such as linear programming and convex optimization, where the transient temperature cannot be applied as the optimal goal, or constraint, due to the non-trivial thermal estimation induced by the thermal capacitance and thermal coupling amongst cores.
- Power and thermal issues in multiprocessor systems create much more non-determinacy at the system design level than before, due to the additional complexity of a parallel system. Temperature is an important primary design

constraint which cannot be ignored if system safety and reliability is considered [27]. It should be regarded as another hard deadline. But most of the research literature ignores hard thermal deadlines, because of the extra complexity this imposes. For instance, [95] assumes that the core temperature can exceed the threshold, and only attempts to minimize the time the temperature is above the threshold. To the best of our knowledge, hard real-time multiprocessor schedulability analysis under predefined thermal constraints has not been fully investigated.

- Several STAS techniques use a thermal simulator (e.g. HotSpot) to obtain the accurate thermal profile used to guide the scheduling. HotSpot uses a thermal RC network to describe the multiprocessor system, and is able to output an accurate transient temperature profile for a given power input, but at the expense of an overhead many orders of magnitude greater than the OS scheduling interval. This is further complicated as the scheduling process to determine a valid schedule is to some extent iterative, and when a scheduling iteration fails, another schedule needs be generated and re-simulated against the thermal constraints. This iterative process between the schedule solver and the thermal simulator is repeated until a schedule is found or all possible solutions are tested, is extremely time consuming, and is unsuitable for task sets with a large number of tasks.

In this chapter, we propose a framework for schedulability analysis for hard real-time systems under strict thermal constraints. An algorithm for performance and thermal optimization which integrates the thermal profile generation into the schedule search is developed. This algorithm eliminates the schedule-thermal iterations associated with classical thermal simulation. This significantly reduces the run time, resulting in an efficient framework for high level static TAS or thermal management.

## 4.1 Preliminary

To overcome the computational efficiency/accuracy problem, we have introduced a fast event-driven LUT approach in Chapter 3. In this approach, we only consider the dynamic power and the non-temperature dependent leakage power (that is, leakage is

a constant), rather than the full leakage power model that will be described in the next chapter. This simplification, similar to that used in [70][71][58][59][93][94], is appropriate, as the processors used in many high performance embedded systems consume less power and operate at lower temperatures than current high performance processors [27]. The next sections develop this methodology.

### 4.1.1 The Task Model

A set of tasks with real-time and thermal constraints can be described by a task graph, as shown in Figure 4.1, where each node denotes a task. The task graph is a directed acyclic graph which explicitly confines the dependence or priority of the task in the task set: e.g. a task can only be released for execution after completion of all its parents. Only released tasks are able to be allocated to a core for execution. In a real-time scenario, a task has several inherent properties, which are listed in Table 4.1. These include: the worst case execution time (WCET) $C_i$ , the relative deadline $D_i$ (relative to the release time $a_i$, which is defined as the latest absolute completion time of all parents) and the power consumption[14] of a task $P_i$. Several indirect properties of a task, which are unknown before runtime, are also listed here. These include: the release time $a_i$, the absolute deadline $d_i = a_i + D_i$ (if $D_i$ is known in advance), the task slack time $S_i = (d_i - a_i) - C_i = D_i - C_i$, while still achieving the deadline during $a_i$ and $d_i$, and the latest absolute start time $s_i = a_i + S_i$ for the execution for a task. Therefore, a task can only be scheduled for execution (at start time $e_i$) within the interval $[a_i, s_i]$, which is defined as the problem window. Figure 4.2 shows the relationship between the various task parameters over time.

| $\tau_i$ | $C_i$ | $D_i$: $d_i$ | $P_i$ |
|---|---|---|---|
| $\tau_1$ | 20 | 40:40 | 45 |
| $\tau_2$ | 25 | 40:75 | 50 |
| $\tau_3$ | 15 | 30:60 | 55 |
| $\tau_4$ | 10 | 20:95 | 35 |

Table 4.1: The task properties

[14] In the simplest case, a task has a single (average) power consumption over its execution period. The task power consists of both the dynamic power and the component of the static power not affected by the core temperature. We will later extend this to the more complicated scenario where a task has several power values over its execution period.

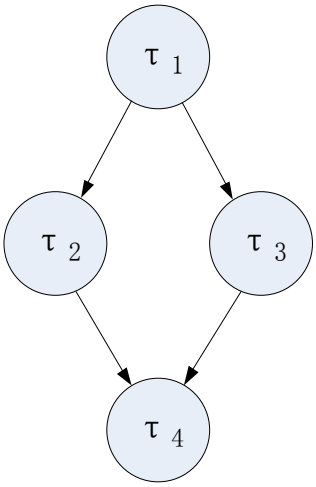



Figure 4.1: A task graph

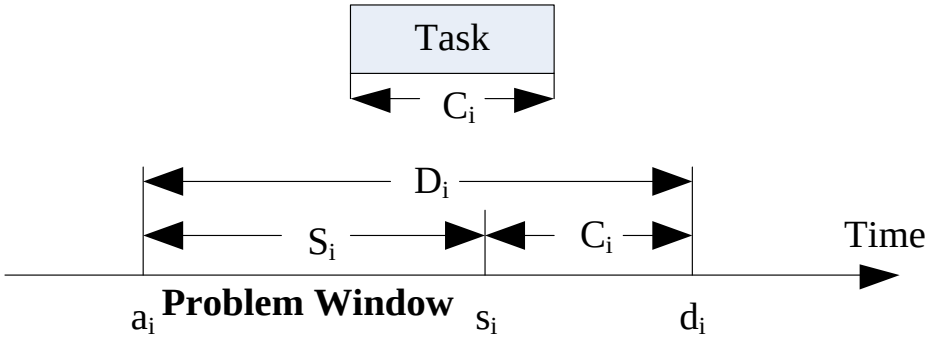



Figure 4.2: The problem window

### 4.1.2 Power Event and LUT

In Chapter 3, we showed that a continuous power profile can be simplified to a sequence of constant power events with little effect on thermal accuracy. We adopt the same power event simplification technique to analyse the schedulability. And we also use the predefined threshold, $P_{AE}$, to generate power events at the desired granularity in this schedulability test framework. This procedure, to convert a continuous task's power profile into a sequence of power events, has been described in Chapter 3, and as such will not be considered further in this chapter.

Initially, but without loss of generality, we assume that a task corresponds to just two power events, a single (increasing) power event and a single (decreasing) power event. That is, between two power events, there is only a single power value (average power between two events), similar to that shown in Table 4.1. In Section 4.4, we extend our framework to include tasks with multiple power event values, as well as power management events such as DVFS.

In Chapter 3, we also showed that the temperature can then be determined by accumulating all the temperature increments and decrements induced by individual power events, by using an LUT-based approach. This means that a transient temperature profile can be quickly obtained by adding all of the individual thermal responses. We also described two ways to use the prebuilt LUTs: one with a smaller overhead which only involves subtracting two rows, and the other with a slightly larger overhead, that accumulates all the LUTs (i.e. accumulates the thermal responses) to get the full thermal map with all the individual temperature transients. In the STAS scenario described here, we use the latter LUT-based method to perform a fast schedulability test, with strict thermal constraints, for STAS in real-time embedded systems. In next section, the formalization of the LUT accumulation operation is detailed.

## 4.2 LUT-Based Operations

In this section, several basic and simplified operations are introduced and formalised for one task and multiple task accumulations.

### 4.2.1 Addition of Two LUTs

In a multiprocessor scenario, if multiple power events occur both temporally and spatially, the temperature transient can be obtained by adding the corresponding thermal responses at each core, namely by adding the transformed LUT at the desired time points. Given two power events at times $t_i$ and $t_j$, and locations $loc_i$ and $loc_j$, the thermal map $\Delta T$ can be expressed as:

$$\Delta T = LUT^{t_i}_{[P_i,loc_i]} + LUT^{t_j}_{[P_j,loc_j]} \quad (4.1)$$

where $LUT^{t_i}_{[P_i,loc_i]}$ is a modified LUT which is obtained by transforming the multiprocessor's single LUT, based on the TDL of $loc_i$ multiplied by the power $P_i$. The time $t_i$ denotes that the first row of the modified LUT which should be aligned to the absolute time point $t_i$, and the addition operation '+' defines the cell-by-cell addition after the two LUTs are temporally aligned. Figure 4.3 intuitively shows the addition of two LUTs. The important point is that the result is still a table, but now with length $Steady + (t_j - t_i)$. When a non-uniform interval LUT is used, it should be extended into the uniform interval LUT before the addition is applied. These extension operations can be done off-line for each task.

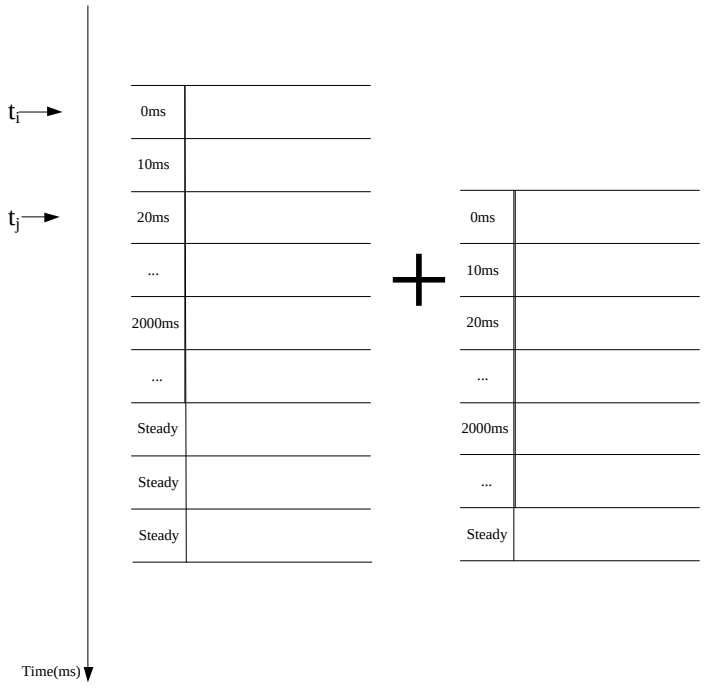



Figure 4.3: The addition of two LUTs

### 4.2.2 Simplified LUT Addition for a Single Task

In Section 4.1.2, we described how a task with a single constant power value can be interpreted as two sequential power events: a power increase $P_i$ at the beginning of task execution $e_i$ and a power decrease $-P_i$ at the end of task execution $e_i + C_i$. Thus,

it is relatively easy to construct a thermal table $Tab^{e_i}_{[C_i,P_i,loc_i]}$ for one task, based on the existing LUT approach:

$$Tab^{e_i}_{[C_i,P_i,loc_i]} = LUT^{e_i}_{[P_i,loc_i]} + LUT^{e_i+C_i}_{[-P_i,loc_i]} = LUT^{e_i}_{[P_i,loc_i]} - LUT^{e_i+C_i}_{[P_i,loc_i]} \quad (4.2)$$

If $C_i \geq Steady$, that is, during the heating stage the core has achieved a steady state temperature for the given power input, then the heating and cooling stages in the temperature profile will be symmetrical, and can be represented by a table of length $2C_i$, as shown in Figure 4.4(a). This is because the heating and cooling response is symmetrical in the thermal RC model. However, as the thermal time constant is typically around 5s to 10s [35], it is more likely that a task's WCET $C_i$ will be shorter than $Steady$, and as such, the heating stage does not reach steady state before the task completes. Therefore, for a heating stage (which does not reach steady state) followed by a cooling stage, at any time instance in the cooling stage, the LUT value $LUT^{e_i+C_i}_{[P_i,loc_i]}$ will always be smaller than the LUT value $LUT^{e_i}_{[P_i,loc_i]}$ in the heating stage. As a result, the heating stage and the cooling stage in the thermal profile for a single task, as shown in Figure 4.4(b) and (c), is asymmetrical. Figure 4.4 shows the thermal profile (based on Equation 4.2) for a single core in a simulated ARM-based multi-core architecture with a single 5Watt task executing on Core(0,0) (i.e. $P_i$ = 5Watt and $loc_i$ = Core(0,0)) for (a) 10s, (b) 1s, and (c) 100ms, respectively. In this example, the time required for the core to reach a steady state temperature is 9.5s. The bottom two plots clearly show that the cooling stage needs a much longer time, relative to the heating stage, for the temperature to fall back to the initial temperature. The total time required for the addition of the two LUTs (as described by Equation 4.2) is $Steady + (e_i + C_i - e_i)$ and thus $Tab^{e_i}_{[C_i,P_i,loc_i]}$ has length $Steady + C_i$ rather than $2C_i$. As a result, the table for a single small duration task, $Tab^{e_i}_{[C_i,P_i,loc_i]}$, would require a large amount of memory due to the large $Steady$ value.

To reduce this memory requirement, it may be possible to ignore the thermal effect for the period after $(2 + n) \cdot C_i$, where n is an integer in the range $0 \leq n \leq Steady/C_i$. That is, we simply truncate the table to length $(2 + n) \cdot C_i$, and just assume that the temperature drops back to the initial temperature. Although this assumption could result in an unacceptable error, particularly in very high power processor architectures

with $n$ close to zero, it is less severe in the low power architectures typically used in embedded systems, due to the lower working temperature [27]. Users should choose an appropriate value for n to meet their specific accuracy requirements. A larger n will more accurately follow the actual thermal profile, but will need more memory space. A smaller n results in a lower overhead for the LUT addition operations and thus can increase the speed of the schedulability test. Table 4.2 (and Figure 4.4) show the errors for different n for a single 5Watt task on Core(0,0).

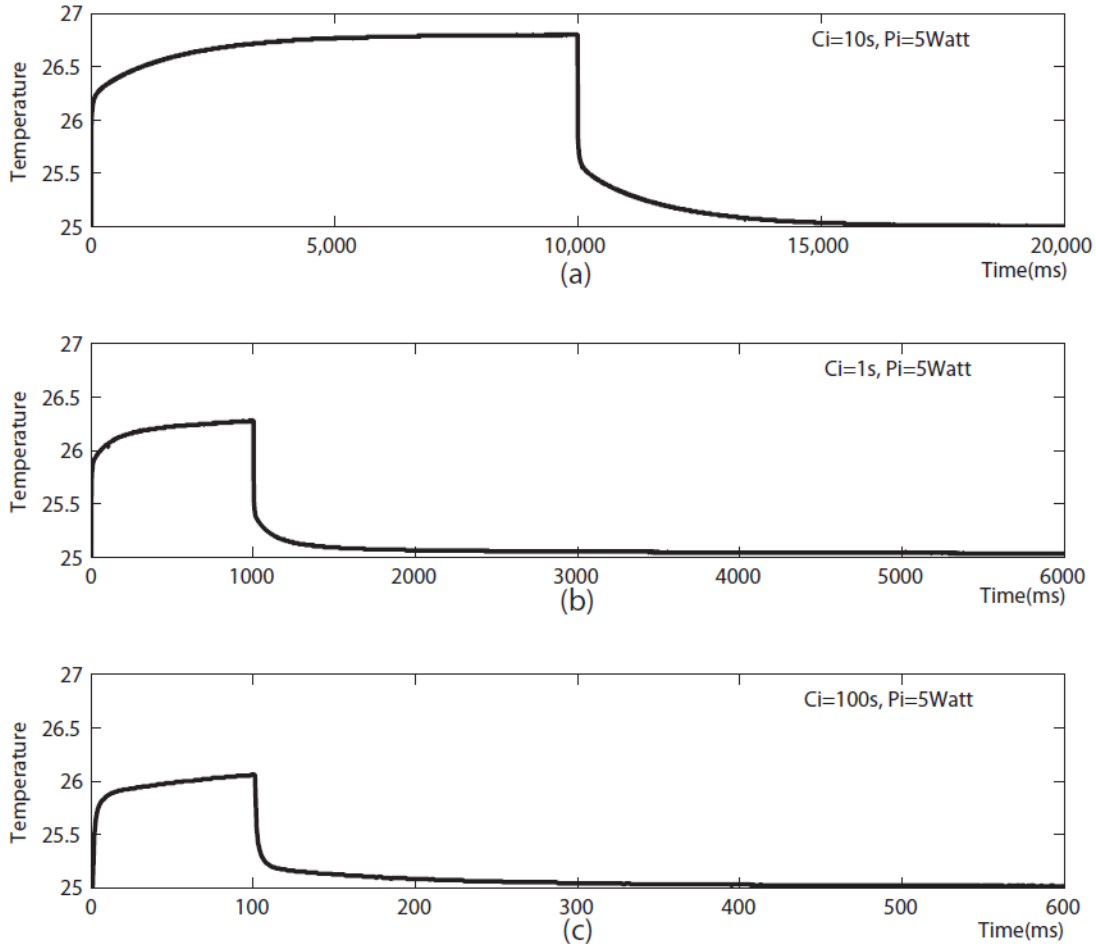



Figure 4.4: The core thermal profile for different $\boldsymbol{C_i}$. (a) The core temperature has reached steady state, (b) and (c) The core temperature has not reached steady state at the end of task execution.

| $\boldsymbol{n}$ | **0** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|---|
| $C_i$=100ms | 0.08 | 0.04 | 0.03 | 0.02 | 0.01 |
| $C_i$=1s | 0.06 | 0.05 | 0.04 | 0.04 | 0.03 |

(Note: $n$= 0 is equal to a length of $\boldsymbol{C_i}$, $n$= 2 is equal to a length of $\boldsymbol{3C_i}$)

Table 4.2: The error (in °C) caused by different table truncation lengths

To demonstrate the effect of different truncated table lengths on the schedulability test runtime and the accumulated error, we use the 15 task and 30 task synthetic task sets, described in Section 4.5.2. These tasks have a worst case execution time $C_i$ which is randomly generated in the range [2ms:120ms]. The absolute deadline $d_i$ is obtained from the randomly generated task slack time $S_i$ which is in the range [8ms:55ms], and the power consumption of the tasks $P_i$ is in range [1.5W:6W].

| $n$ | 0 | | 1 | | 2 | | 3 | | 4 | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Task Set** | **15task** | **30task** | **15task** | **30task** | **15task** | **30task** | **15task** | **30task** | **15task** | **30task** |
| Runtime (s) | 1.2 | 45 | 1.7 | 69 | 2.0 | 89 | 2.5 | 118 | 3.1 | 141 |
| Accu. Error (°C) | 0.62 | 0.94 | 0.36 | 0.51 | 0.21 | 0.30 | 0.14 | 0.19 | 0.06 | 0.12 |

Table 4.3: Accumulated error and runtime for different length of truncated table

As seen from above table, $n = 2$ provides a good balance between accuracy and the speed of calculation. As the focus of this work is TAS for lower power embedded systems, we will assume a truncated table length of $3C_i$, $(n = 2)$ unless otherwise stated. This simplified (truncated) thermal table ($TT$) for one task is denoted by $TT_{[C_i,P_i,loc_i]}$.

### 4.2.3 Addition of Task Tables

As stated in Section 4.2.1, the temperatures induced by different power events (determined from the LUT) can be added together to give the complete thermal map for the multiprocessor. In a similar way, the task table can also be used to accumulate the temperatures induced by different tasks. If we define the task execution start point as $e_i$ , then the temperature change resulting from the execution of two tasks $\tau_i$ and $\tau_j$ can be expressed as:

$$\Delta T = TT^{e_i}_{[C_i,P_i,loc_i]} + TT^{e_j}_{[C_j,P_j,loc_j]} \tag{5.3}$$

If $e_j \geq e_i$, the resultant table length is $(2+n) \cdot C_i + (e_j - e_i)$. The overall thermal map for all tasks in a given task set can be determined by accumulating the individual tables, as:

$$ThermalMap = T_{init} + \sum_{i} TT^{e_i}_{[C_i,P_i,loc_i]} \quad (5.4)$$

where $T_{init}$ is the initial thermal map temperature. Using the thermal map $ThermalMap$ allows us to quickly and easily determine when and where core overheating occurs (a cell value in $ThermalMap$ is larger than some predefined threshold) in the static TAS. Thus, it gives a very quick technique for determining if a particular static TAS is successful or not.

While the LUT based thermal map technique proposed above is fast compared to Hotspot (as shown in Chapter 3), its memory requirement, particularly for large task sets, may become excessive as each task requires its own table for each possible core location. To reduce this memory requirement, we make use of a transformation function $trans()$ to convert the table for a particular TDL to a table corresponding to an individual core, as in Chapter 3.6. That way, each task only requires a table corresponding to each TDL (for example, a 4×4 multiprocessor only needs 3 tables for each task). The task TDL tables are calculated prior to the execution of the scheduling algorithm. Calculating the task tables dynamically for an individual core (from the static task TDL tables) does increase the computational complexity slightly, but, as $trans()$ is just a column mapping, it is not overly time consuming.

## 4.3 Static Thermal Aware Scheduling Algorithm

A given task should start execution at any time point during the problem window defined by Figure 4.2 in Section 4.1.1. If a task is unable to be scheduled within the problem window, it must be rejected and the schedule fails. The process of searching for the appropriate start point for task execution can be considered to be equivalent to that of scheduling the task. The task's execution start time $e_i$ can be scheduled anywhere within its problem window, and accordingly, the task's *TT* could be moved to any point in the problem window along the time axis. As such, our algorithm should consider the following objective:

- Schedulability: That is, determine the start time for task execution, for all tasks in the given task set under the predefined temperature threshold constraint. If

any task is unable to be scheduled within its problem window, the schedule is considered a failure.

Assuming that the schedulability criterion is satisfied, one of two additional objectives can be considered:

- Maximizing performance: Assuming the schedulability criterion is satisfied, a task in a given task set should start as early as possible so that the whole task set can complete execution in the shortest possible time.
- Minimizing peak temperature: Assuming the schedulability criterion is satisfied, and then a task in a given task set should be scheduled to minimize the peak temperature.

The temperature increment for a task is described by the *TT*, and thus, the static TAS problem can be considered as simply moving the task tables within their problem windows and accumulating the tables.

### 4.3.1 Schedulability and Performance Maximization

To make the static TAS problem more efficient in finding an appropriate schedule for a given task set, we can use different search strategies according to the different optimization objectives. The first two objectives can be achieved by using an identical search technique: For each task, we move its *TT* forward in the problem window, as shown in Figure 4.5(a).

Before executing the static TAS algorithm, we define the thermal threshold for each core, as $T_{jmax}$, and initialize the $ThermalMap$ using the initial temperature $T_{init}$ of each core. We also pre-construct the static TDL-based thermal table for each task before running the scheduling algorithm.

Because the release time $a_i$ of each task depends on its parents' completion time, we use a recursive algorithm, given in Algorithm 4.1, to search all the task problem windows to test for schedulability, and thus obtain the optimal solution for performance maximization. If schedulability validation is successful for a given task set, then the first solution of $e_i$ is the one that will give the highest performance, as

searching forward in each task's problem window will result in the fastest possible execution of the task set. Hence the algorithm terminates after the first solution is found. As the forward search algorithm gives the highest performance, we refer to Algorithm 4.1, later in the text, as the high performance (HP) algorithm.

**Algorithm 4.1: Forward search algorithm for maximizing performance**

**Input**: $i, a_i, s_i$
**Output**: $e_i$ or unsuccessful

**FUNCTION** Solve($i, a_i, s_i$)
  **IF** $a_i \geq s_i$ **THEN**
    Print("Schedulability test is failed!");
    Algorithm terminates;
  **END IF**
  **IF** $\tau_i$*'s parents are not visited or not null* **THEN**
    Choose another start node *j* in topological sorting; //start nodes means its precedencies have all been visited in Directed-Acyclic Graph (DAG)
    Solve($j, a_j, s_j$);
  **END IF**
  **FOR** $e_i = a_i$ *to* $s_i$ **DO**
    **FOR** *each child* $\tau_j$*of task* $\tau_i$ **DO**
      Update $\tau_j$'s problem window $[a_j = MAX(e_i + C_i, a_j), s_j = a_j + S_j]$; //problem window has nothing to do with the scheduling location
    **END FOR**
    //Select either $alloc\_policy()$; //exhaustive search is optional
    $\{loc_i\} = alloc\_policy(\tau_i, ThermalMap, IdleMap)$;
    //$\{loc_i\} = 1$ to *NUMCORE* and idle in time slot $[e_i,\ e_i + C_i]$; //This is optional for exhaustive search
    **FOR** *each core in* $\{loc_i\}$ **DO**
      $ThermalMap = ThermalMap + TT^{e_i}_{[C_i,P_i,loc_i]}$;
      **IF** *any cell in* $ThermalMap \geq T_{jmax}$ **THEN**
        $ThermalMap = ThermalMap - TT^{e_i}_{[C_i,P_i,loc_i]}$; // *ThermalMap* is a global variable
        Print("overheat at time and core!");
        **RETURN**; //returning to other recursive level to continue searching
      **ELSE**
        Record $loc_i$ and $e_i$;
        **IF** $\tau_i$ *is the last task in task set* **THEN**
          Print("Schedulability test is passed!");
          Algorithm terminates;
        **END IF**
        **IF** *using* $alloc\_policy()$ **THEN**
          $\Gamma = sibling\_reorder(\tau_i's\ all\ child, EDF)$;
        **ELSE**
          $\Gamma$ is in original order;
        **END IF**
        **FOR** *each* $\tau_i$*'s child* $\tau_j$ *according to order* $\Gamma$ DO
          Solve($j, a_j, s_j$);
        **END FOR**
      **END IF**
      $ThermalMap = ThermalMap - TT^{e_i}_{[C_i,P_i,loc_i]}$;
    **END FOR**
  **END FOR**
  Print("Schedulability test is failed!");
**END FUNCTION**

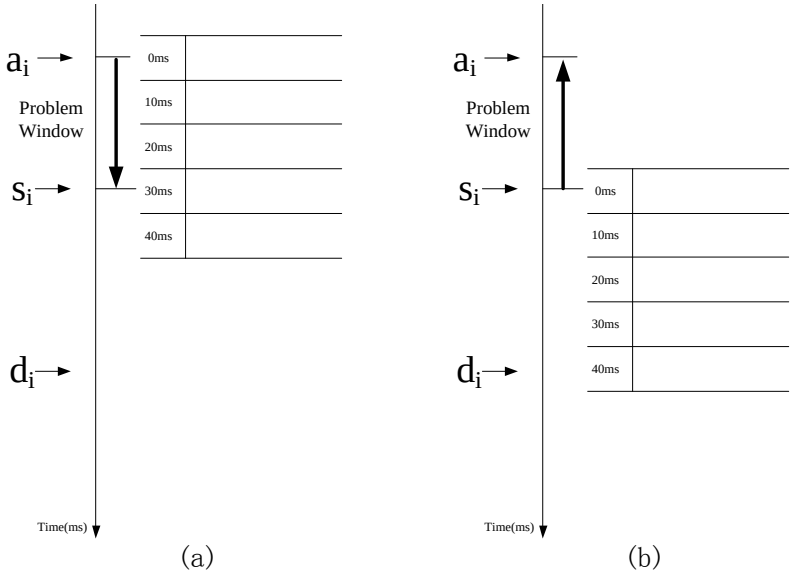



Figure 4.5: Searching (a) Forward in time, and (b) Backward in time

In Algorithm 4.1, the function $alloc\_policy()$ is a user defined allocation policy that determines which core a task is allocated to, and is independent of the scheduling algorithm. As a result, Algorithm 4.1 only determines the schedulability of a task set with a specific allocation policy. For example, a non-thermal-aware scheduling algorithm (e.g. random [92] or minimum core ID [92]) or a heuristic TAS algorithm (e.g. coolest-first [92] or neighbour-aware [92]) based on the $ThermalMap$ at release time $a_i$, could be used to allocate a task. The thermal simulation is integrated into the scheduling algorithm. If an allocation policy is not specified in advance, the schedulability validation must be extended to cover every possible idle core at every time point in the problem window. This is reflected in Algorithm 4.1 by the need for the user to select from either a specific allocation policy or the exhaustive search statement, just below the statement "select either $alloc\_policy()$ or exhaustive search from below". The exhaustive search will find the optimum schedule (if one exists) and can be used to analyse the performance of a particular heuristic scheduling algorithm.

The function $sibling\_reorder()$ determines the schedule order among a task $\tau_i$'s children, as the children are all available for scheduling at the same time point. This allows well known scheduling policies, such as: first come first served (FCFS), earliest deadline first (EDF), shortest problem window (SPW), lowest/highest power first or local priority, etc., to be used. It should be noted that $sibling\_reorder()$ only affects the performance maximization solution, and is irrelevant in terms of the

schedulability test as all possible locations and starting time points for each task are considered in the search scope. That is, $sibling\_reorder()$ is only used when a specific allocation policy determined by $alloc\_policy()$ is invoked, as all possible sibling orders[15] are implicitly included when using the exhaustive search routine.

If our objective is just testing for schedulability, without considering performance, then searching backward (that is moving the table backward along the time axis), as shown in Figure 4.5(b) may be more efficient for schedulability validation. Searching forward will more likely lead to an overheating condition (that is the temperature threshold is exceeded), since searching forward results in a shorter slack time between tasks and allows less time for the core temperature to decrease. Searching backward, on the other hand, allows us to easily check that at least one schedule is possible for the task set. This backward search is similar to Algorithm 4.1, except that the start time point $e_i$ is moved from $s_i$ towards $a_i$.

However, if our purpose is to find the optimal solution with respect to minimizing the peak temperature, rather than just to test schedulability, then we need to use an exhaustive search strategy to cover all available schedules, not just the first successful solution. This is because searching backwards cannot promise that the first solution is the global optimal one (as explained in next section). To achieve this, the termination statements in Algorithm 4.1 should be removed.

### 4.3.2 Heuristic Peak Temperature Minimization

As described above, it is possible to use Algorithm 4.1 with a forward search to maximize performance, by making the idle interval (the gap) between the tasks as short as possible under the thermal constraints. Alternatively, by using a backward search it is possible to make the idle interval between children and their parents as long as possible to allow the most time for the chip to cool down. However, this is not always true as siblings could be executed in parallel and their tables could overlap on the time axis. This would result in both a temporal and spatial contribution to the

[15] If one task has *NUMCHILD* children, then there are *NUMCHILD*! possible sibling orders among these children. So if global maximum performance is required, all possible sequences should be traversed.

temperature increment. Intuitively, minimizing the peak temperature can be achieved by distributing the sibling tasks along the time axis, whereas a backward search will likely cluster the sibling tasks close to their deadlines, which could affect the temperature profile of subsequent tasks. Figure 4.6 shows the rationale for distributing tasks during a backward search. Figure 4.6(a) shows a likely first solution for a backward search, where two sibling tasks are scheduled close to their latest start time. This would result in a temperature accumulation due to the two tasks occurring at similar times. Figure 4.6(b) solves this by spreading the sibling tasks on the time axis.

Extending this to the general case for N sibling tasks $\tau_{i=[1..N]}$, we consider the sibling with the longest problem window and evenly split this into $N$ segments, where the $i^{th}$ time point is denoted by $\varphi_i$. That is, $\varphi_0$ corresponds to the siblings' release time and $\varphi_N$ corresponds to the endpoint of the longest problem window. We then reduce each sibling's problem window $[a_i, s_i]$ to $[a_i, MIN(\varphi_i, s_i)]$. Figure 4.6(c) shows this when using an EDF reordering strategy, where task $\tau_2$'s problem window is shortened to $[a_2, \varphi_1]$. However, EDF cannot necessarily promise a good solution for peak temperature minimization. Instead, we introduce a metric, the weighted peak distance (WPD), to measure the effectiveness of a sibling task order. Given the distance $\xi_{i,j}$ between the completion time points of two sibling tasks $\tau_i$ and $\tau_j$, with reduced problem windows, defined as:

$$\xi_{i,j} = \left|\left(MIN(\varphi_j, s_j) + C_j\right) - \left(MIN(\varphi_i, s_i) + C_i\right)\right| \tag{4.5}$$

WPD can then be defined as:

$$WPD = \sum_{i=1}^{N} \sum_{j=i+1}^{N} (P_i + P_j) \cdot \xi_{i,j} \tag{4.6}$$

A bigger $WPD$ value indicates that the temperature peaks for each task would be spread further on the time axis. Thus, Figure 4.6(d) is heuristically a better solution than Figure 4.6(c), because $\xi_{2,3}$ is much smaller in Figure 4.6(c). Thus, if we can find a sibling order that maximizes $WPD$ and the schedule is valid, then we can assume that the first solution is the solution we want. The $sibling\_reorder()$ implementation using the WPD metric for minimizing the peak temperature during the backward

search is shown in Algorithm 4.2. We refer to this algorithm as the heuristic peak temperature minimization (HPTM) algorithm.

**Algorithm 4.2: Sibling reorder for backward search**

**Input**: sibling task set $\tau_{1\cdots N}$, original task order $\Gamma = \{1 \cdots N\}$
**Output**: new order $\Gamma$

**FUNCTION** reorder_sibling($\tau_{1\cdots N}$)
  **FOR** $i = 1$ *to* $N$ **DO**
    **FOR** $j = 1$ *to* $N$ **DO**
      $complete_{tab[i][j]} = MIN(\varphi_j, s_i) + C_i$;
    **END FOR**
  **END FOR**
  $MaxWPD = 0$;
  **FUNCTION** gen_all_permutation($\Gamma, a$)
    **IF** $a < N$ **THEN**
      **FOR** $k = a$ *to* $N$ **DO**
        swap($\Gamma[a], \Gamma[k]$);
        gen_all_permutation($\Gamma, a + 1$);
        swap($\Gamma[a], \Gamma[k]$);
      **END FOR**
    **ELSE**
      $WPD = 0$;
      **FOR** $i = 1$ *to* $N$ **DO**
        **FOR** $j = 1$ *to* $N$ **DO**
          $WPD = WPD + (P_i + P_j) \cdot \left| complete\_tab[i][\Gamma[i]] - complete\_tab[j][\Gamma[j]] \right|$;
        **END FOR**
      **END FOR**
      **IF** $WPD > MaxWPD$ **THEN**
        $MaxWPD = WPD$;
        Record $\Gamma$;
      **END IF**
    **END IF**
  **END FUNCTION**
  Invoke gen_all_permutation($\Gamma'$,1);
  **RETURN** $\Gamma$;
**END FUNCTION**

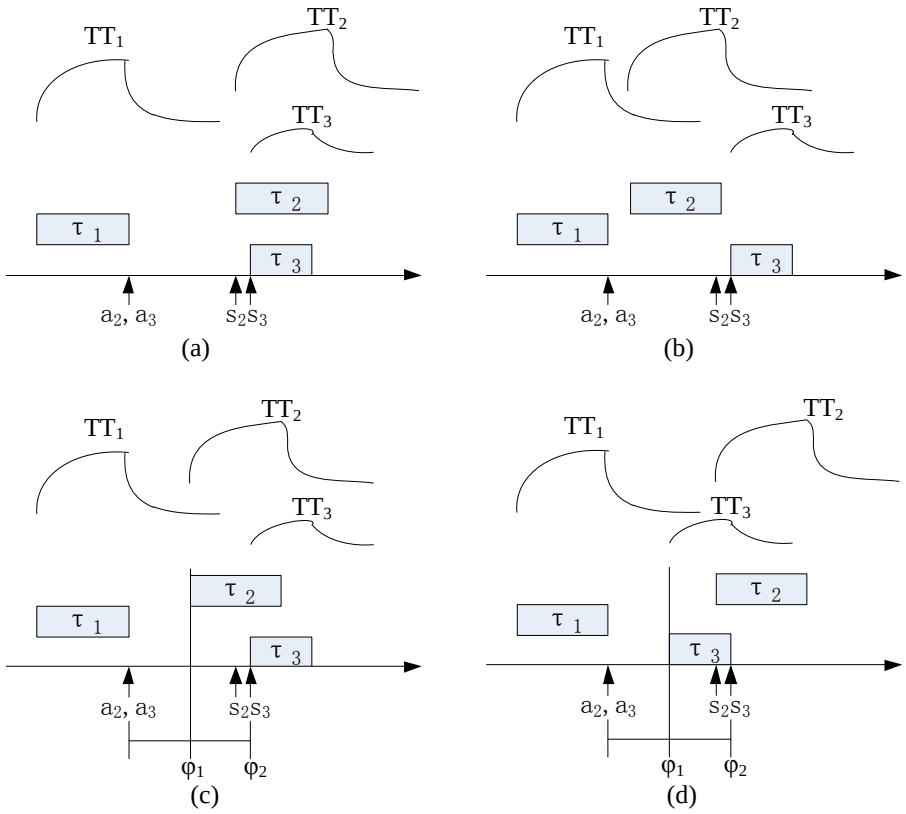



Figure 4.6: A heuristic for minimizing the temperature during a backward search

## 4.4 Problem Extension

The algorithms proposed so far give an effective framework to determine constrained schedulability in static thermal-aware scheduling. This framework shows that any power event on a core in a multiprocessor can be tested and scheduled by moving the tasks' thermal tables and adding them together. This framework can be easily extended to more complicated scenarios, so long as they can be converted to a sequence of power events. Typical scenarios include:

- A single task which has several sequential power values.
- A task which is intermittently put into a sleep state, such as when power-gating is enabled to cool down the core.
- Task execution when dynamic frequency scaling is enabled during task execution.

The above scenarios can be modified to fit into our framework by splitting the task into subtasks. That is, a larger task, with time varying properties, can be split into several subtasks with static task properties. For, example, in the first scenario, the task is split into subtask segments of constant power, each of which execute sequentially and, apart from the first subtask, all have a problem window of zero. In the second scenario, the WCET is extended and needs to be recalculated. In the last scenario, both the power value and the WCET need to be re-calculated and assigned to each subtask since frequency scaling can linearly affect both power and execution time. Figure 4.7 shows the task graph of Figure 4.1, but with task $\tau_3$ split into 3 subtasks. As such, these more complicated scenarios can be converted into a sequence of simple tasks which can then be scheduled using Algorithms 4.1 and 4.2.

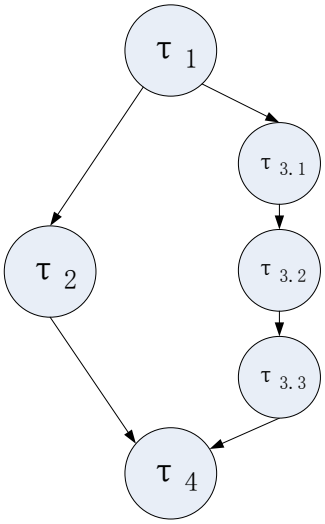



Figure 4.7: Task splitting for Figure 4.1

### 4.4.1 Using Slack to Cool Down the Chip

If a valid schedule exists, under the predefined thermal constraint, we could take advantage of power-gating and the available slack (that is, the interval between the task completion point and the deadline) to intermittently put the task into a sleep mode and further reduce the heat dissipation of a core. In fact this is identical to the second and third scenarios mentioned at the beginning of Section 4.4. In Algorithm 4.1, we can obtain the earliest start point $e_i$ for each task $\tau_i$ by using the forward search algorithm.

In terms of power-gating, we predefine a cool-down interval (the duration of the sleep state) as $t_{sl}$. Hence, the following condition must be satisfied:

$$NSL_i \cdot t_{sl} \leq s_i - e_i \tag{4.7}$$

where $NSL_i$ is the number of times that the task is placed into the sleep state. Intermittent periods of sleep could help to avoid high temperatures over a short temporal scale, while still being able to meet the deadline and the thermal constraint since the slack is verified from the previous schedulability test. We refer to this scenario as stop-go (S-G) scheduling.

In a similar way, we could also extend a task's WCET (that is, slow down task execution) by using frequency scaling to fill the slack. To achieve this, we assume that a task $\tau_i$ can be split into several subtasks $\tau_{i,j}$, where each subtask corresponds to a constant frequency. If $F_i$ denotes the maximum frequency under which a task's WCET is obtained, the following condition must be satisfied:

$$\begin{cases} \sum_j t_{i,j} \leq d_i - e_i \\ \sum_j t_{i,j} \cdot f_{i,j} \leq C_i \cdot F_i \end{cases} \tag{4.8}$$

where $t_{i,j}$ denotes the subtask execution time and $f_{ij}$ denotes the subtask frequency. Equation 4.8 represents the general multiple subtask case. If task splitting is not used, then a task $\tau_i$'s frequency could be scaled as $f_i/F_i = C_i/(d_i - e_i)$. We refer to this scenario as dynamic frequency scaling (DFS) scheduling.

## 4.5 Experiment

We use both actual real-time benchmark applications and synthetic real-time task sets to validate the schedulability performance of our framework. We then show that our framework can also optimize critical metrics in static TAS, such as performance, peak temperature and so on. All experiments are simulated using an Intel Core 2 Duo E8500 with 8GB RAM operating at 3.16GHz with the Ubuntu 10.10 OS.

### 4.5.1 Validation of the Framework

In this section, a real-time embedded benchmark, PapaBench [106], used for WCET and scheduling analysis is used to test our framework. This benchmark is part of the Paparazzi project [106], a real-time application developed for an unmanned aerial vehicle, running on dual ATMEL AVR micro-controllers. PapaBench v0.4 has support for ARM architecture processors, and as such we can use a multi-core ARM simulator to profile the power consumption for each task. In this experiment, we simulate a dual core processor with a shared L2 cache, similar to the Nvidia Tegra 2. Each core has a separate L1 cache, included inside the core. The L2 cache is shared and is not considered part of the processor core in this simulation. The initial and environmental temperature is set to 25°C (for the embedded processor scenario with low power consumption) and the temperature threshold is defined as 30°C.

Figure 4.8 shows the task graph and the corresponding micro-control-units (MCUs) abstracted from the PapaBench source code. The power profile for each task in PapaBench is generated from the source code using SimpleScalar-ARM (which includes Wattch) and is shown in Table 4.4. Note that the deadline of each task is an absolute deadline, and as such the problem window $s_i = d_i - C_i$ for each task can be determined before the algorithm runs. The power value of the tasks listed in Table 4.4 only includes the processor power which is profiled by Wattch [43], and the power consumed by I/O and peripherals (e.g. the GPS module, radio module, sensors and actuators) is not taken into account. We use the same task set partition as in PapaBench to determine which core (MCU0/Core0 or MCU1/Core1) is assigned for each task. Thus, this predefined task allocation negates the need for $alloc\_policy()$ or the exhaustive search in Algorithm 4.1.

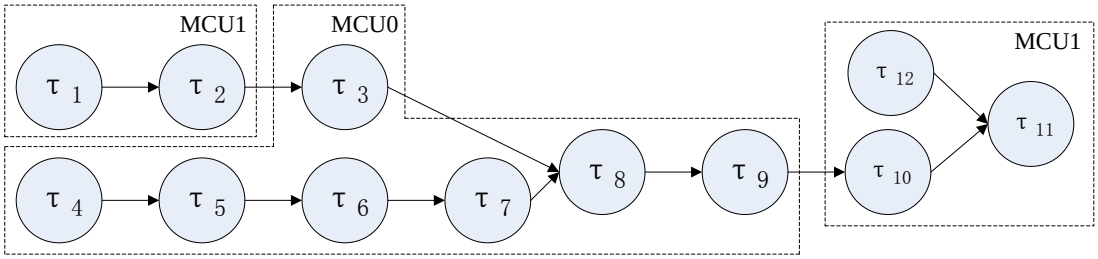



Figure 4.8: The task graph for PapaBench

| Task | Description | $C$ (ms) | $d$ (ms) | $P$ (Watt) |
|---|---|---|---|---|
| $\tau_1$ | Receive Radio Command | 4 | 25 | 3.2 |
| $\tau_2$ | Send Radio Command to MCU0 | 5 | 25 | 3.5 |
| $\tau_3$ | Receive Radio Command in MCU0 | 6 | 25 | 2.7 |
| $\tau_4$ | Receive GPS Data | 6 | 250 | 3.5 |
| $\tau_5$ | Navigation | 6 | 250 | 3.9 |
| $\tau_6$ | Altitude Control | 5 | 250 | 4.3 |
| $\tau_7$ | Pitch Control | 4 | 250 | 4.1 |
| $\tau_8$ | Stabilization | 4 | 50 | 5.1 |
| $\tau_9$ | Send Servo Data to MCU1 | 6 | 50 | 3.8 |
| $\tau_{10}$ | Receive Servo Data in MCU1 | 3 | 50 | 2.9 |
| $\tau_{11}$ | Transmit Servos | 6 | 50 | 4.9 |
| $\tau_{12}$ | Check Failsafe | 5 | 50 | 2.5 |

Table 4.4: Tasks characteristics from PapaBench after simulation and profiling

Figure 4.9 shows the resulting schedule obtained from Algorithm 4.1, which shows that the PapaBench task set is able to be successfully scheduled within the temperature constraints. As Algorithm 4.1 gives a solution which represents the fastest possible execution of the task set, the schedule shown in Figure 4.9 indicates the best performance. This schedule has a completion time of 47 milliseconds for the 30°C temperature constraint. Furthermore, the temperature profile of the two cores for the schedule shown in Figure 4.9 is given in Figure 4.10. The solid line represents the thermal profile determined using our proposed thermal-aware scheduling framework, with a table length of $3C_i$, while the dashed line represents the thermal profile determined by Hotspot for this particular schedule. The two plots in Figure 4.10 represent the temperature profile of MCU0 and MCU1 respectively. Figure 4.10 shows that there is good agreement between the two thermal profiles, and as such, we conclude that our truncated table approach is accurate when used to check the thermal constraint, while saving a large amount of processing time and memory for task table storage.

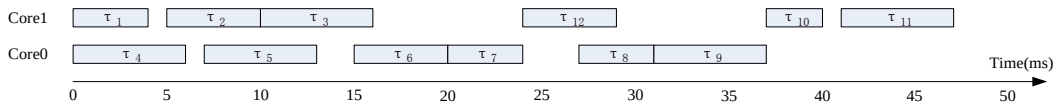



Figure 4.9: A successful thermal schedule for PapaBench

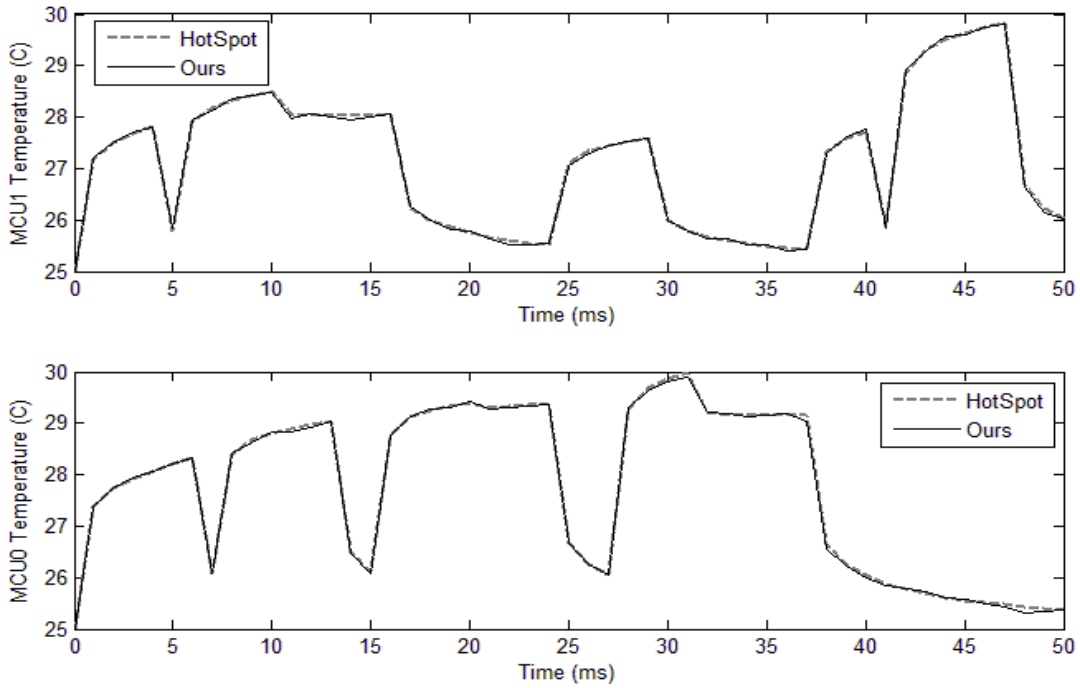



Figure 4.10: Thermal Profile Comparison of Figure 4.9

From this schedulability test, we gain some interesting insights that could be useful for improving the fast schedulability analysis even further:

- The gap between consecutive tasks usually occurs if there is an increase in task power (e.g. tasks $\tau_1$, $\tau_2$ and $\tau_4$, $\tau_5$, etc.). If there is a decrease in power between two consecutive tasks there is no gap (e.g. tasks $\tau_2$, $\tau_3$ and $\tau_6$, $\tau_7$, etc.). The gap between tasks allows the core to cool down.
- The sibling tasks usually run in the gaps in the other cores' task execution (e.g. task $\tau_{12}$) as this can avoid the overheating induced by spatial heat accumulation. However, it is still possible to overlap sibling task execution as the heat transfer between the cores is not dramatic for short task execution.

### 4.5.2 Effectiveness of the Schedulability Test

The previous sections used relatively small benchmarks to demonstrate our algorithm and to show that the framework is useful for real-time thermal aware scheduling. In this section, we use much larger task sets with tighter constraints and a larger multiprocessor architecture to demonstrate the effectiveness of our thermal aware

scheduling algorithm. Previous schedulability analysis [91] has been limited to expressing task properties and their constraints as temporal metrics whose interrelationships can be explicitly expressed by a set of analytical equations or inequalities. However, due to the non-linearity in the induced temperature, it is not possible to directly accumulate temperature into an analytical expression. The only way to accurately check the temperature and assure that it is under a specified threshold is to use a thermal simulator, such as Hotspot [35], to obtain the temperature profile of an entire schedule. That is, the scheduling solver generates a possible candidate schedule where each task must meet its deadline, and then sends its corresponding power profile to the thermal simulator (in this case: HotSpot) to check the temperature. If the schedule exceeds the thermal threshold, the schedule returns as a fail and another possible candidate schedule is generated and passed to the thermal simulator. This procedure is carried out iteratively, until a successful schedule is found or until no possible schedule exists. We refer to this method as the traditional method for thermal simulation. As the scheduling algorithm is already an NP-Hard problem [91], adding thermal constraints results in an unacceptable runtime. Thus it is important to evaluate our proposed thermal-aware scheduling framework to show that it can significantly reduce the computational time for static TAS.

To demonstrate the benefits of our scheduling algorithm (in this case, the HP algorithm), we conduct an experiment involving 5 synthetic task sets, which contain 15, 18, 20, 30 and 50 tasks, respectively. The worst case execution time $C_i$ is randomly generated in the range [2ms:120ms], the absolute deadline $d_i$ is obtained from the randomly generated task slack time $S_i$ which is in the range [8ms:55ms], and the power consumption of the tasks $P_i$ is in range [1.5W:6W]. The first task set (containing 15 tasks) has a slightly tight relative deadline for each task ($S_i$ is in the range [8ms:15ms]). As the number of tasks increases, the deadline is relaxed (with $S_i$ in the range [25ms:55ms] for the 50 task set). A task set is restricted so that it only allows tasks with at most 5 parents and 5 children.

We conduct our experiments using two different simulated low power multi-core processor layouts. The first has a 2×4 layout, with two rows of four cores, with each core having a size of 2mm×4mm. This layout is meant to be representative of current high end multi-core processors for embedded applications. The second layout consists

of a 4×4 layout. The 2×4 layout has less inter-core thermal coupling, as all cores have at least one exterior edge. Additionally, this layout, having fewer cores, will present a greater challenge to the scheduler. The 4×4 layout will have a larger inter-core thermal coupling, particularly for the 4 cores in the centre. However, the scheduling problem will be simplified as there are now more cores to schedule tasks to. In this experiment, the initial (and the environment) temperature is 25°C, while the predefined temperature threshold for any core is 35°C.

Table 4.5 shows the runtime to complete the schedulability test for the 5 task sets on the 2×4 layout. The “Schedulable?” row indicates if a task set can be successfully scheduled under the 35°C temperature constraint. A task set which is not schedulable tends to take more time than a task set (of comparable size) which can be successfully scheduled as the former must exhaust all possible schedules while the latter will terminate as soon as a schedule is successful. In this experiment using the 2×4 layout, the task sets with 18 tasks and 20 tasks are not schedulable under the thermal constraints, whereas the other task sets all are. The reason these two tasks are unschedulable is that they have very tight deadlines which (for the 8 core implementation) does not allow enough slack to allow the chip to cool down sufficiently. The processing time of the traditional method (using Hotspot as the thermal simulator, and referred to as TM in Table 4.5) is very large and does not complete in reasonable time for the largest task set. On the other hand, our proposed thermal-aware scheduling framework allows for much larger, more complex, task sets to be examined within an acceptable time period (only 11.5 minutes for the 50 tasks in our largest task set). The proposed algorithm shows a speed up of 2 to 3 orders of magnitude compared to the traditional method. Figure 4.11 shows the thermal profile for the two hottest cores (Core2 and Core6) in the 2×4 core layout for the successfully scheduled 15 task set determined using our proposed thermal-aware scheduling framework (compared to Hotspot). Again there is good agreement between the thermal profiles.

| **Runtime** | **15 Tasks** | **18 Tasks** | **20 Tasks** | **30 Tasks** | **50 Tasks** |
|---|---|---|---|---|---|
| Schedulable? | Y | N | N | Y | Y |
| Proposed (s) | 2 | 25 | 43 | 89 | 692 |
| TM (s) | 1290 | 23450 | 45741 | 58722 | -- |
| Speedup | 645x | 938x | 1064x | 660x | -- |

Table 4.5: The HP algorithm runtime comparison for the 2×4 core layout

We also conducted a similar experiment on the 4×4 layout, shown in Table 4.6. The task set with 20 tasks is able to be scheduled on the 4×4 layout as it is much easier to schedulable with a greater number of cores. The ease in finding a suitable schedule as the number of cores increases is reflected in the shorter execution times of our proposed framework compared to that of the 2×4 layout. Again, the proposed algorithm shows a speed up of almost 3 orders of magnitude compared to the traditional method.

| **Runtime** | **15Tasks** | **18Tasks** | **20 Tasks** | **30 Tasks** | **50Tasks** |
|---|---|---|---|---|---|
| Schedulable? | Y | Y | Y | Y | Y |
| Proposed (s) | 2 | 2 | 9 | 32 | 428 |
| TM (s) | 1035 | 1274 | 7928 | 31922 | -- |
| Speedup | 518x | 637x | 881x | 997x | -- |

Table 4.6: The HP algorithm runtime comparison for the 4×4 core layout

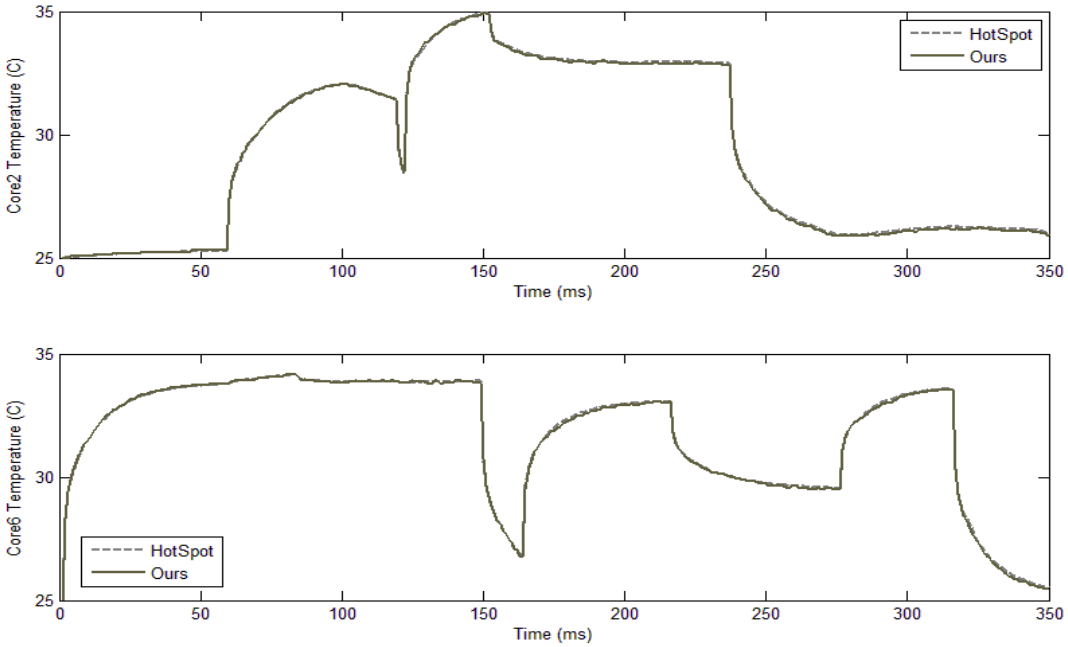



Figure 4.11: The thermal profile of two hottest cores in the 15-task example for the 2×4 core layout

### 4.5.3 Heuristic Thermal Optimization based on Static TAS

In this section, we examine the performance of the various scheduling algorithms described in Sections 4.3 and 4.4. However, before we test our proposed heuristic

algorithms, the non-TAS case is examined to show why a thermal threshold constraint and STAS are necessary. In this experiment, we use the same 2×4 multi-core processor as in Section 4.5.2, with the three successfully scheduled task sets from Section 4.5.2 (i.e. the 15 task, 30 task and 50 task sets). Again, we use an initial core temperature of 25°C, but with no thermal constraint. We examine the worst case core temperature for a random allocation policy and an earliest deadline first (EDF) policy. These are compared to the absolute worst case (that is the hottest scenario for all possible scheduling solutions) determined using an exhaustive search.

| Task Set | Random Allocation | EDF Allocation | Exhaustive Search |
|---|---|---|---|
| 15 tasks | 37.8 | 37.9 | 38.1 |
| 30 tasks | 41.1 | 41.3 | 41.6 |
| 50 tasks | 39.9 | 40.6 | 40.8 |

Table 4.7: Maximum core temperature (in °C) using non-TAS scheduling

By comparing the data in Table 4.7 with the results for the HP algorithm using a thermal constraint of 35°C (presented in Table 4.8), we can conclude that if a thermal threshold is not applied, conventional scheduling algorithms can result in high individual core temperatures, approaching that of the worst case temperature. This simple experiment shows that thermal aware scheduling may provide additional opportunities for controlling core temperature.

In the remainder of this section, we examine the performance of our proposed scheduling algorithms. These include the high performance (HP) forward search algorithm (Section 4.3.1), the backward search (BS) algorithm (Section 4.3.1), the heuristic peak temperature minimization (HPTM) algorithm (Section 4.3.2), and implementations of S-G scheduling (Section 4.4.1) and DFS scheduling (Section 4.4.1). These algorithms are compared, in terms of the peak core temperature and the algorithm execution time, to the optimal solution (determined by an exhaustive search). We use the same 2×4 multi-core processor as in Section 4.5.2, with the three successfully scheduled task sets from Section 4.5.2 (i.e. the 15 task, 30 task and 50 task sets). We also use three of the task sets from PapaBench, fir and bs from SNU benchmark [108]), but in this case with the dual core architecture described in Section 4.5.1.

The initial and environmental temperature is 25°C and the thermal threshold for the 2×4 multi-core architecture is set to 35°C, while the threshold for the dual core architecture is set to 30°C. The peak core temperature (Temp) and the algorithm execution time (Time) for the various scheduling algorithms are presented in Table 4.8.

| * | | **Opt.** | **HP** | **BS** | **HPTM** | **S-G** | **DFS** |
|---|---|---|---|---|---|---|---|
| | | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) |
| 1 | **15 Tasks** | 33.7 (36524) | 34.9 (2) | 34.5 (3) | 34.2 (7) | 34.2 (13) | 34.1 (10) |
| 2 | **30 Tasks** | 32.9 (81781) | 34.2 (89) | 33.9 (91) | 33.4 (116) | 33.4 (272) | 33.7 (161) |
| 3 | **50 Tasks** | 33.7 (188170) | 34.9 (692) | 34.7 (705) | 34.2 (991) | 33.9 (2382) | 34 (1011) |
| 4 | **PapaBen** | 28.4 (22679) | 29.1 (2) | 29.1 (2) | 28.9 (4) | 28.7 (7) | 28.8 (5) |
| 5 | **fir** | 27.9 (40015) | 29 (4) | 28.9 (4) | 28.6 (9) | 28.7 (15) | 28.7 (10) |
| 6 | **bs** | 28.1 (5612) | 28.6 (1) | 28.6 (1) | 28.6 (2) | 28.5 (5) | 28.6 (3) |

* Entry 1-3 is for a simulated 2×4 multi-core processor layout with a thermal threshold of 35℃, while 4-6 is for simulated dual core architecture with a thermal threshold of 30℃.

Table 4.8: Temperature optimization for the schedulable task sets

As can be seen from Table 4.8, even using our fast algorithm, calculating the optimal schedule (in terms of the minimum core temperature) requires a considerable time (almost 52 hours for the 50 task set) because of the need to perform an exhaustive search of all schedule possibilities. If only one successful schedule is found using the exhaustive search, all other algorithms (e.g. HP, BS, HPTM, S-G, DFS) would also eventually find this single schedule, resulting in all columns of the table being almost identical (that is an identical temperature and similar algorithm runtime). However, this case is expected to be rare, and usually the first successful schedule will not be the optimal one.

It is important to note here, that the in calculating the optimal schedule we use our LUT-based *TT* scheduling technique and not Hotspot. Using Hotspot with an exhaustive search for large task sets (such as the 50 task set) would require a considerable amount of time to complete. Of the other algorithms, the HP algorithm generally has the shortest runtime, but leads to the highest core temperature, as it tries to make all tasks complete as early as possible. The shorter gap between consecutive tasks cannot guarantee that there is enough time to cool down the chip. The rationale behind all the other optimizations listed here is to use the gaps to separate task execution or to use the possible slack to de-throttle the chip performance. The BS algorithm simply makes a task start as late as possible, but does not appropriately cater for the case where sibling tasks run in parallel. This is particularly noticeable in

the large task sets (e.g. 30 and 50 tasks) where there is a reasonable high level of sibling task parallelism. The HPTM algorithm attempts to evenly distribute the sibling tasks, and achieves a lower peak core temperature for the different task sets and benchmarks. The algorithms which de-throttle chip performance, such as S-G and DFS, perform even better (as would be expected), but at the expense of an increased execution time. This thermal improvement is particularly noticeable for the larger task sets (e.g. 30 and 50 tasks) as the average slack time is greater. For S-G, this means that there are more opportunities for putting the core into intermittent sleep, but calculating this is more complex resulting in a lengthy runtime of the scheduling algorithm. DFS performs slightly worse than S-G (in general) as de-throttling the frequency still results in a significant leakage power component, whereas putting the core to sleep consumes very low leakage power. This experiment shows that the core temperature can be optimized at the expense of performance, while satisfying a task set's schedulability in terms of both deadline and thermal constraints.

To show that our conclusions are more general, we also changed the task properties of the synthetic task sets to reflect two possible basic scenarios: different utilization (task execution time/task relative deadline) and different power consumption.

The higher task utilization scenario is tested by generating tasks with execution times in the range [80ms:120ms] and task slack times in the range [8ms:25ms]. The lower task utilization scenario is generated with task execution times in the range [2ms:60ms] and task slack times in the range [25ms:55ms]. Rows 1-3 in Table 4.9 are for the high utilization group and rows 4-6 are for low utilization group. A thermal threshold of 35°C is assumed.

| * | | **Opt.** | **HP** | **BS** | **HPTM** | **S-G** | **DFS** |
|---|---|---|---|---|---|---|---|
| | | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) |
| 1 | **15 Tasks** | 36.7 (15782) | - (9) | - (10) | - (22) | - (39) | - (25) |
| 2 | **30 Tasks** | 33.6 (41528) | 34.9 (92) | 34.6 (72) | 34.5 (98) | 34.5 (184) | 34.6 (137) |
| 3 | **50 Tasks** | 34.1 (107201) | 34.9 (737) | 34.8 (721) | 34.7 (921) | 34.7 (2875) | 34.7 (1595) |
| 4 | **15 Tasks** | 33.2 (49743) | 34.8 (3) | 33.8 (3) | 33.3 (7) | 33.5 (12) | 33.5 (10) |
| 5 | **30 Tasks** | 32.2 (125151) | 34.3 (62) | 32.9 (85) | 32.4 (78) | 32.4 (120) | 32.6 (95) |
| 6 | **50 Tasks** | didn’t complete | 33.9 (195) | 33.3 (301) | 32.9 (376) | 33.1 (1059) | 33 (578) |

Table 4.9: High utilization scenario and low utilization scenario

In the high utilization group, the 15 task set is not schedulable with a 35°C thermal constraint. The runtime for performing an exhaustive search (the optimal solution) is

smaller, compared with the optimal group in Table 4.8, due to the smaller search space in the high utilization scenario. The runtime for the HP algorithm increases implying that finding a solution in the high utilization scenario is much harder using the forward search algorithm as the shorter slack time between tasks affects core cooling. All temperature values in the high utilization group are much higher than the temperatures shown in Table 4.8. This implies that a higher utilization will increase the peak temperature. HPTM has similar temperature results to S-G and DFS as a shorter slack minimises the advantages of HPTM.

For the low utilization group, the exhaustive search becomes significantly longer (the 50 task set does not complete within 2 days) due to the larger size of the problem window. Generally, the other runtimes and temperatures are smaller, implying that it is easier to determine a schedule without encountering core overheating. In this situation, HPTM is close to the optimal solution.

We also examined high power consumption and low power consumption task sets. The high power task set has power consumption in the range [4W:6W], while the low power task set has power consumption in the range [1.5W:4W]. The task execution time and slack are identical to the task sets described in Section 4.5.2. The results are presented in Table 4.10, where rows 1-3 are for the high power task sets and rows 4-6 are for the low power task sets.

| * | | **Opt.** | **HP** | **BS** | **HPTM** | **S-G** | **DFS** |
|---|---|---|---|---|---|---|---|
| | | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) | Temp (Time) |
| 1 | **15 Tasks** | 34.8 (31741) | 34.9 (5892) | 34.8 (7624) | 34.8 (11034) | 34.8 (19726) | 34.8 (15734) |
| 2 | **30 Tasks** | 37.3 (80245) | - | - | - | - | - |
| 3 | **50 Tasks** | 37.2 (189650) | - | - | - | - | - |
| 4 | **15 Tasks** | 31.2 (35289) | 32.9 (2) | 31.9 (2) | 31.5 (6) | 31.7 (12) | 31.5 (9) |
| 5 | **30 Tasks** | 32.2 (83872) | 33.1 (34) | 32.9 (46) | 32.6 (105) | 32.6 (187) | 32.7 (121) |
| 6 | **50 Tasks** | 32.7 (181924) | 33.8 (167) | 33.5 (197) | 33.1 (368) | 33.3 (2100) | 33.2 (1232) |

Table 4.10: High power consumption scenario and low power consumption scenario

For the high power scenario, the 30 and 50 task sets are not schedulable, while for the low power task sets the schedulability test is relatively simple (as the runtimes and core temperatures are smaller than those in Table 4.8). The 15 task high power task set (shown in row 1) requires much more time to find a valid schedule due to the higher temperature induced by the higher power tasks.

## 4.6 Summary

In this chapter, we propose a framework for thermal-aware scheduling of real-time task sets under a predefined thermal constraint. The power events, which contribute to a change in core temperature, are determined with the help of a task graph which allows us to construct a LUT-based thermal table. These thermal tables allow us to accurately and efficiently determine the thermal profile for a core in a multiprocessor system using simple table accumulation. This simple accumulation process can be integrated into a static task scheduling algorithm, to rapidly test for thermal-aware schedulability. We propose a forward search algorithm which determines the minimum-time schedule for a given thermal constraint, a backward search algorithm to quickly test for schedulability and a peak temperature minimization heuristic for performance/thermal optimization. We also discuss how our framework can be modified to include sleep functionality or frequency scaling to de-throttle chip performance. We demonstrate the performance of our proposed algorithms for thermal constrained scheduling using a number of practical and synthetic real-time benchmarks. We show that we are able to schedule large task sets (up to 50 tasks) in reasonable time (less than 11 minutes), which is 2 to 3 orders of magnitude faster than using scheduling with existing thermal simulation tools.

# Chapter 5

# Predictive Dynamic Thermal-Aware Scheduling with Leakage Power Modelling

We have previously classified DTAS into three categories (look-ahead, look-current and look-backward) and summarized the common issues in Chapter 2. Additionally, we emphasized the difference between DTAS and DTM: the former is a fine-grained active technique for thermal management, while the latter is a coarse-grained passive approach. The disadvantages of current DTAS research, particularly relating to multiprocessor systems, were outlined in Chapter 2 and are again summarized here for completeness and as our motivation for improving high level thermal management. These disadvantages include:

- DTAS in most other literature (e.g. ARMA[70] and PDTM[71]) is mainly based on the readings from on-chip digital thermal sensors (DTS), because this is the easiest and most direct way to determine the temperature on chip. We have analysed the inherent disadvantage of DTS in Chapter 2.2.2. Due to the long response time of DTS (usually 30-150ms [74]), it is not suitable for fine-grained DTAS, which needs to operate with the OS scheduler (e.g. the Linux default time tick is 10milliseconds [102]). Therefore, the predictive or proactive methods, which are proposed in [70][71] based on DTS, are more like the coarse-grained DTM policies, rather than DTAS.
- The training-based approaches used in ARMA and PDTM can efficiently obtain the temperature trend estimation. But their accuracies are affected by the training application sets, and covering all possible workload patterns for different applications is extremely difficult. In terms of DTM, the training-based approach is acceptable as the time interval between two consecutive DTM events is usually quite long and the processor workload of an application may change during this interval. However, for DTAS, two consecutive DTAS events can be very short but the temperature is unable to change dramatically

due to workload variations in this period because of the large thermal mass of a core[16].

- Techniques such as, TSIC [65], TEMPEST [60] and PDTM [71] use an empirical thermal model to simulate the heat exchange between adjacent cores. These models do not fully account for the thermal impact from other cores, and are thus these models are less accurate than the models used in thermal simulators [35][36][42][56].
- The more accurate thermal simulators (such as, HotSpot [35] and 3D-ICE [109]) cannot be efficiently integrated into the online thermal management and optimization process due to their large computational overhead, compared to the DTAS interval.
- Most DTAS research in the literature [71][92] does not use a complex power model. Usually, they ignore the leakage power consumption or regard the leakage power as a constant value which is not affected by temperature. This non-temperature-dependent leakage model eliminates the iterations needed for convergence between power and temperature in the thermal estimation. As a result, the constant leakage model will lead to the underestimation of temperature for each core, which increases the risk of real overheating.

In this chapter, we present an event driven thermal estimation method suitable for DTAS, based on power events and the prebuilt LUT introduced in Chapter 3. Moreover, this event-driven LUT approach is extended to include a leakage power model with reduced computational overhead, while still providing good accuracy. We also propose a technique for using occasional sensor based calibration to eliminate the effects of long term temperature drift. The chapter presents:

- A fast and accurate event-driven thermal estimator with an accuracy comparable to that of HotSpot, but with a computational overhead approximately three-orders of magnitude less, making it suitable for DTAS or fast high level thermal simulation.
- A temperature-leakage power relationship is added to the LUT-based model described in Chapter 3.

[16] This includes the effect of the heat spreader and heat sink.

- A technique to provide long term temperature calibration, based on existing high latency DTS, is added to eliminate thermal drift.
- A technique for improving the scalability of the thermal estimator for large many-core systems is proposed.
- Four look-ahead DTAS polices are developed based on a predicted thermal map derived using our fast event-driven thermal estimation. These policies are compared to existing DTAS policies from the literature.

Sections of this chapter have previously been published in [4][3] of the publications listed in Appendix C, and these sections have been reproduced with permission. 

## 5.1 Preliminary

Leakage power has become an important design consideration in modern processor design. In fact, since the 90nm technology node, leakage power and techniques for leakage power reduction have become a major concern in both high power and low power electronic systems. In the previous chapter we made the assumption that for low power embedded solutions, we could ignore the temperature dependent leakage component of the total leakage power, as the processor temperature was not extreme. However, for high performance processors, such as used in server or desktop applications, it is not possible to ignore the temperature dependent leakage component. Thus, it is important that a full leakage model is added to our fast LUT-based thermal estimator if it is to be used in the more general case. In this section, we introduce the leakage power model used with our LUT-based estimator, as well as the necessary data structures to support power events in a DTAS scenario.

### 5.1.1 Leakage Power Modelling in High Level Optimization

There are several important observations relating to leakage power indicating that leakage power should not be ignored in any sensor-less, or reduced sensor, thermal estimation.

- The sub-threshold leakage current dominants the total leakage power (gate leakage current can be ignored for high level optimization [53]) and the sub-threshold leakage for a module can be approximated by $A \cdot e^{-B/T}$, where $A$ and $B$ are constants, and $T$ is the temperature [53]. This exponential function can be replaced by a piece-wise linear (PWL) function [52] allowing the fast estimation of leakage power.
- A transistor consumes leakage power when idle (i.e. not switching). This leakage component is referred to as inactive leakage power, $P_{inactive}$ [52]. This component can be estimated using the PWL or exponential functions described earlier.
- Dynamic power, induced by switching activity, contributes to the transistor gate temperature. This affects the leakage power. Leakage induced by dynamic power is referred to as active leakage, $P_{active}$. The ratio of leakage power to dynamic power does not vary significantly with activity [53][56]. For example, [53] gives a ratio between static and dynamic power for the 28nm FinFET technology node in the range from 30% to 38%, based upon benchmarking on different architectures.

The total leakage power, $P_{static}$, consists of the inactive and active leakage components, as:

$$P_{static} = P_{active} + P_{inactive} \tag{5.1}$$

A simple and efficient n-piece-wise linear (PWL) function was proposed in [52] for estimating the inactive leakage power-temperature relationship, which has been detailed in Chapter 2.1.3. The simple 1-PWL implementation gives sufficient accuracy for high level thermal modelling, and is described by:

$$P_{inactive}(T) = F_{tech} S_{core} (\alpha T + \beta) \cdot V \tag{5.2}$$

where $\alpha, \beta$ can be predetermined by HSPICE simulation and experimentation; $F_{tech}$ is the leakage current per unit area, and depends on the manufacturing technology, design style and supply voltage $V$; and $S_{core}$ denotes the area of the core.

For active leakage, [56] gives an empirical equation that shows the ratio between the

dynamic power and the static leakage induced by signal activities, as:

$$P_{active}(T) = P_{dynamic} \cdot \left( \frac{R_0}{V_0 T_0^2} e^{\frac{B_{tech}}{T_0}} \cdot VT^2 \cdot e^{\frac{-B_{tech}}{T}} \right) \quad (5.3)$$

where $T_0$ is the ambient temperature; $R_0$ is the ratio between dynamic power and static leakage at $T_0$ and nominal voltage $V_0$; and $B_{tech}$ is a process technology constant that depends on the ratio between the threshold voltage and the sub-threshold slope, computed using the leakage current and saturation drive current numbers from ITRS 2001. The total static power is then the sum of the inactive and active leakage components, both of which are temperature dependent.

### 5.1.2 Power Events and their Data Structure

We have already defined the power event concept and explained how to capture power events in Chapter 3.2. In an online scenario, the event-driven LUT approach only updates the thermal map (the temperature estimation) every time a power event occurs. Thus, an additional data structure is needed to store all the power events which can affect the thermal map. The power events $ae$ are globally recorded for all cores on chip, according to their sequence, by an event list $\mathbb{Q}$ in the OS kernel. $\mathbb{Q}$ is a queue, as shown in Figure 5.1, where new events are enqueued onto the tail of $\mathbb{Q}$ and older ones are dequeued from the head of $\mathbb{Q}$. Each element tuple $ae(t, P, loc)$ in $\mathbb{Q}$ denotes one power event with several properties which have been explained in Chapter 3.2. In addition, two global variables $t_c$ and $t_p$ represent the current time instant and the time instant of the previous event occurrence (i.e. the last thermal map update). As shown in Figure 5.1, a new power event at the current time instant $t_c = 90$ is appended behind the element representing the last power event occurring at $t_p = 80$. In the next section, we show how to use such an event queue, including the conditions for event dequeuing, to update the thermal map dynamically.

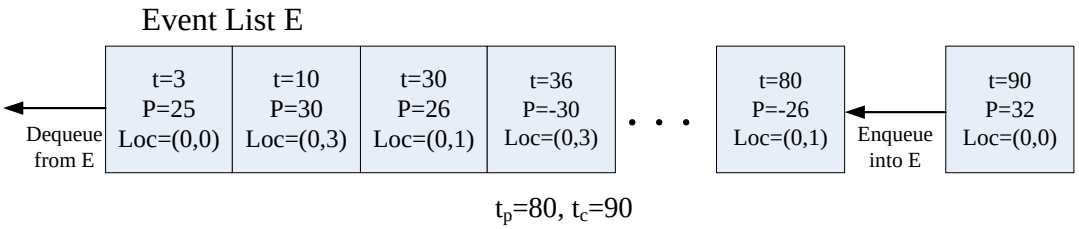



Figure 5.1: Data structure related to power events

## 5.2 Online Thermal Estimation in a Non-Temperature-Dependent Leakage Power Scenario

In Chapter 3.7 we presented two methods for using the prebuilt LUTs: firstly, table operations which were used in the schedulability test and STAS (offline) scenario described in Chapter 4; and secondly, row operations, with an even lower overhead, which are suitable for a DTAS (online) scenario. Both are deduced from the superposition principle, described by Theorem 3, which has an important prerequisite: the leakage power is constant (that is, a non-temperature-dependent power model).

In this section, we will firstly develop a leakage power LUT-based model for online thermal estimation assuming that the leakage power is not affected by temperature. Obviously, as described previously we cannot ignore the temperature-leakage power relationship, particularly for high performance processors. Thus, in a subsequent section, we will extend this to a non-linear temperature-dependent scenario, which represents a full leakage model.

The thermal map, which records each core's temperature, is updated each time an atomic power event occurs. The current thermal map $T_{t_c}$ at time $t_c$ can be calculated based on the previous thermal map $T_{t_p}$ at time $t_p$. That is, the current thermal map is obtained by adding the temperature increment of each core in the interval $\Delta t$ to the previous thermal map, as:

$$T_{t_c} = T_{t_p} + \Delta T_{\Delta t = t_c - t_p} \tag{5.4}$$

where $T_{t_c}$ , $T_{t_p}$ and $\Delta T_{\Delta t = t_c - t_p}$ are there $N$-element vectors which denote the current thermal map, the previous thermal map and the temperature increment of the $N$ cores,

respectively. The calculation of the temperature increment in the interval from $t_p$ to $t_c$ is given as:

$$\Delta T_{\Delta t=t_c-t_p} = \sum_{ae\in\mathbb{Q}} ae.P \times trans(\Re_{t_c-ae.t}^{tdl(ae.loc)} - \Re_{t_p-ae.t}^{tdl(ae.loc)}, ae.loc) \quad (5.5)$$

where $\Re_{t_c-ae.t}^{tdl(ae.loc)}$ represents the row corresponding to the time instant $t_c - ae.t$ in the LUT pointed to by the core location $ae.loc$. Similarly, $\Re_{t_p-ae.t}^{tdl(ae.loc)}$ denotes the row corresponding to the time instant $t_p - ae.t$ in the same LUT. The function $tdl()$, that indicates the TDL that the core at location $ae.loc$ belongs to, and the function $trans()$, used to simplify the calculation due to layout symmetry, were described in Chapter 3.6. In this case, the function $trans()$ transforms the temperature increment values $\Re_{t_c-ae.t}^{tdl(ae.loc)} - \Re_{t_p-ae.t}^{tdl(ae.loc)}$ associated with the first argument, based on the core location passed as the second argument. Thus, the temperature increment induced by one power event, described by Equation 5.5, can easily be obtained by indexing two rows of the LUT, subtracting them (the row operation proposed in Chapter 3.7), performing a simple transformation (a column reorder) and then multiplying by the power value $ae.P$. Lastly, based on Theorem 3, the thermal increments induced by every individual power event can be accumulated to give the full thermal response for the entire chip.

When an event occurs, in addition to updating the thermal map, the event list also needs to be updated. Older events, which no longer induce a temperature change, should be dequeued from $\mathbb{Q}$. The criterion $t_c - ae.t > Steady$ decides which events should be dequeued, where *Steady* is the time constant from initial state to steady state and its value is listed in the last row of the LUT. Deleting older events, which have reached steady state, shortens the event list and improves the performance of event driven estimation, as once an event reaches steady state it does have any further effect on core temperature. An example showing our proposed thermal estimation technique is presented in Appendix B.

Our proposed technique is not only used to update the thermal map, but could also be applied to thermal map prediction to estimate the temperature at next time period $t_n$. Algorithms for performing a thermal map prediction are presented in Section 5.4.

## 5.3 Online Thermal Estimation on a Temperature-Dependent Leakage Power Scenario

In practice, as mentioned in Section 5, leakage power is a function of temperature and cannot be ignored. If the leakage power varies with temperature, the previous simplified linear thermal model converts into a non-linear model whose solution depends on an iterative procedure [35][53] which is not computationally efficient.

For example, Hotspot can be modified to include the leakage power model described by Equations 5.1—5.3. However this then requires that Hotspot is called using an iterative procedure, where the leakage power is updated according to the current temperature and added to the dynamic power as a new power input to Hotspot for the next iteration. This iterative process, where the leakage power varies with both dynamic power and core temperature, needs to be performed at a reasonably fine time interval otherwise the temperature errors become significant. Unfortunately, this increases the already high computational overhead of HotSpot significantly. However, it does allow us to get an accurate transient temperature profile, so that we can compare the performance of our LUT-based leakage aware thermal estimator. That is, Hotspot, with the leakage power model added, is being used only for comparison purposes, and is not used in our online thermal estimation or DTAS implementations. This modified Hotspot procedure for the temperature-dependent scenario is shown in Figure 5.2.

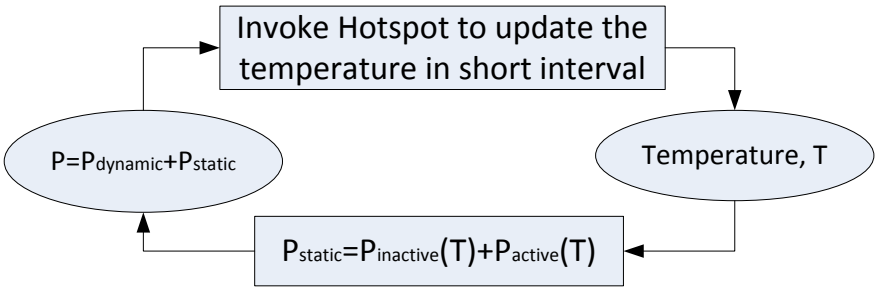



Figure 5.2: Iterative procedure to use HotSpot in temperature-dependent scenario

To demonstrate the temperature dependant leakage effect, we compare this procedure with the original HotSpot (which has a non-temperature-dependent model described by the LODE set in Equation 2.9 and cannot deal with the power-temperature leakage relationship) by using a set of power inputs under different initial temperatures.

Figure 5.3 shows an example of the temperature difference between the temperature-dependent and the non-temperature-dependent case for a processor core with different initial temperatures and task powers. Several observations can be concluded from this comparison:

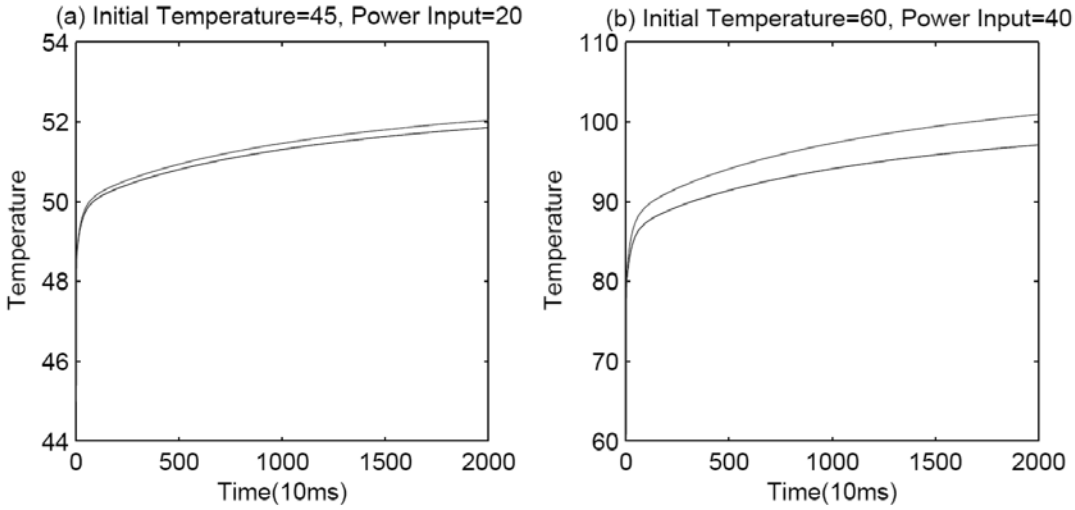



Figure 5.3: The temperature profile for the temperature-dependent (upper) vs. non-temperature-dependent (lower) simulations for different initial temperature and power

- The non-temperature-dependent model underestimates the temperature as it ignores the leakage power induced by temperature.
- A higher initial temperature or a larger initial power input to a core has more effect on the final steady state temperature.

To eliminate the iterative overhead between leakage power and temperature evaluation, we propose an empirical calibration factor $r$ to compensate the offset between two temperature curves based on the above observations. This calibration factor is a function of time $t$, power input $P$ and core temperature $T$, and adjusts the thermal response (the LUT values in our case) to account for the temperature leakage power dependence. This converts the non-linear problem into an approximately linear one, with a small error. For a given architecture, and for various $P, T$ pairs, we can plot the temperature profiles over t for both the non-temperature-dependent case and the temperature-dependent case, as shown in Figure 5.3. The ratio between the two profiles is $r$. After examining the temperature difference profiles for a number of processor architectures, such as Alpha 21264, ARM7TDMI, PowerPC 405, we found $r$ to be similar to the simplified expression for annealing, $T(t) = \frac{T_0}{1+kt}$. We make the observation that for any power event the ratio between the non-temperature dependent

leakage-temperature profile and the temperature dependent profile will initially be one (both temperature responses start at the same initial temperature) and at steady state will exhibit a steady state offset $S$. This offset will be a function of temperature $T$ and power $P$. Thus, we determine the calibration factor $r$ as:

$$r(t,P,T) = S - \frac{S-1}{kt+1} \tag{5.6}$$

where $k$ is the annealing rate and has a hyperbolic characteristic:

$$k = A + Bt^{-\gamma} \tag{5.7}$$

The calibration factor at the current time instant is based on the thermal characteristics at the previous instant. Thus we introduce a variable change to modify Equation 5.6 as:

$$r(t,P,T) = S - \frac{S-1}{(A+Bt^{-\gamma})(t-1)+1} \tag{5.8}$$

where $S$, $A$ and $B$ are functions of both the temperature $T$ and the power $P$, and are related to the leakage parameters in Section 5. The calibration factor $r$ thus depends on the power input $P$, the core temperature $T_{t_p}^{ae.loc}$ at time $t_p$ and the time interval $\Delta t = t_c - t_p$ between sequential power events. The core location $ae.loc$ only introduces a very small error and can be ignored. $\gamma$ is related to the leakage parameters, and is in the range $0.5 \leq \gamma \leq 1.4$[17]. However, to simplify the calculation we let $\gamma = 1$. The simplified calibration factor is then given as:

$$r(t,P,T) = S - \frac{S-1}{At - B/t - A + B + 1} \tag{5.9}$$

Plotting $S$, $A$ and $B$ versus temperature $T$ and power $P$ (Figure 5.4) shows that the surfaces are close to planar. Thus, for a given architecture, it is possible to use spot values for $S, A, B$ at three different $P$ and $T$ points and thus define the individual planes. This is done offline as shown in Algorithm 5.1. For example, for the 4×4 CMP layout, using $F_{tech} = 0.048\, ample/mm^2$, $S_{core} = 16mm^2$, $\alpha = 0.04$, $\beta = 0.2$,

[17] The $\gamma$ value is determined offline by simulation, for a specific processor layout. For the Alpha, $\gamma = 0.752$; for the 4 × 4 CMP, $\gamma = 0.823$.

$B_{tech} = 1.3V$ and $V = 1.3V$ [53][56] in the leakage model of Section 5, we can determine a 3×3 leakage parameter matrix $L$ offline, where $L \cdot [T\ P\ 1]^T = [S\ A\ B]^T$, as:

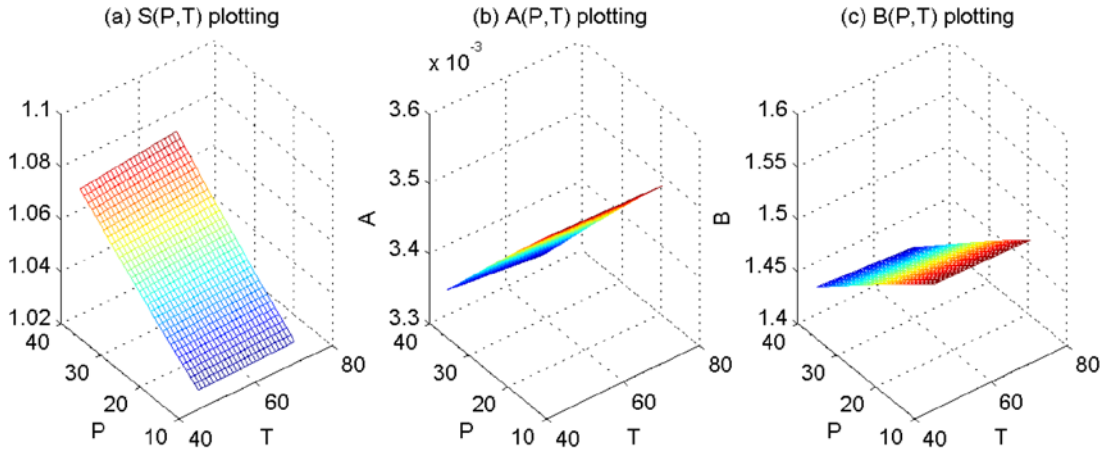



Figure 5.4: Plots of S, A, B versus power and temperature for a 4×4 CMP

$$L = \begin{bmatrix} 0.0013 & 1.7426 \times 10^{-4} & 1.0061 \\ -7.03 \times 10^{-7} & -6.0070 \times 10^{-6} & 0.0033 \\ -0.0035 & -2.8987 \times 10^{-4} & 1.7552 \end{bmatrix}$$

Due to the non-linear property of the temperature-dependent leakage model, temperature increment in the heating stage and temperature decrement in the cooling stage are no longer symmetrical. Because the leakage power in the cooling stage is also affected by the temperature, especially by the higher temperature at the end of the heating stage, the actual cooling progress is a little slower than that of the non-temperature-dependent leakage model case. As such, we apply an annealing factor ($0.99 \leq \sigma \leq 1$) to account for the difference between heating and cooling. We use: if $ae.P < 0$, then $\sigma = 0{:}\,995$; else $\sigma = 1$.

In Algorithm 5.1, $PF_{non\_dep}$ and $PF_{dep}$ denote the profiles for the non-temperature-dependent case and the temperature-dependent case respectively. And $pf^t_{non\_dep}$ and $pf^t_{dep}$denote the spot temperature on the corresponding profile, at time point $t$. For a specific architecture, Algorithm 5.1 only needs to be executed once to generate the matrix *L*, describing the *S*, *A* and *B* planes. The online temperature dependant thermal estimator uses *L* to evaluate the calibration factor r to calibrate the linear thermal response represented by our prebuilt LUTs. Using this method to calibrate the LUT requires a change to the temperature increment calculation (Equation 5.5), as:

$$\Delta T_{\Delta t=t_c-t_p} = \sum_{ae\in\mathbb{Q}} \sigma \times ae.P \times trans(\Re_{t_c-ae.t}^{tdl(ae.loc)} \cdot r(t_c - ae.t, ae.P, T_{t_p}^{ae.loc}) - \Re_{t_p-ae.t}^{tdl(ae.loc)} \cdot r(t_p - ae.t, ae.P, T_{t_p}^{ae.loc}), ae.loc) \quad (5.10)$$

**Algorithm 5.1: Plot S, A, B planes for Different Architecture**

**Input**: $P_{range}$ and $T_{range}$, and leakage parameter $F_{tech}, S_{core}, \alpha, \beta, B_{tech}, V, \gamma$.
**Output**: $L$ denoting S, A, B planes

**FOR** $P = P_{range\_start}\ to\ P_{range\_end}$ **DO**
  **FOR** $T = T_{range\_start}\ to\ T_{range\_end}$ **DO**
    Obtain temperature profile $PF_{non\_dep}$ from original HotSpot according to $P, T$;
    Obtain temperature profile $PF_{dep}$ from iterative HotSpot according to $P, T$ and leakage parameters;
    Find two steady temperature $S_{non\_dep}, S_{dep}$ in both profiles;
    $S = S_{dep}/S_{non\_dep}$;
    Fetch the temperature values $pf_{non\_dep}^{t_1,t_2}, pf_{dep}^{t_1,t_2}$ on both profiles at any time $t_1, t_2$;
    $r_1 = pf_{dep}^{t_1}/pf_{non\_dep}^{t_1}$; $r_2 = pf_{dep}^{t_2}/pf_{non\_dep}^{t_2}$;
    Substitute $S, r_1, r_2, t_1, t_2$ into Equation 42 to form an equation set with two unknowns $A, B$;
    Solve $A, B$;
    Record $S, A, B$ for one combination of $P, T$;
  **END FOR**
**END FOR**
Plot S plane; Plot A plane; Plot B plane;
**RETURN** L;

The two calibration factors are determined at the time instances $t_c - ae.t$ and $t_p - ae.t$, corresponding to the two fetched rows of the LUT. The *S*, *A* and *B* parameters are calculated once only, based on the core temperature $T_{t_p}^{ae.loc}$ and the power input $ae.P$ at the previous event instant $t_p$.

While this approach simplifies the calculation of the temperature leakage power dependency, it does have the potential to introduce errors into the temperature calculation, such as in a thermal runaway scenario (a very high temperature and power combination could make the temperature and power non-convergent) where the *S*, *A*, and *B* plots can no longer be approximated by plane surfaces, as shown in Figure 5.5. For example, at a temperature $T \geq 111$℃ and power $P \geq 48Watts$, thermal runaway occurs (distinguished by the circled regions in Figure 5.5). Once thermal runaway occurs, the chip no longer works normally, and in the worst case is permanently damaged. However, a task's DTAS prediction should indicate that the temperature threshold for the core would be exceeded and therefore it would not be allocated. In the worst case, global DTM would trigger if there was an inappropriate DTAS allocation, preventing thermal runaway.

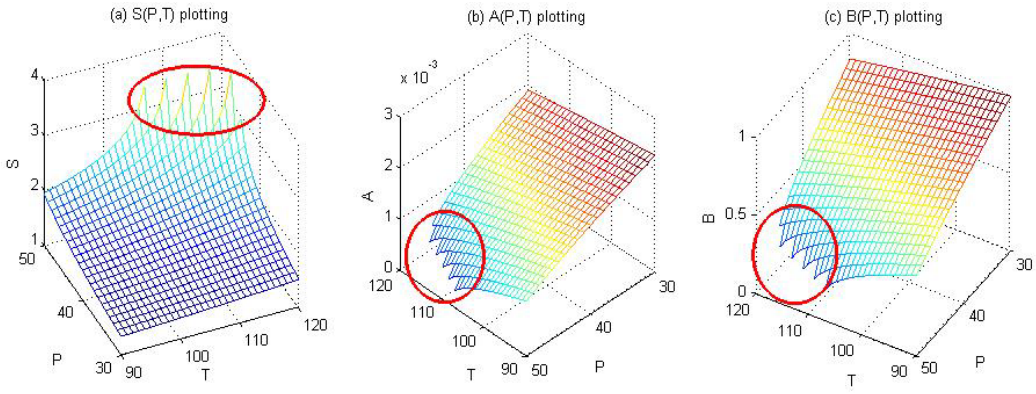



Figure 5.5: Thermal runaway described by S, A, B plots

## 5.4 Algorithms for Thermal Map Monitoring and Prediction

At each power event, the OS scheduler needs to update the thermal map and the event list. The procedure outlined in Algorithm 5.2, takes the previous thermal map $T_{tp}$ , the event list $\mathbb{Q}$, the current time instant $t_c$, the previous time instant $t_p$ and the newly arrived tasks $ae_{new}$ as inputs and returns the current thermal map $T_{t_c}$. Firstly, the algorithm traverses the event list $\mathbb{Q}$ and for each power event $ae$ calculates the two calibration factors $r_{t_c}$ and $r_{t_p}$ and then the temperature increment induced by the event (as in Equation 5.10). The temperature increments induced by each event in $\mathbb{Q}$ are accumulated in $\Delta T$. The current thermal map, $T_{t_c}$, is then obtained by adding $\Delta T$ to the last thermal map $T_{t_p}$. After updating the thermal map, older events are deleted from $\mathbb{Q}$ based on the criterion: $t_c - ae.t > Steady$. Finally, newly arrived events $ae_{new}$ are enqueued and the last time interval $t_p$ is updated to $t_c$.

Equations 5.4 and 5.5 can also be used as an estimator for a future thermal map at the next time instant $t_n$. Algorithm 5.3 is similar to Algorithm 5.2, except that we use current thermal map $T_{t_c}$, $\mathbb{Q}$, $t_n$ and $t_c$ as the four inputs to produce an estimate of the thermal map $T_{t_n}$, at the next (future) time instant. The current thermal map can come from Algorithm 5.2 or the readings from thermal sensors. In our simulated system, where task arrival is unknown (but can only occur relative to an OS timer tick), the event horizon is set to the next OS timer tick. However, the time interval between $t_n$ and $t_c$ could be any reasonable value decided by user. If the incoming task set was well known/defined, this interval could be determined based on the anticipated task

arrival times, and could be several OS timer ticks. Predicting the future thermal map is useful for guiding a scheduler, and allows various heuristic DTAS algorithms to be developed. That way, we could avoid allocating a task which would cause a core to overheat (or does not allow the core to cool sufficiently) by favouring a core with a smaller temperature increment (or a larger temperature decrement).

**Algorithm 5.2: Event-Driven Thermal Map Monitoring**

**Input**: $T_{t_p}, \mathbb{Q}, t_c, t_p, ae_{new}$
**Output**: $T_{t_c}$

$\Delta T = 0$;
**FOR** each $ae$ in $\mathbb{Q}$ **DO**
  Index two rows $\Re_{t_c-ae.t}^{tdl(ae.loc)}$ and $\Re_{t_p-ae.t}^{tdl(ae.loc)}$ from LUT;
  $[S \quad A \quad B]' = L \cdot \left[T_{t_p}^{ae.loc} \quad ae.P \quad 1\right]'$;
  $r_{t_c} = r(t_c - ae.t, S, A, B)$; $r_{t_p} = r\left(t_p - ae.t, S, A, B\right)$;
  $inc = trans(\Re_{t_c-ae.t}^{tdl(ae.loc)} \cdot r_{t_c} - \Re_{t_p-ae.t}^{tdl(ae.loc)} \cdot r_{t_p})$;
  $\Delta T = \Delta T + \sigma \times ae.P \times inc$;
**END FOR**
$T_{t_c} = T_{t_p} + \Delta T$;
**FOR** each $ae$ in $\mathbb{Q}$ **DO**
  **IF** $t_c - ae.t > Steady$ **THEN**
    Dequeue $ae$ from $\mathbb{Q}$;
  **END IF**
**END FOR**
Enqueue $ae_{new}$ into $\mathbb{Q}$;
$t_p = t_c$;
**RETURN** $T_{t_c}$;

**Algorithm 5.3: Event-Driven Thermal Map Prediction**

**Input**: $T_{t_c}, \mathbb{Q}, t_n, t_c$
**Output**: $T_{t_n}$

$\Delta T = 0$;
**FOR** each $ae$ in $\mathbb{Q}$ **DO**
  Index two rows $\Re_{t_c-ae.t}^{tdl(ae.loc)}$ and $\Re_{t_n-ae.t}^{tdl(ae.loc)}$ from LUT;
  $[S \quad A \quad B]' = L \cdot \left[T_{t_c}^{ae.loc} \quad ae.P \quad 1\right]'$;
  $r_{t_n} = r(t_n - ae.t, S, A, B)$; $r_{t_c} = r(t_c - ae.t, S, A, B)$;
  $inc = trans(\Re_{t_n-ae.t}^{tdl(ae.loc)} \cdot r_{t_n} - \Re_{t_c-ae.t}^{tdl(ae.loc)} \cdot r_{t_c})$;
  $\Delta T = \Delta T + \sigma \times ae.P \times inc$;
**END FOR**
$T_{t_n} = T_{t_c} + \Delta T$;
**RETURN** $T_{t_n}$;

## 5.5 Algorithm Extension

Two additional issues relating to the practical implementation of online thermal estimation and prediction need to be examined. These are: 1) long term drift or accumulated error can affect the accuracy of the temperature estimation. Can some calibration method be used to avoid this? 2) In a many-core scenario (with a very large number of cores), can the thermal map be updated in an effective (and scalable) way?

### 5.5.1 Thermal Calibration

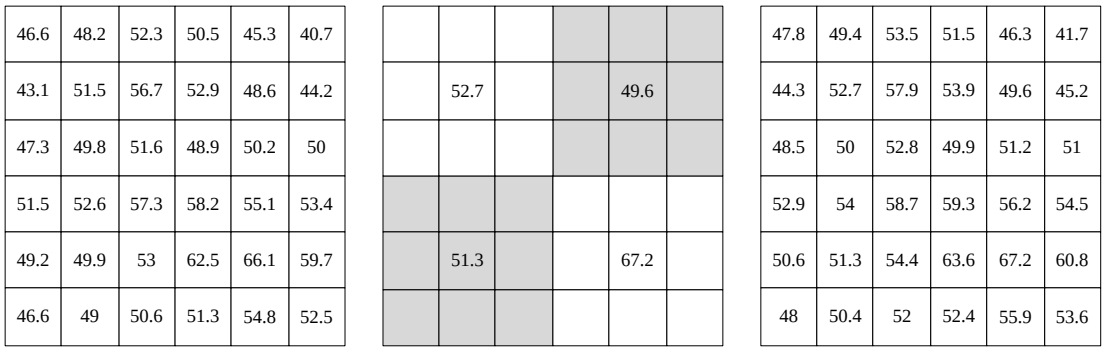
Our proposed sensor free thermal estimator, acts like an open-loop system as each event point update does not use any temperature feedback from the real chip itself, and is just based on the last thermal map. Additionally, it does not account for variations in individual core characteristics due to process variations, etc. While this could result in accumulated[18] errors in the long term, we show in Section 5.7.2 that the method is suitable for DTAS and is accurate enough for short to medium term thermal estimation. The reason for this minimal error is:

- Each power event introduces only a small error in the leakage calibration stage (as Equation 5.10 provides a good approximation for the effect of leakage power).
- The errors introduced by an increase in power tend to counteract those introduced by a decrease in power.

However, to eliminate any long term temperature drift, we propose using the on-chip DTS for coarse grained temperature calibration. These sensors are not suitable for fine grained thermal estimation due to their relatively slow access times. For multi-core systems with individual sensors on each core, it is relatively easy to use the sensor information to directly update the thermal map. However this approach is unlikely to scale well to large many-core systems, and instead, we would suggest that a single DTS, representative of a core in a cluster, be used to provide long term temperature

[18] It should be noted that DTS also suffers from similar problems.

stability. The reading from this sensor would be used to adjust the individual core temperature entries in the thermal map based on the difference between the current thermal map temperature and the measured temperature, as shown in Figure 5.6. Other techniques, such as [75], could be used if additional accuracy of the thermal map calibration is required.




(a) Estimated Thermal Map by Our Approach

(b) Readings From Thermal Sensor in Many-Core

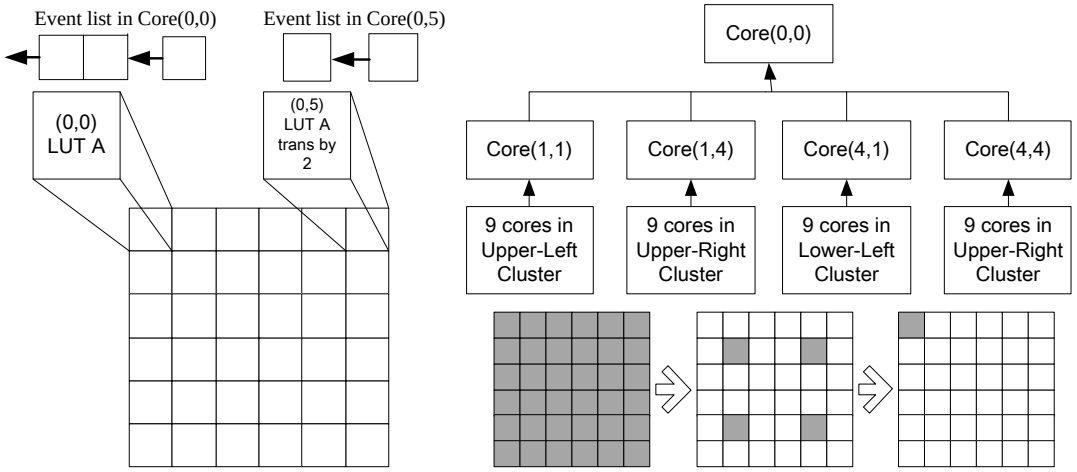
(c) Calibrated Thermal Map for Next Round Estimation

Figure 5.6: Example of calibration using real temperature readings

### 5.5.2 The Scalability of our Online Approach

The event-driven thermal estimation method, running as part of the scheduler in a centralized OS is only suitable for current CMP or small many-core systems. If the number of cores becomes large (e.g. several hundreds or even thousands as predicted), then a fully centralized algorithm will not be effective, as the event list will grow with an increase in the number of cores. To overcome this problem, we propose that the event-driven accumulation can be distributed to a single core in a subset of cores. That is, calculation is performed in parallel over regions of the processor array, by a single core in each region. Figure 5.7 shows a basic framework for a distributed version of our event-driven method, for a subset cluster size of one. For each core subset, we build an event list to replace the global event list. Similarly, the global LUT is no longer useful, and instead each core subset stores the LUT relevant to its location. In Figure 5.7, this is equivalent to a single transformed table, specific to the core's TDL, and as such the transformations in Chapter 3.5 and in Equations 5.5 and 5.10 are not needed. Therefore, each core subset only needs to calculate the temperature increments, calibrated by $r$, induced by its own power events. The final

step uses a single designated core (globally) to accumulate the results from all the cores to complete the thermal estimation. The only significant overhead induced by this distributed version is the communication necessary for accumulating the individual results from each core. It should be noted that the scheduler will still need to control task allocation globally after obtaining the overall thermal map.




(a) Basic Framework for Distributed Version

(b) Optimization of Hierarchy Collection

Figure 5.7: A distributed version of our online algorithm

## 5.6 Heuristic Predictive Task Allocation

If the OS kernel can estimate the temperature, better heuristic task allocation methods could be proposed to guide thermally aware DTAS. We propose several predictive task allocation policies based on our event driven thermal estimator.

### 5.6.1 Future Coolest First

Algorithm 5.3 allows the prediction of the future thermal map at the next time interval. Future coolest first simply finds the coolest idle core in the future thermal map and allocates the new task to that core. The rationale is that the coolest core is the most likely to be able to accommodate the power consumption of the new task and thus avoid overheating. This heuristic policy is very simple and efficient, being similar to coolest first [92] except that it uses the predicted temperature.

### 5.6.2 Future Neighbour Aware

Future neighbour aware, similar to neighbour aware [92], considers the future temperature of cores in the neighbourhood of the idle candidate. A thermal weight $T_w$ is calculated as:

$$T_w = a_1 T + a_2 T_{an} + a_3 T_{dn} + \frac{a_4}{N_{fe}} + \frac{a_5}{N_{ic}} \tag{5.11}$$

where $T$ is the temperature of the candidate core, $T_{an}$ is the average temperature of immediately adjacent cores, $T_{dn}$ is the average temperature of the diagonally adjacent neighbours, $N_{fe}$ is the number of free edges of the candidate core and $N_{ic}$ is the number of idle cores in the neighborhood, $a_1$, $a_2$, $a_3$, $a_4$ and $a_5$ are predefined constants to adjust the importance of each factor. Then, the idle core with the smallest weight indicates the best candidate for task allocation.

### 5.6.3 Future Task Aware

This policy considers both the temporal and spatial scale for thermal distribution. For each power task, we define a metric $R_{TT}$ relating temperature and the remaining task runtime, as:

$$R_{TT} = T \times (\frac{e}{t_n - t_s} - 1) \tag{5.12}$$

where $T$ is the is the temperature of the candidate core, $e$ is the execution time, $t_s$ is the task start time and $t_n$ is the prediction time instant, and where $R_{TT} = 0$ for an idle core. For each idle candidate, we then calculate the weight as:

$$T_w = \frac{a_1}{N_{an}} \sum_{core \in an} R_{TT_{an}} + \frac{a_1}{N_{dn}} \sum_{core \in dn} R_{TT_{dn}} \tag{5.13}$$

where $an$ and $dn$ denote the immediately adjacent core set and the diagonally adjacent core set, respectively, $N_{an}$ and $N_{dn}$ are the number of cores in the two sets. $a_1$ and $a_2$ are predefined constants. Intuitively, we allocate the task to the core with

the smallest weight (i.e. the smallest average $R_{TT}$ around the neighbourhood). The rationale is that the temperature change at the start of a power task is rapid, and slows down over time. Also, the smaller the remaining execution time, the shorter the interval till the core becomes idle.

### 5.6.4 Future Temperature Trend

For each new task, we classify idle cores into two sets based on the difference in the current and predicted temperature, as: a temperature-increasing set ($Core_+$) and a temperature decreasing set ($Core\_$). The weight for each set is:

$$\begin{aligned} T_{w+} &= T \times a_+ \quad if\ core \in Core_+ \\ T_{w-} &= \frac{T}{a_-} \quad if\ core \in Core\_ \end{aligned} \tag{5.14}$$

where $a_+$ is the temperature increment and $a\_$ is the temperature decrement. For each set, we choose the core with the smallest weight, giving two possible candidates for task allocation: one is from $Core_+$ and the other is from $Core\_$. We randomly allocate the task to one of these cores. The rational here is that a smaller temperature increment (or a larger temperature decrement) in the next time interval should be better. This policy takes advantage of both the current and the future thermal map.

## 5.7 Experiments

In this section, we present the results of performance comparisons between our LUT-based event driven thermal estimator (both for thermal simulation and for DTAS), the HotSpot thermal simulator, and a number of fast thermal estimators [61][62] and DTAS algorithms [71][92][105] from the research literature. We use HotSpot in two ways: firstly, we use an iterative approach which accounts for the temperature leakage power dependence (referred to as variant-P) as described in Section 5.3 and secondly, the standard HotSpot approach which does not account for the temperature leakage power dependence (referred to as invariant-P). For invariant-P, the power task sets used by HotSpot are generated by Wattch at the same resolution as the HotSpot time

increment interval. For variant-P the power input to HotSpot is the sum of the power input for invariant-P and the static power calculated from Equations 5.1—5.3 at the previous time increment. All experiments are conducted on a PC with a 2GHz Intel Core 2 Duo, and 2GB of memory.

### 5.7.1 Validating the Event-Driven Estimator

To validate the accuracy and efficiency of our fast LUT-based thermal estimator, we examine its capabilities for uniprocessor core-level and micro-architectural level simulation. Core-level thermal estimation, where a single temperature reading is used to represent the core temperature, is suitable for high level thermal optimization, such as DTM or DTAS on a per core basis. At the micro-architectural level, thermal estimation or simulation must give a detailed thermal distribution on a per module basis, to aid thermal-aware floorplanning.

Firstly, we examine the accuracy of our estimator by examining the effect, on the uniprocessor core-level temperature, of categorizing a continuous power trace as a sequence of atomic events. Figure 5.8 shows the predicted core temperature, determined from an Alpha 21264 uniprocessor simulator using the power event profile from Figure 3.1. The lower pair of traces represent a synthetic continuous power (solid line) and the atomic power using a $P_{AE} = 3W$ (dashed line), while the upper trace represents the variant-P core temperature determined by HotSpot (iteratively called to include the leakage power/temperature relationship). The discrete points (dots) overlaying the HotSpot temperature profile represent the core temperature determined by our event driven thermal estimator at an atomic event in the lower (dashed) power trace. Figure 5.8 shows that there is good agreement between the temperatures determined by the two techniques (with a maximum temperature error of 0.1%).

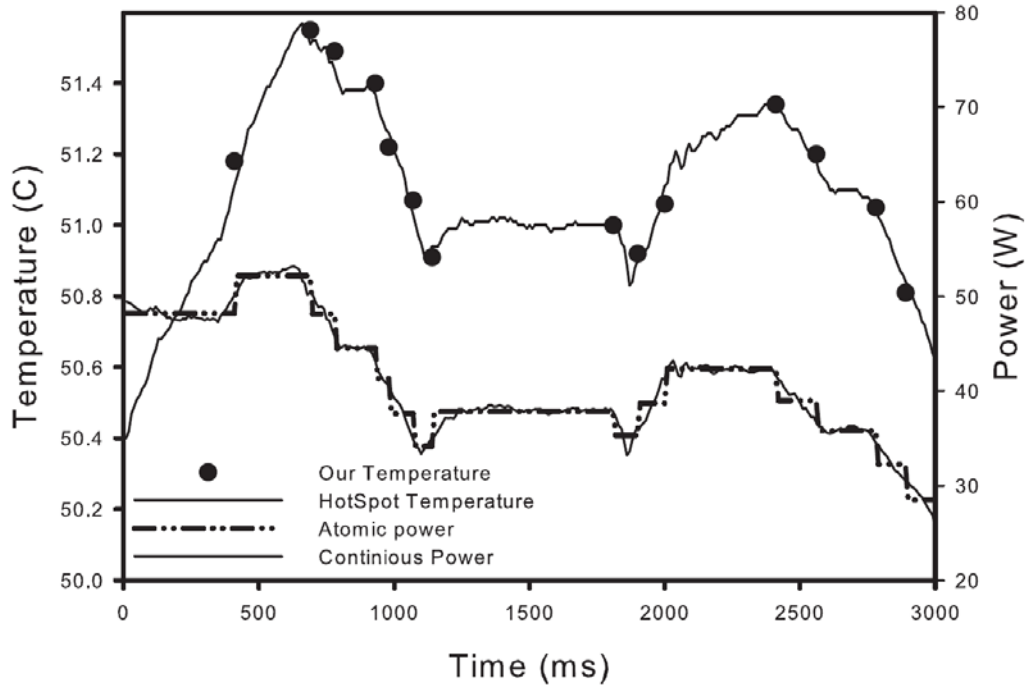



Figure 5.8: The variant-P thermal simulation of a single core processor using a synthetic power input

Next we examine a more realistic application, an MPEG2 decoder, running on the Alpha simulator over a much longer time period (47 seconds). Figure 5.9 shows that our estimated temperature (including the leakage power/temperature relationship), based on atomic events, accurately matches the results from HotSpot, but at a much reduced computational overhead. The execution time for the HotSpot simulation is 11372ms (using a 10ms interval), while the total execution time for our LUT-based thermal estimator is just 4.36ms (547 atomic events with a $P_{AE}$ of $1.2W$). This represents a runtime improvement of 3 orders of magnitude.

Our estimator has a significantly reduced computational overhead compared to HotSpot as we do not need to solve a large equation set in real-time. The algorithm complexity of our method is $\Theta(e \cdot N) = \Theta(r_{ae} \cdot Steady \cdot N)$, where e is the average number of atomic power events in the event list (the average numbers of atomic power events for the synthetic power input (above) and for MPEG2 are 48 and 92, respectively), $r_{ae}$ is the average arrival rate of power events, $N$ is the number of cores (modules) in the processor. Since $N$ and $Steady$ are a constant for a given processor, the algorithm complexity only depends on $e$ and $r_{ae}$.

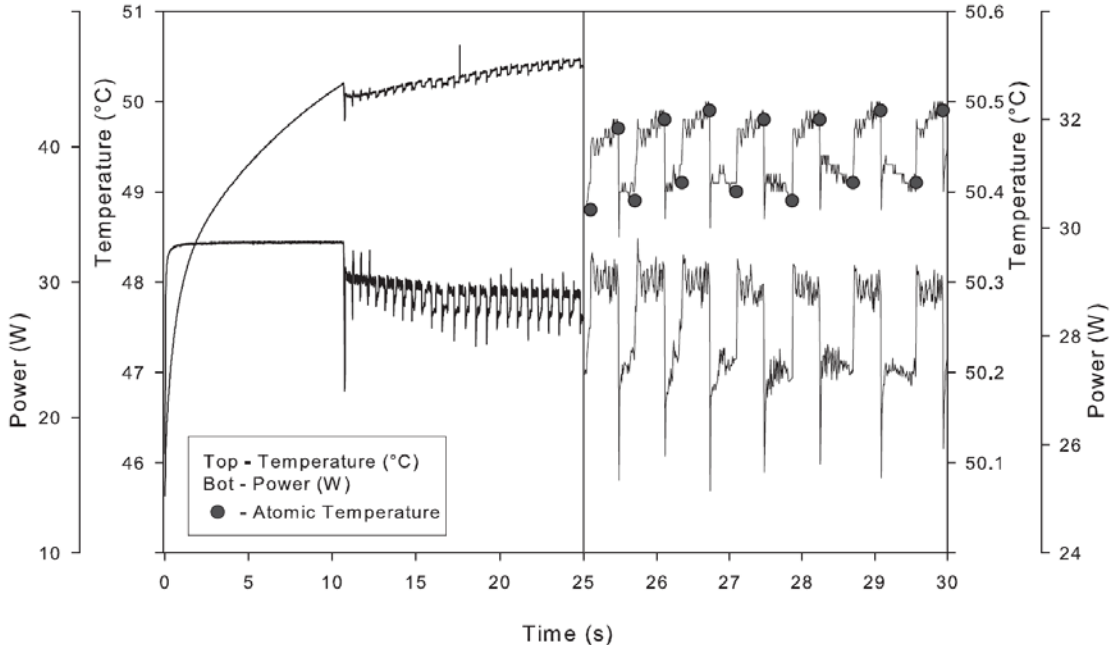



Figure 5.9: The variant-P thermal simulation for an MPEG2 decoder at the core level

Lastly, we use the Alpha 21264 simulator to examine the suitability of our estimator for uniprocessor core-level and micro-architectural level simulation. This experiment consists of 3 separate scenarios.

In the first scenario (Table 5.1), we perform a variant-P core level simulation (including the temperature/leakage power relationship) with a $P_{AE} = 3Watt$. In this scenario, the HotSpot iteration time is set at 10ms. The average temperature error (absolute error relative to the HotSpot temperature), the average computational overhead for a single thermal map update, and the actual simulation runtime for each benchmark are presented in Table 5.1. Table 5.1 shows that there is good temperature agreement with HotSpot, but with a 3 order of magnitude reduction in the simulation runtime. It should be noted that the total simulation execution time for the benchmarks in Table 5.1 is approximately the product of the overhead for a thermal map update by the number of atomic events in the queue, over the whole simulation.

| Scenario | Benchmark | gcc | gzip | bzip | lucas | mesa | parser | swim | vortex | mpegdec |
|---|---|---|---|---|---|---|---|---|---|---|
| | Execution Runtime (s) | 9.5 | 8 | 18.6 | 15.5 | 6.7 | 10.2 | 3.7 | 25.5 | 47 |
| **Scenario 1** Core Level $P_{AE} = 3Watt$ Variant-P | Our Temp. Error (°C) | 0.14 | 0.16 | 0.06 | 0.13 | 0.2 | 0.16 | 0.25 | 0.03 | 0.07 |
| | Our Overhead (µs) | 10 | 8 | 9 | 8 | 7 | 6 | 10 | 11 | 8 |
| | Our Runtime (ms) | 0.49 | 0.496 | 0.803 | 0.464 | 0.364 | 0.312 | 0.15 | 1.034 | 3.36 |
| | HotSpot Runtime (ms) | 2175 | 1923 | 4378 | 3879 | 1769 | 2472 | 955 | 6550 | 11372 |

Table 5.1: Core Level error and overhead in temperature-dependent leakage power scenario

The second scenario (Table 5.2) is similar to the scenario above, except that we

perform a 30 module micro-architectural level simulation with a $P_{AE} = 1.2Watt$. Again, there is close agreement between our event driven method and the results from HotSpot with a 3 order of magnitude reduction in simulation time. The thermal map overhead has increased relative to scenario 1 (above) because of the need to accumulate all the events on the 30 modules in the micro-architectural simulation.

| Scenario | Benchmark | gcc | gzip | bzip | lucas | mesa | parser | swim | vortex | mpegdec |
|---|---|---|---|---|---|---|---|---|---|---|
| | Execution Runtime (s) | 9.5 | 8 | 18.6 | 15.5 | 6.7 | 10.2 | 3.7 | 25.5 | 47 |
| **Scenario 2** MA Level $P_{AE} = 1.2Watt$ Variant-P | Our Temp. Error (°C) | 0.31 | 0.22 | 0.11 | 0.17 | 0.34 | 0.22 | 0.29 | 0.12 | 0.09 |
| | Our Overhead (μs) | 285 | 234 | 245 | 221 | 246 | 204 | 292 | 278 | 201 |
| | Our Runtime (ms) | 27.93 | 25.74 | 56.6 | 43.54 | 21.4 | 24.15 | 15.48 | 70.9 | 124.02 |
| | HotSpot Runtime (ms) | 77185 | 62709 | 145672 | 121375 | 52955 | 80143 | 29171 | 197447 | 386190 |

Table 5.2: MA Level error and overhead in temperature-dependent leakage Power scenario

In the third scenario (Table 5.3), we examine the performance of our LUT-based estimator by comparing it with some of the fast thermal estimators [61][62] from the literature. FATA [61] implements an improved 4$^{th}$ order Runge-Kutta solver with an adaptive step size. TMM [62] takes advantage of moment matching in the frequency domain, where temperature can be calculated as the convolution of the power input's response. Since FATA and TMM (and the original HotSpot) do not model the temperature/leakage power dependency for transient temperature estimation, we only consider the invariant-P case so as to provide a fair comparison (this is why FATA and TMM are not included in scenario 1 and scenario 2 above. That is, we have removed the temperature/leakage power relationship described by Equations 5.6-5.9 from our estimator. We have set the window size of TMM to 10000. The average temperature error (as absolute error relative to the HotSpot temperature) and the actual simulation runtime for each benchmark are presented in Table 5.3. Our thermal estimation has a 2000--3000x speedup compared to HotSpot, and is 20--40x faster than FATA and TMM while maintaining similar temperature errors.

| Scenario | Benchmark | gcc | gzip | bzip | lucas | mesa | parser | swim | vortex | mpegdec |
|---|---|---|---|---|---|---|---|---|---|---|
| | Execution Runtime (s) | 9.5 | 8 | 18.6 | 15.5 | 6.7 | 10.2 | 3.7 | 25.5 | 47 |
| **Scenario 3** MA Level $P_{AE} = 1.2Watt$ Invariant-P | Our Temp. Error (°C) | 0.145 | 0.164 | 0.082 | 0.132 | 0.159 | 0.178 | 0.251 | 0.034 | 0.069 |
| | FATA Temp. Error (°C) | 0.097 | 0.112 | 0.038 | 0.135 | 0.181 | 0.136 | 0.182 | 0.056 | 0.067 |
| | TMM Temp. Error (°C) | 0.08 | 0.105 | 0.05 | 0.062 | 0.172 | 0.128 | 0.091 | 0.051 | 0.059 |
| | HotSpot Runtime (ms) | 51869 | 43679 | 101357 | 84893 | 36488 | 55952 | 20119 | 135238 | 255755 |
| | Our Runtime (ms) | 18.94 | 18.12 | 37.92 | 30.15 | 16.79 | 20.17 | 9.32 | 47.35 | 87.29 |
| | FATA Runtime (ms) | 519.6 | 487.7 | 1029.8 | 763.8 | 312.6 | 750.2 | 193 | 1274.4 | 2158.3 |
| | TMM Runtime (ms) | 678 | 651.9 | 1535.7 | 1081.4 | 499.8 | 799.3 | 283 | 1788.9 | 3148.1 |

Table 5.3: MA Level error and overhead in non-temperature-dependent leakage power scenario

These experiments show that it is feasible to estimate core/module temperature based

on a power trace decomposed into atomic events. This demonstrates that only a few power events are sufficient and accurate enough to describe the temperature transient, thus suggesting that event driven estimation is likely to be suitable for high level DTAS, DTM and fast thermal simulation in a multi/many core environment.

### 5.7.2 Event Driven Thermal Estimation for CMP systems

We have developed a 4×4 CMP thermal estimator, which can update the thermal map and track atomic power events using our event driven approach. The thermal RC model and parameters used for each processor are the same as those used in the HotSpot simulations, and thus the LODE used to build the LUTs is identical to that used in HotSpot. The leakage parameter matrix, $L$, for the 4×4 CMP is the same as the one pre-calculated in Section 5.3. Both the LUTs and the leakage parameter matrix are calculated offline.

To test the performance of our proposed thermal estimator we generate a number of artificial power task sets based on power profiling of selected applications in SPEC CPU 2000. These power task sets are derived using SimpleScalar Alpha and Wattch, which are then converted into consecutive atomic power events. For example, the GCC benchmark can be converted into several tasks with an average power consumption in the range [15W:30W][19], determined using a power threshold $P_{AE} = 5W$, and with execution times in the range [10ms:500ms]. Our event driven approach is unrelated to the features of a task set, and is applicable to any power profile. We assume that the arrival time of the tasks is randomly distributed in the range [0s:30s]. Each task set contains around 200 power events (parts of the benchmarks) that are randomly distributed among the cores.

We use these task sets to examine the performance of our estimator, relative to HotSpot, on a 4×4 CMP. Here, we examine the contribution of the leakage power to the CMP core temperature by examining both the invariant-P (no temperature/leakage power dependence) and the variant-P (which accounts for the temperature/leakage power dependence) scenarios. In this experiment, to reduce the required simulation

[19] L2 cache power is not included.

time, we assume that the core has already been heated above ambient temperature. That is, we assume that the processor has been operational for some time and that the core temperature is at an initial temperature of 45°C ($T_{t=0ms}$= 45°C). This relatively high initial temperature has been chosen as it enables us to better illustrate differences in the two modelling scenarios and is applicable for the general purpose computing domain, where the operational core temperature is significantly higher than ambient temperature. The HotSpot time increment is 100μs, and DTM (e.g. DVFS, migration and clock gating, etc.) is not triggered. We simply observe whether our estimation of the thermal behaviour is similar to that produced by HotSpot.

Figure 5.10 shows the temperature estimation results for a 5000ms task set subsection for selected cores in the CMP. The bottom solid line (grey) shows the invariant-P HotSpot temperature transient for cores (0,0), (1,1), (1,3) and (3,3) of the 4×4 CMP, while the upper solid line (black) shows the temperature profile for the HotSpot variant-P scenario (i.e. the complete leakage power model). The asterisks (*) represent our event driven estimation of the core temperature, at each update point where an atomic event occurs. Figure 5.10 shows that there is a significant difference between the two temperature profiles determined by HotSpot. These results are not unexpected, and show that the temperature leakage power dependence cannot be ignored in high power processors, thus validating the inclusion of a full leakage model into our event driven estimator. As such, we will not consider the HotSpot invariant-P model further. Figure 5.10 also shows the accuracy of our proposed fast event driven estimation (the asterisks coincide with the HotSpot iterative leakage power (variant-P) model line), validating our calibration factor based approach described in Section 5.3.

To determine the speed-up of our event driven estimator, we compare our algorithm to HotSpot, using different time intervals when building the LUTs. To make the comparison fairer, we also vary the time increment interval for HotSpot. Table 5.4 compares the absolute worst case temperature error, the average temperature error and the simulation run-time for the 30s task sets described earlier. The worst case error is relative to the HotSpot temperature profile determined using a time increment interval of 100μs. Table 5.4 shows that the error increases with the granularity of the time interval used to build the LUTs (uniform interval LUT). The HotSpot error decreases with granularity, but as the HotSpot interval approaches the task execution time, the

error increases significantly. Note that a finer granularity in the LUT also increases the memory requirements for our event driven estimator. In most cases, a 10ms time interval is a good choice as this gives good accuracy, while keeping the memory usage relatively low. The runtime increases with granularity for the LUT-based estimator because additional interpolation is required. The overhead for HotSpot decreases with increasing increment interval. Table 5.4 also shows that our event driven estimator has a significantly reduced runtime compared to HotSpot (3 orders of magnitude for a 10ms update).

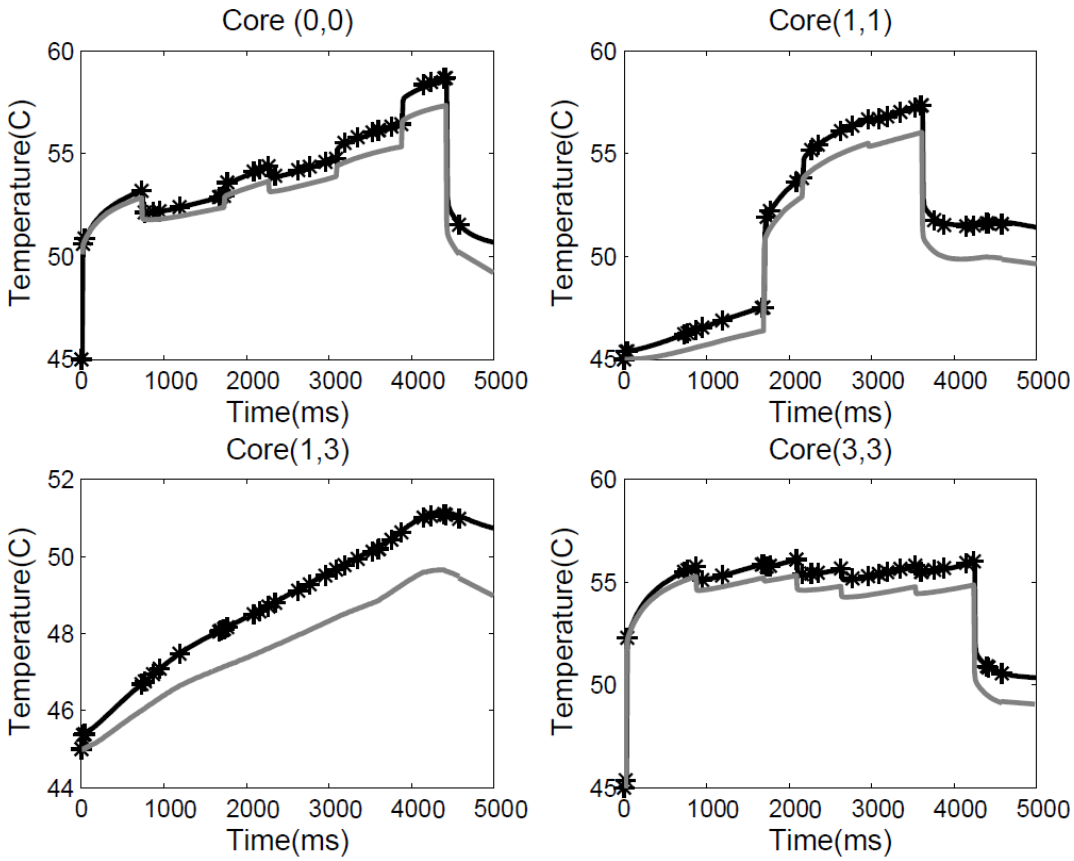



Figure 5.10: Thermal estimation validation for 4×4 CMP

| | 1ms-Interval LUT | | 5ms-Interval LUT | | 10ms-Interval LUT | | 50ms-Interval LUT | | 100ms-Interval LUT | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Ours | HotSpot | Ours | HotSpot | Ours | HotSpot | Ours | HotSpot | Ours | HotSpot |
| WC Error (°C) | 0.82 | 0.67 | 1.31 | 3.52 | 1.44 | 4.12 | 2.25 | 4.82 | 3.16 | 6.17 |
| Ave Error (°C) | 0.65 | 0.43 | 0.79 | 1.24 | 0.89 | 1.58 | 1.21 | 1.95 | 1.78 | 2.66 |
| Runtime (ms) | 20.3 | 42415 | 22.5 | 41374 | 24.4 | 39736 | 26.7 | 38645 | 31.1 | 37221 |
| Memory (MB) | 1.22 | -- | 0.535 | -- | 0.136 | -- | 0.047 | -- | 0.02 | -- |

Table 5.4: Performance, average error, overhead comparison

To determine the long term accuracy of our temperature estimator we run the simulator for a longer time period (e.g. 2min, 5min, 10min, and 20min). In this experiment we do not use calibration from external sensor readings. The power task set, from the previous experiment described in the second paragraph of Section 5.7.2, is regenerated to contain more power events over a longer time interval. The new task set contains 20000 tasks whose arrival instants are randomly distributed in the range [0s:20min], and whose execution times are extended to the range [10ms:2min]. A 10ms-interval LUT and a $P_{AE} = 5W$ is used for this scenario. Table 5.5 shows the average error and the worst case error for the different simulation periods. The long term simulation of our open-loop thermal estimator does not result in significant errors, compared with HotSpot. As explained in Section 5.5.1., the reasons for this accuracy are that each power event introduces only a small error in the leakage calibration stage and errors introduced by an increase in power tend to counteract those introduced by a decrease in power. The worst case error usually appears at the end of power tasks with a long execution time (e.g. larger than 20s). Thus, we can conclude that Equation 5.9 is suitable for cases where the power task execution time is below 20s, but underestimates the leakage power error closer to the steady state.

| **Runtime Length** | **2min** | **5min** | **10min** | **20min** |
|---|---|---|---|---|
| Average Temp. Error | 1.10% | 1.95% | 2.97% | 2.35% |
| Worse Temp. Error | 2.65% | 4.21% | 4.82% | 4.63% |

Table 5.5: Algorithm performance for varying runtime

The next experiment examines how the power threshold value $P_{AE}$ (defined in Chapter 3 for profiling the power event) affects the error and overhead. As mentioned in Chapter 3.2, the $P_{AE}$ threshold value is used to define atomic events and thus affects the time complexity of our event driven thermal method. While a coarse grained power profile is suitable for DTAS, a smaller $P_{AE}$ can be used to observe more detail in the thermal behaviour (such as for thermal simulation at the architectural level). However, very small values of $P_{AE}$ result in a fine-grained power profile, generating a large number of power events which need to be queued to the event list, resulting in a very significant computational overhead. For this experiment, a subset of the task set from the first example (described in the second paragraph of Section 5.7.2) is regenerated using the different $P_{AE}$ threshold values shown in Table 5.6. As expected, the temperature error increases with $P_{AE}$ while the overhead decreases. We would

suggest a $P_{AE}$ value of 2W to 5W for DTAS and a $P_{AE}$ value of 0.5W to 1W for thermal simulation.

| $P_{AE}$ | 0.05Watt | 0.2Watt | 1Watt | 5Watt |
|---|---|---|---|---|
| Average Temp. Error | 0.004% | 0.29% | 0.46% | 1.05% |
| Average Overhead ($\mu s$) | 683 | 379 | 198 | 126 |

Table 5.6: Algorithm performance for varying input power granularity

### 5.7.3 Heuristic Predictive Task Allocation for DTAS

In this section, we examine the performance of our fast event driven thermal estimator, in a DTAS scenario, when combined with our proposed heuristic scheduling policies. We compare our future coolest first (FC), future neighbour aware (FN), future task aware (FTA) and future temperature trend (FTT) policies, all based on the future thermal map, with previous dynamic scheduling approaches (e.g. coolest first (C) [92], neighbour aware (N) [92], and historical window for possibility of allocation (HWP) [29]), which are based on the current thermal map or historical thermal information in terms of the peak temperature, the average temperature and the spatial diversity. We also examine the non-TAS case with a random core allocation to show the effect on core temperature of not using DTAS. In FN we use the same parameters as in [92] ($a_1 = 0.45$, $a_2 = 0.25$, $a_3 = 0.15$, $a_4 = 5.1$ and $a_5 = 2.2$) and in FTA we use $a_1 = 0.7$ and $a_2 = 0.3$. Some existing DTM policies can also be modified to a DTAS scenario and thus can also be used for comparison purposes. We have modified the incremental task allocation (ITA) algorithm from [105] (and converted from a 3D model to a 2D model). Here we assume that the speed for each core is a constant, and that tasks arriving simultaneously are sorted in descending power order. We also compare our DTAS algorithms with the predictive dynamic thermal management (PDTM) algorithm proposed by [71]. Here we use the temperature values from our simulator (rather than from DTSs) to generate the historical temperature profile used for the recursive application-based thermal prediction. We implement the predictive thermal model of [71], let $\Delta t_m = t_n - t_c = 10ms$, and assume priority adjustment is disabled. For this study, we generate 10000 task sets with similar characteristics to those described in the second paragraph of Section 5.7.2.

In the first scenario (Table 5.7), DTM is not used for comparing the peak/average

temperature and spatial diversity and we assume that the cores can always work normally without any safety threshold. The purpose of this assumption is to exclude the effect of DTM (e.g. the use of migration or DVFS when the temperature threshold is exceeded) and just observe if the dynamic scheduling policies are efficient in terms of minimizing the peak/average temperature and the spatial/temporal diversity on chip. Here, spatial/temporal diversity is used to measure the degree of thermal balancing among the cores. A lower diversity indicates that the policy has a better ability to balance the thermal side effects on chip, thus reducing ageing due to rapid heating/cooling and dramatic temperature differences. These two metrics are defined in [92].

| Metrics | Non-TAS | C | N | HWP | PDTM | ITA | FC | FN | FTA | FTT |
|---|---|---|---|---|---|---|---|---|---|---|
| Peak T (°C) | 133.7 | 119.65 | 104.5 | 98.57 | 99.76 | 110.31 | 106.12 | 98.63 | 95.72 | 94.35 |
| Ave T (°C) | 121.5 | 112.13 | 102.64 | 96.15 | 99.45 | 102.19 | 102.56 | 95.04 | 95.29 | 91.81 |
| Ave SD (°C) | 15.68 | 12.37 | 4.83 | 3.24 | 3.35 | 3.61 | 8.76 | 4.32 | 3.85 | 4.12 |
| Ave TD (°C) | 20.56 | 28.75 | 24.65 | 14.53 | 15.75 | 14.84 | 28.32 | 25.51 | 16.78 | 18.45 |
| Overhead (μs) | 8 | 37 | 98 | 145 | 235 | 25 | 146 | 157 | 172 | 152 |

Table 5.7: Temperature Optimization and Comparison amongst Different DTAS Policies

Table 5.7 shows that TAS based policies are superior to the non-TAS case. Without DTAS, the average temperature significantly higher than even the worst of the TAS policies (e.g. coolest first). Additionally, the results for the dynamic scheduling polices based on our predictive future thermal map have much better effect on the peak temperature (Peak T) and the average temperature (Ave T) minimization. FC and FN are superior to the simple C and N schedulers, showing that the look-ahead approach is an effective technique for avoiding unpredictable hot cores on chip. In terms of minimizing temperature, FN and HWP[20] have a similar effect, while the two more complex predictive polices FTA and FTT, perform better than HWP and PDTM. ITA, with the smallest overhead, is only comparable to the simple C and N schedulers. It is notable that HWP and PDTM both achieve better average spatial diversity (Ave SD) and average temporal diversity (Ave TD) results than FTA and FTT. The reason is that we do not take advantage of the future thermal map to carry out task migration, as in [70] and [71], and the future time slot is too short compared to historical windows (longer historical windows are more useful to balance the thermal distribution than shorter prediction if we don't use migration or other DTM mechanisms). In other

[20] The temperature history used in our implementation of this algorithm is derived directly from HotSpot, rather than simulated thermal sensors.

words, our policies only consider how to heuristically optimize the temperature by using the instantaneous temperature (e.g. the future thermal map or current thermal map), rather than using the temperature transient over a long period. Our proposed algorithms have a higher computational overhead (Overhead), determined as the interval between the start of the evaluation of the future thermal map to the point where a core is identified for allocation of the task, than C, N and ITA, however this overhead is still acceptable particularly considering the improvement in thermal performance. Compared to HWP, our FTT technique has similar performance (better temperature minimization but slightly worse temperature diversity) with only a slight increase in complexity. However, it is uncertain if HWP and PDTM, with their reliance on sensor data, are able to scale to large many-core systems.

In the second scenario (Table 5.8), we use the same task set as above, but set a thermal safety threshold (we use 80°C) for each core to trigger DTM. Our simulator implements task migration in DTM by putting tasks on the hot core back into the ready queue and simply shutting the core down for a period of 50ms (during this time it consumes no power). The migrated tasks are inserted at the head of the ready queue for immediate scheduling to some cooler core. Other DTM mechanisms (e.g. DVFS) are not used in this example. We simply count the number of DTM trigger times (DTM TT) and observe the overall completion time (Complete T) of all 10000 power tasks. Intuitively, a reduced number of DTM trigger times indicates a smaller overhead induced by DTM, meaning that the whole task set would finish earlier.

Table 5.8 shows that the non-TAS case has significantly more DTM trigger events and thus leads to more chance of missing a software task deadline, resulting in a much longer completion time compared to the DTAS algorithms. Table 5.8 also shows that policies based on predictive allocation alleviate the DTM loading and its overhead. This is because allocating a task to the right core (before it starts running) is better than frequent migration at run-time, in terms of the completion time, as proactive allocation can avoid the repeated allocation/de-allocation on the same core and reduce the system overhead dramatically. FTA and PDTM performed best in terms of the number of DTM trigger events, with FTA having a reduced completion time due to the algorithm’s reduced overhead. In fact, when considering all 10000 power tasks, we could save as much as 8%--16% off the execution time when using our predictive

policies, comparing to the non-predictive policies. We conclude that scheduling, guided by the future thermal profile, will improve the overall system performance under strict DTM thermal constraints.

| Metrics | Non-TAS | C | N | HWP | PDTM | ITA | FC | FN | FTA | FTT |
|---|---|---|---|---|---|---|---|---|---|---|
| DTM TT (times) | 2321 | 1873 | 1625 | 1478 | 1357 | 1646 | 1573 | 1512 | 1377 | 1407 |
| Complete T (s) | 389.32 | 312.17 | 281.99 | 268.32 | 264.76 | 278.12 | 271.9 | 270.93 | 260.62 | 262.33 |

Table 5.8: DTM Times and Overall Task Completion Time amongst Different DTAS Policies

The last scenario (Table 5.9) shows the efficiency of our heuristic policies for use in a soft real-time system which allows tasks to be discarded if a deadline is missed. We generate a similar power task set to Scenario 1, with an additional attribute: the deadline for each task. The deadline (relative to the start time of the task) is assumed to be its execution time multiplied by a factor which is randomly distributed in the range [1.3:4]. DTM for migration is still enabled. We investigate the average response time (Resp Time) and the rejection ratio (Rejection Ratio) for the task set. The average response time is the latency between the arrival time and the start time of a task, while the rejection ratio is the number of tasks that are discarded. In evaluating how thermal-aware scheduling can affect the key measurements in real-time system performance, a lower response time and a lower rejection ratio are better.

Table 5.9 shows that the non-TAS case has a higher response time and a higher rejection ratio than for the DTAS algorithms. Table 5.9 also shows similar trends as in Scenario 1 and 2, indicating that policies that reduce DTM events are also better for time constrained scenarios. Temperature minimization and thermal balancing means that more tasks can be scheduled to be executed simultaneously making it possible to schedule a task earlier. FTA outperforms the other methods since it is the only heuristic policy to use task attributes in the scheduling decision. While ITA has a reduced response time due to its simplicity, it has an unacceptably high rejection ratio (~40% of tasks were rejected).

| Metrics | Non-TAS | C | N | HWP | PDTM | ITA | FC | FN | FTA | FTT |
|---|---|---|---|---|---|---|---|---|---|---|
| Resp Time (ms) | 1937 | 1267 | 965 | 872 | 905 | 507 | 987 | 893 | 642 | 728 |
| Rejection Ratio | 62.1 | 45.7 | 36.3 | 32.2 | 37.8 | 38.8 | 39.5 | 31.7 | 23.7 | 25.5 |

Table 5.9: Soft Real-Time Performance amongst Different DTAS Policies

Firstly, these results show that the non-TAS scheduling policy is significantly worse than the TAS policies. This demonstrates why DTAS is an important system level

process. The results also show that our future task aware (FTA) and future temperature trend (FTT) policies are generally superior to existing DTAS techniques in terms of core temperature, DTM trigger times, task response times and task rejection ratios, and are comparable to HWP and PDTM in terms of spatial and temporal diversity. We have not compared our algorithms to algorithms for course-grained DTM, such as [51][72]. However, our fine-grained DTAS policies could be combined with these DTM policies, to decide which core a task is migrated to after a DTM event.

## 5.8 Summary

In this chapter, a fast event driven thermal estimation method, which includes both the dynamic and leakage power models, for monitoring temperature and guiding dynamic TAS (DTAS) is proposed. The fast event driven thermal estimation is based upon a thermal map, with occasional thermal sensor-based calibration, which is updated only when a high level event occurs. To minimize the overhead, while maintaining the estimation accuracy, prebuilt look-up-tables and predefined leakage calibration parameters are used to speed up the thermal solution. Experimental results show our method is accurate, producing thermal estimations of similar quality to an existing open-source thermal simulator, while having a considerably reduced computational complexity. Based on this predictive approach, we take full advantage of a projected future thermal map to develop several heuristic policies for DTAS. We show that our proposed predictive policies are significantly better, in terms of minimizing average/peak temperature, reducing the dynamic thermal management overhead and improving other real-time features, than existing DTAS schedulers, making them highly suitable for heuristically guiding thermal aware task allocation and scheduling.

# Chapter 6
# Conclusions and Future Work

In this chapter, we summarise the contributions presented in this thesis, and discuss several unimplemented but promising directions for our future research in high level thermal-aware scheduling and management in many-core systems.

## 6.1 Contributions

The main contributions of this work have been detailed in Chapters 3 to 5. These include:

- A fast event-driven thermal estimator which uses several pre-calculated LUTs (representing the thermal response of a 1 Watt power input to a processor core or TDL) to accumulate the temperature increment for the entire multi-core chip. Compared to the traditional time-triggered thermal simulation, our proposed power event-driven approach can significantly reduce the calculation overhead and the frequency of calculation, since we only need to evaluate the temperature when an event occurs. To reduce the storage of these LUTs, we use the symmetry of a multi-core layout (TDLs) and a non-uniform time interval. The accuracy of the LUT approach is also verified by comparing with the open source thermal simulator, HotSpot, which has been validated in the literature and is widely used in academic research. We also define two necessary LUT operations (i.e. table operations and row operations) and explain how these could be used in different STAS and DTAS scenarios.
- The LUT table operations are used in an STAS scenario to test the schedulability of a real-time task set under a strict thermal constraint. Imposing a strict thermal constraint is important as excessive temperature (and temperature fluctuations) can result in computation speed degradation, aging, and unreliable system behaviour, but has not been considered as a hard constraint in other STAS literature, due to the overhead associated with

conventional thermal estimation. As a result, the schedulability analysis for hard real-time task sets in a TAS scenario has not been previously implemented. A table representing the thermal response of an individual task is essentially moved along the time axis, and can be accumulated with other task thermal response tables by aligning to the absolute time instance. This approach can efficiently generate an accurate thermal map and thus the detailed temperature transient associated with a scheduled task. This fast schedulability test for hard real-time task sets can then be used as a framework for other thermal optimisations (e.g. performance maximization and peak temperature minimization) in STAS. We then propose several performance and thermal optimizations which are tested using both practical benchmarks and synthetic task sets. The experiments show that we are able to schedule large task sets (up to 50 tasks) in reasonable time (less than 11 minutes), which is 2-3 orders of magnitude faster than using scheduling with existing thermal simulation tools. Our proposed thermal optimizations can also be used to reduce the peak temperature, thus keeping temperature below a safe threshold thereby enhancing the reliability of the real-time embedded systems.

- LUT row operations are then used in a DTAS scenario for updating the thermal map when a power event occurs. This row operation is fast enough to obtain the temperature increment between two consecutive events and is suitable for online purposes. However, these operations are only suitable for a non-temperature dependent leakage model (based on LODEs that can be linearly accumulated). To extend our fast online thermal estimator for realistic leakage power modelling, we develop an empirical calibration factor that can compensate the temperature offset in a temperature-dependent leakage model (a non-linear model) and hence convert this non-linear problem back to an approximated linear problem. This calibration factor is then used to eliminate the iterative process needed to evaluate the temperature-leakage power relationship used by other simulators. Thus, in an online (DTAS) scenario, we can still use the calibrated row operation to efficiently evaluate the thermal map, making our thermal model accurate and fast enough to guide fine (time) grained task allocation. We also proposed several heuristic policies based on the (near future) predicted thermal map. Experimental results show that our method is accurate, producing thermal estimations of similar quality to an

existing open-source thermal simulator, while providing a 3 order of magnitude improvement in terms of computational overhead. We show that our proposed predictive policies are significantly better, in terms of minimizing average/peak temperature, reducing the dynamic thermal management overhead and improving other real-time features, compared to existing DTAS schedulers, making them highly suitable for heuristically guiding thermal aware task allocation and scheduling.

In summary, we have proposed a fast event-driven LUT-based thermal estimation technique, and related thermal optimizations, for both high level STAS and DTAS.

## 6.2 Future Work

There are still several important and interesting aspects to our research which have not yet been implemented. On-going work includes:

- Our multi-core ARM simulator needs to be continuously developed and improved.
    a. In the current version of the multi-ARM simulator, we have assumed that the shared L2 cache and the main memory are combined into a single unified memory which can accommodate any application's code section. This assumption and implementation does not reflect the cache hierarchy in a real processor, and as such, it is necessary to implement the detailed L2 cache behaviour and its communication to main memory.
    b. The current version of the multi-ARM power estimator uses the original power model for the uniprocessor implementation. This power model is accurate enough to evaluate the core's power consumption, but it is unable to model the power consumed by the inter-core communications (e.g. SCU and bus). Thus, we need to add an updated SCU and bus power model into the simulator.
    c. The current version of the multi-ARM simulator does not support supervisor mode, and thus does not support an OS running on it.

Implementing supervisor mode and the related CP15 instructions would allow multi-threaded programs to run under OS control. This would provide a more realistic simulation environment.

- A counter-based power profiler is needed for offline validation and online calibration purposes. We have initially implemented a counter-based utilization profiler on an ARM Cortex-A9 dual core processor by using 6 counters to sample 18 system events (e.g. cache coherency events, execution units event, load/store queue events, cache hit and miss, etc.). However, the utilization results are not accurate enough due to counter rotating[21]. We need to adjust and improve the accuracy of this counter-based power profiler. After obtaining the accurate utilization results, we can validate the simulator-based profiler using the counter-based profiler, since similar utilization will result in similar power consumption for each functional unit.

There are a number of other promising projects which would naturally follow on from our work on a fast event-driven LUT based thermal estimator for TAS. These include:

- It would be useful to profile the power consumption and the temperature transient directly on the real many-core processor to build an accurate power/thermal model and validate both simulator-based and counter-based power estimation. To calibrate the actual chip thermal characteristics, we would execute (relatively) constant power tasks on individual cores till thermal steady state and use the built-in DTS to determine the steady state (time averaged) temperature on all cores. By using tasks of different power and allocating to different cores, it will be possible to determine the thermal characteristics of the chip. We would then implement our proposed framework for thermal management by modifying the scheduler in the open source Linux OS. We would then run a series of benchmarks and capture the thermal profile using a thermal imaging camera. This would then be used to further modify the characteristics of the thermal RC network used in the power-thermal estimator.
- In our future STAS and DTAS research, we need to consider the inter-task communications in the task set model (particularly for embedded systems),

[21] Because the number of build-in counters is less than the number of profiled events we need to rotate the counter to sample the different events in a fixed period.

and also take the power consumption and heat dissipation of the inter-core communication into account. We would need to determine how the NoC [111] heat dissipation affects the adjacent cores in a many-core architecture and the communications channels among cores (if the thermal management is enabled on NoC) as this will affect the scheduling results.

# Reference

[1] http://www.itrs.net/, “International Technology Roadmap for Semiconductors 2011 Edition (ITRS 2011)”, last accessed in Jun 2012.

[2] Anantha Chandrakasan and Robert W. Brodersen, “Low Power CMOS Design”, published by Wiley-IEEE Press, Feb 1998.

[3] S. M. Sze, “Modern Semiconductor Device Physics”, published by John Wiley and Sons, 1998.

[4] Duncan A. Grant and John Gowar, “Power MOSFETs: Theory and Applications”, published by John Wiley and Sons, 1989.

[5] Abhishek Kumar, “Leakage Current Controlling Mechanism Using High-K Dielectric + Metal Gate”, International Journal of Information Technology and Knowledge Management, Vol. 5, No. 1, pp. 191-194, Jan-Jun 2012.

[6] "http://www.dailytech.com/Intels+3D+Transistors+Boost+Performance+Lower+Power+Consumption+for+Ivy+Bridge/article21547.htm", Brandon Hill, “Intel’s 3D Transistors Boost Performance, Lower Power Consumption for Ivy Bridge”.

[7] Yongseok Cheon, Pei-Hsin Ho, Andrew B. Kahng, Sherief Reda and Qinke Wang, “Power-aware placement”, in proceedings of the 42nd Design Automation Conference (DAC'05), Jul, 2005.

[8] Siozios, Kostas and Soudris, Dimitrios, “A Power-Aware Placement and Routing Algorithm Targeting 3D FPGAs”, IET Journal of Low Power Electronics, Vol. 4, No. 3, pp. 275-289, Dec 2008.

[9] Prasun Ghosal and Tuhina Samanta, “Thermal-Aware Placement of Standard Cells and Gate Arrays: Studies and Observations”, in proceeding of IEEE Computer Society Annual Symposium on VLSI, 2008.

[10] Venkatesh Arunachalam and Wayne Burleson, “Low-Power Clock Distribution in a Multilayer Core 3D Microprocessor”, in proceedings of the 18th ACM Great Lakes symposium on VLSI (GLSVLSI'08), 2008.

[11] Deepak Sekar, Calvin King, Bing Dang, Todd Spencer, Hiren Thacker, Paul Joseph, Muhannad Bakir and James Meindl, “A 3D-IC Technology with Integrated Microchannel Cooling”, in proceedings of Interconnect Technology Conference (IITC'08), 2008.

[12] Anantha P. Chandrakasan, Miodrag Potkonjak, Renu Mehra, Jan Rabaey, and Robert W. Brodersen, “Optimizing Power Using Transformations”, IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14, No. 1, Jan 1995.

[13] Jose Monteiro, Srinivas Devadas and Abhijit Ghosh, “Retiming Sequential Circuits for Low Power”, in proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD'93), 1993.

[14] Ranganath Gopalan, Chandramouli Gopalakrishnan and Srinivas Katkoori, “Leakage Power Driven Behavioral Synthesis of Pipelined Datapaths”, in proceedings of the IEEE Computer Society Annual Symposium on VLSI, 2005.

[15] Jose Monteiro, Srinivas Devadas, Pranav Ashar and Ashutosh Mauskar ,“Scheduling Techniques to Enable Power Management”, in proceedings of the 33rd Design Automation Conference (DAC'96), Jun 1996.

[16] Hao Li, Srinivas Katkoori, and Wai-Kei Mak, “Power Minimization Algorithms for LUT-Based FPGA Technology Mapping”, ACM Transactions on Design Automation of Electronic Systems, Vol. 9, No. 1, pp. 33–51, Jan 2004.

[17] Anand Raghunathan, Sujit Dey and Niraj K. Jha, “Glitch Analysis and Reduction in Register Transfer Level Power Optimization”, in proceedings of the 33rd Design Automation Conference (DAC'96), Jun 1996.

[18] Julien Lamoureux, Guy G. Lemieux, Steven J.E. Wilton, ”GlitchLess: An Active Glitch Minimization Technique for FPGAs”, in proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'07), Feb 2007.

[19] Abhijit Ghosh, Srinivas Devadas, Kurt Keutzer and Jacob White, “Estimation of Average Switching Activity in Combinational and Sequential Circuits”, in proceedings of the 29th Design Automation Conference (DAC'92), Jun 1992.

[20] David Brooks and Margaret Martonosi, “Dynamic Thermal Management for High-Performance Microprocessors”, in proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA'01), Jan 2001.

[21] James Donald and Margaret Martonosi, “Techniques for Multicore Thermal Management: Classification and New Exploration”, in proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06), 2006.

[22] http://cccp.eecs.umich.edu/research/poweraware.php, “Power Aware Compilation”, last accessed in Jun 2012.

[23] M. D. Galanis, A. Milidonis, G. Theodoridis, D. Soudris and C. E. Goutis, "A Partitioning Methodology for Accelerating Applications in Hybrid Reconfigurable Platforms", in proceedings of the 8th Design Automation and Test in Europe (DATE'05), 2005.

[24] Le Cai and Yung-Hsiang Lu, "Energy Management Using Buffer Memory for Streaming Data", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 24, No. 2, Feb 2005.

[25] Ravishankar Rao and Sarma Vrudhula, "Energy Optimal Speed Control of a Producer–Consumer Device Pair", ACM Transactions on Embedded Computing Systems, Vol. 6, No. 4, Article 30, Sep 2007.

[26] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Waler Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe and Anant Agarwal, "Baring It All to Software: Raw Machines", MIT Laboratory For Computer Science TR-709, Mar 1997.

[27] G. Paci, P. Marchal, F. Poletti and L. Benini, "Exploring Temperature-Aware Design in Low-Power MPSoCs", in proceedings of the 9th Design Automation and Test in Europe (DATE'06), May 2006.

[28] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin, "Multiprocessor System-on-Chip (MPSoC) Technology", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 10, Oct 2008.

[29] Ayse Kivilcim Coskun, Tajana Simunic Rosing, Keith Whisnant, "Temperature Aware Task Scheduling in MPSoCs", in proceedings of the 10th Design Automation and Test in Europe (DATE'07), Apr 2007.

[30] http://techresearch.intel.com/ProjectDetails.aspx?Id=1, "Intel Single-Chip Cloud Computer (SCC)", last accessed in Aug 2011.

[31] Matteo Monchiero, Ramon Canal, Antonio Gonzalez, "Power/Performance/Thermal Design-Space Exploration for Multicore Architectures", IEEE Transactions on Parallel and Distributed Systems, Vol. 19, No. 5, May 2008.

[32] David Brooks, Vivek Tiwari and Margaret Martonosi, "Wattch: A Framework for Architecture-Level Power Analysis and Optimizations", in proceedings of the 27th International Symposium on Computer Architecture (ISCA'00), 2000.

[33] Premkishore Shivakumar and Norman P. Jouppi," CACTI 3.0: An Integrated Cache Timing, Power, and Area Model", WRL Research Report 2001/2, Aug 2001.

[34] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh and Sharad Malik, "Orion: A Power-Performance Simulation for Interconnection Networks", in proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture (MICRO'02), 2002.

[35] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan and David Tarjan, "Temperature-Aware Microarchitecture" , in Proc. of the 30th Annual International Symposium on Computer Architecture (ISCA'03), 2003.

[36] Mircea R. Stan, Kevin Skadron, Marco Barcella, Wei Huang, Karthik Sankaranarayanan and Sivakumar Velusamy, "HotSpot: a Dynamic Compact Thermal Model at the Processor-Architecture Level", Microelectronics Journal, Vol. 34, Issue 12, pp. 1153-1165, Dec 2003.

[37] Wojciech Giziewicz, "Brief Introduction to HSPICE Simulation (internal document for the coursework)", http://web.mit.edu/6.012/www/SPICEtutorial.pdf, last accessed in Jun 2012.

[38] www.qemu.org/, "QEMU: Open Source Processor Emulator", last accessed in Jun 2012.

[39] http://www.cs.berkeley.edu/~kubitron/cs252/simtutorial.html, "Simics tutorial (internal document for the coursework)", last accessed in Jun 2012.

[40] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill and David A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset", published in Computer Architecture News (CAN), Sep 2005.

[41] Doug Burger, Todd M. Austin, and Steve Bennett, "Evaluating Future Microprocessors: the SimpleScalar Tool Set", Internal Technical Report 1308, Computer Sciences Department, University of Wisconsin.

[42] David Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma and M. G. Rosenfield, "New methodology for early-stage microarchitecture-level power-performance analysis of microprocessors", IBM Journal of Research and Development, Vol. 47, No. 5/6, September 2003.

[43] David Brooks, V. Tiwari, and M. Martonosi, “Wattch: A framework for architecture-level power analysis and optimizations,” in Proc. of the 27th Annual International Symposium on Computer Architecture (ISCA'00), 2000.

[44] Noel Eisley, Vassos Soteriou, LiShiuan Peh, “High-Level Power Analysis for Multi-Core Chips”, in proceedings of 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06), Oct 2006.

[45] Russ Joseph and Margaret Martonosi, “Run-Time Power Estimation in High Performance Microprocessors”, in proceedings of the 2001 ACM International Symposium on Low Power Electronics and Design (ISLPED '01), 2001.

[46] http://oprofile.sourceforge.net/docs/intel-corei7-events.php, “Intel Core i7 (Nehalem) Events”, last accessed in Jun 2012.

[47] http://www.arm.com, “ARM Cortex-A9 Technical Reference Manual”, last accessed in Jun 2012.

[48] Canturk Isci and Margaret Martonosi, “Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data”, in proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03), 2003.

[49] Kyeong-Jae Lee and Kevin Skadron, “Using Performance Counters for Runtime Temperature Sensing in High-Performance Processors”, in proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), 2005.

[50] Min Bao, Alexandru Andrei, Petru Eles, and Zebo Peng, “On-line Thermal Aware Dynamic Voltage Scaling for Energy Optimization with Frequency/Temperature Dependency Consideration”, in proceedings of the 46th Design Automation Conference (DAC'09), 2009.

[51] Inchoon Yeo and Eun Jung Kim, “Temperature-Aware Scheduler based on Thermal Behavior Grouping in Multicore Systems”, in proceedings of the 12th Design Automation and Test in Europe (DATE’09), 2009.

[52] Yongpan Liu, Robert P. Dickt, Li Shang and Huazhong Yang, “Accurate Temperature-Dependent Integrated Circuit Leakage Power Estimation is Easy”, in proceeding of the 10th Design Automation and Test in Europe(DATE’07), Apr 2007.

[53] Jung Hwan Choi, Aditya Bansal, Mesut Meterelliyoz, Jayathi Murthy and Kaushik Roy, “Leakage Power Dependent Temperature Estimation to Predict

Thermal Runaway in FinFET Circuits", in proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD'06), 2006.

[54] Alexandru Andrei, Petru Eles, Zebo Peng, Marcus T. Schmitz and Bashir M. Al-Hashimi, "Energy Optimization of Multiprocessor Systems on Chip by Voltage Selection", IEEE Transactions on Very Larger Scale Integration (VLSI) Systems, Vol. 15, No. 3. pp:262--275, 2007.

[55] Min Bao, Alexandru Andrei, Petru Eles, Zebo Peng, "Temperature-aware voltage selection for energy optimization", in proceedings of the 11th Design Automation and Test (DATE'08), Apr 2008.

[56] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan, "Temperature-Aware Microarchitecture: Extended Discussion and Results", Extended Internal Report of [35], 2003.

[57] Sushu Zhang and Karam S. Chatha," Approximation Algorithm for the Temperature-aware Scheduling Problem", in proceeding of the 10th Design Automation and Test in Europe(DATE'07), Apr 2007.

[58] Sushu Zhang and Karam S. Chatha, "Thermal Aware Task Sequencing on Embedded Processors", in proceedings of the 47th Design Automation Conference (DAC'10), 2010.

[59] Srinivasan Murali, Almir Mutapcic, David Atienza, Rajesh Gupta, Stephen Boyd and Giovanni De Micheli, "Temperature-Aware Processor Frequency Assignment for MPSoCs using Convex Optimization," in proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07), 2007.

[60] Ashutosh Dhodapkar, Chee How Lim, George Cai, and W. Robert Daasch, "TEM$^2$P$^2$EST: A Thermal Enabled Multi-model Power/Performance ESTimator", in proceedings of the First International Workshop on Power-Aware Computer Systems, Nov, 2000.

[61] Xi Chen, Robert P. Dick and Li Shang, "Properties of and Improvements to Time-Domain Dynamic Thermal Analysis Algorithms", in proceedings of the 13th Design Automation and Test (DATE'10), 2010.

[62] Pu Liu, Hang Li, Lingling Jin, Wei Wu, Sheldon X.-D. Tan, and Jun Yang, "Fast thermal simulation for runtime temperature tracking and management", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 25, No. 12, Dec 2006.

[63] http://www1.ansys.com/customer/content/documentation/121/ans_the.pdf, "Thermal Analysis Guide of ANSYS", last accessed in Jun 2012.

[64] http://www.comsol.com/, "COSMOL: Multiphysics Modeling and Simulation Software", last accessed in Jun 2012.

[65] Kyriakos Stavrou and Pedro Trancoso, "TSIC: Thermal Scheduling Simulator for Chip Multiprocessors", in proceedings of the 10th Panhellenic Conference on Informatics (PCI'05), vol. 3746 of LNCS, pp. 589-599, Nov 2005.

[66] Yong Zhan and Sachin S. Sapatnekar, "Fast Computation of the Temperature Distribution in VLSI Chips Using the Discrete Cosine Transform and Table Look-up", in proceedings of the 2005 Asia and South Pacific Design Automation Conference (ASPDAC'05), Jan 2005.

[67] B. Wang and P. Maxumder, "Fast Thermal Analysis for VLSl Circuits via Semi-analytical Green's Function in Multi-layer Materials," in proceedings of 2004 IEEE International Symposium on Circuits and Systems (ISCAS'04), May 2004.

[68] Pierre Michaud Yiannakis Sazeides, "ATMI: Analytical Model of Temperature in Microprocessors", in proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, Jun 2007.

[69] Amit Kumar, Li Shang, Li-Shiuan Peh, and Niraj K. Jha," System-Level Dynamic Thermal Management for High-Performance Microprocessors", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 1, Jan 2008.

[70] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Kenny C. Gross, "Proactive Temperature Balancing for Low Cost Thermal Management in MPSoCs," in proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD'08), 2008.

[71] Inchoon Yeo, Chih Chun Liu, and Eun Jung Kim, "Predictive Dynamic Thermal Management for Multicore Systems," in proceedings of the 45th Design Automation Conference (DAC'08), Jun 2008.

[72] Fabrizio Mulas, Michele Pittau, Luca Benini, Andrea Acquaviva, and David Atienza, "Thermal Balancing Policy for Streaming Computing on Multiprocessor Architectures," in proceedings of the 11th Design Automation and Test (DATE'08), 2008.

[73] Yufu Zhang and Ankur Srivastava, “Accurate Temperature Estimation Using Noisy Thermal Sensor,” in proceedings of the 46th Design Automation Conference (DAC'09), Jun 2009.

[74] Efraim Rotem, Jim Hermerding, Cohen Aviad and Cain Harel, “Temperature Measurement in the Intel Core Duo Processor”, http://arxiv.org/ftp/arxiv/papers/0709/0709.1861.pdf, last accessed in Jun 2012.

[75] Shervin Sharifi and Tajana S. Rosing, “Accurate Direct and Indirect On-Chip Temperature Sensing for Efficient Dynamic Thermal Management”, IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol. 29, No. 10, Oct 2010.

[76] Rajarshi Mukherjee and Seda Ogrenci Memik, “Systematic Temperature Sensor Allocation and Placement for Microprocessors”, in proceedings of the 43th Design Automation Conference (DAC'06), Jul 2006.

[77] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli, “A Survey of Design Techniques for System-Level Dynamic Power Management”, IEEE Transactions on Very Large Scale Integration Systems, Vol. 8, No. 3, Jun 2000.

[78] http://researchweb.watson.ibm.com/MET/Toolset/Turandot/turandot.html, “Turandot”, last accessed in Jun 2012.

[79] Pedro Chaparro, Jose Gonza lez, Grigorios Magklis, Qiong Cai, and Antonio Gonza lez, “Understanding the Thermal Implications of Multicore Architectures”, IEEE Transactions on Parallel and Distributed Systems, Vol. 18, No. 8, August 2007.

[80] Yingmin Li, David Brooks, Zhigang Hu, Kevin Skadron,”Performance, Energy, and Thermal Considerations for SMT and CMP”, in proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA’05), 2005.

[81] Sandy Irani and Kirk R. Pruhs, “Algorithmic Problems in Power Management”, ACM SIGACT News, Vol. 36, Issue 2, pp. 63-76, Jun 2005.

[82] N. Bansal and Kirk R. Pruhs, “Speed Scaling to Manage Temperature”, in proceeding of 2005 Symposium on Theoretical Aspects of Computer Science, 2005.

[83] F. Yao, A. Demers, and S. Shenker. “A Scheduling Model for Reduced CPU Energy”, in proceedings of 1995 IEEE Syposium on Foundations of Computer Science, pp. 374, 1995.

[84] Han-Saen Yun and Jihong Kim, “On Energy-Optimal Voltage Scheduling for Fixed Priority Hard Real-Time Systems”, ACM Transactions on Embedded Computing Systems, Vol. 2, No. 3, pp. 393-430, 2003.

[85] Woo-Cheol Kwon and Taewhan Kim, “Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors”, in proceedings of the 40th Design Automation Conference (DAC’03), Jul 2003.

[86] Minming Li, Becky Jie Liu and Frances F. Yao. “Min-Energy Voltage Allocation for Tree-Structured Tasks”, in proceedings of International Computing and Combinatorics Conference, 2005.

[87] L. Niu, “System-Level Energy-Efficient Scheduling for Hard Real-time Embedded Systems,” in proceedings of the 14th Design Automation and Test (DATE’11), 2011.

[88] Francesco Paterna, Andrea Acquaviva, Alberto Caprara, Francesco Papariello, Giuseppe Desoli, Luca Benini, “An Efficient On-line Task Allocation Algorithm for QoS and Energy Efficiency in Multicore Multimedia Platforms,” in proceedings of the 14th Design Automation and Test (DATE’11), 2011.

[89] Jason Cong and K. Gururaj, “Energy Efficient Multiprocessor Task Scheduling under Input-Dependent Variation”, in proceedings of the 12th Design Automation and Test (DATE’09), 2009.

[90] Yuan Xie and Wei-Lun Hung, “Temperature-Aware Task Allocation and Scheduling for Embedded Multiprocessor Systems-on-Chip (MPSoC) Design”, Journal of VLSI Signal Processing 45, pp. 177–189, 2006.

[91] Marek Chrobak, Christoph Durry, Mathilde Hurandy and Julien Robert, “Algorithms for Temperature-Aware Task Scheduling in Microprocessor Systems”, in proceedings of the 4th Algorithmic Aspects in Information and Management(AAIM’08), Jun 2008.

[92] Kyriakos Stavrou and Pedro Trancoso, “Thermal-Aware Scheduling for Future Chip Multiprocessors”, EURASIP Journal on Embedded Systems, Vol. 2007, Article 48926, 2007.

[93] Srinivasan Murali, Almir Mutapcic and David Atienza, “Temperature Control of High-Performance Multi-core Platforms Using Convex Optimization”, in proceeding of the 11th Design Automation and Test in Europe(DATE’08), Mar 2008.

[94] Thidapat Chantem, Robert P. Dick and X. Sharon Hu, "Temperature-Aware Scheduling and Assignment for Hard Real-Time Applications on MPSoCs", in proceeding of the 11th Design Automation and Test in Europe(DATE'08), Mar 2008.

[95] Ayse Kivilcim Coskun, Tajana S. Rosing, Keith A. Whisnant, and Kenny C. Gross, "Static and Dynamic Temperature-Aware Scheduling for Multiprocessor SoCs," IEEE Transactions on Very Large Scale Integration Systems, Vol. 16, No. 9, Sep 2008.

[96] https://computing.llnl.gov/tutorials/pthreads/, "POSIX Threads Programming", last accessed in Jun 2012.

[97] http://en.wikipedia.org/wiki/Message_Passing_Interface, "Message Passing Interface", last accessed in Jun 2012.

[98] http://en.wikipedia.org/wiki/Cache_coherence, "Cache Coherency", last accessed in Jun 2012.

[99] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0407f/BABEBFBH.html, "ARM Cortex-A9 MPCore Technical Reference Manual", last accessed in Jun 2012.

[100] www-micro.deis.unibo.it/~magagni/amba99.pdf, "AMBA Specification", last accessed in Jun 2012.

[101] http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf, "ARM and Thumb-2 Instruction Set Quick Reference Card", last accessed in Jun 2012.

[102] Iven Ukhoh, Min Bao, Petru Eles and Zebo Peng, "Steady-State Dynamic Temperature Analysis and Reliability Optimization for Embedded Multiprocessor Systems", in proceedings of the 49th Design Automation Conference (DAC'12), Jun 2012.

[103] Vijaykrishnan Narayannan and Yuan Xie, "Reliability Concerns in Embedded System Designs", Embedded Computing Magazine, pp. 118—120, Jan 2006.

[104] Vinay Hanumaiah, Ravishankar Rao, Sarma Vrudhula, and Karam S. Chatha, "Throughput Optimal Task Allocation under Thermal Constraints for Multi-Core Processors," in proceedings of the 46th Design Automation Conference (DAC'09), Jul 2009.

[105] Chiao-Ling Lung, Yi-Lun Ho, Ding-Ming Kwai and Shih-Chieh Chang, "Thermal-Aware On-Line Task Allocation for 3D Multi-Core Processor

Throughput Optimization," in proceeding of the 14th Design Automation and Test in Europe(DATE'11), Apr 2011.

[106] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun and Marianne De Michiel, "PapaBench : A Free Real-Time Benchmark", in proceedings of Workshop on Worst-Case Execution Time (WCET'06), 2006.

[107] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge and Richard B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", in proceedings of IEEE International Workshop of the Workload Characterization, 2001.

[108] http://www.cprover.org/goto-cc/examples/snu.html, "SNU Real-time Benchmarks", latest accessed April 2012.

[109] A. Sridhar, A. Vincenzi, M. Ruggiero, T. Brunschwiler, David Atienza, 3D-ICE: Fast compact transient thermal modeling for 3D-ICs with inter-tier liquid cooling, in proceedings of the 2010 International Conference on Computer-Aided Design (ICCAD'10), Nov 2010.

[110] http://lava.cs.virginia.edu/HotSpot/index.htm, "Hotspot v5.0," last accessed Jun 2012.

[111] Tobias Bjerregaard and Shankar Mahadevan, "A Survey of Research and Practices of Network-on-Chip", ACM Computing Surveys, Vol. 38, Article 1, Mar 2006.

[112] http://linux.die.net/man/8/adjtimex, "Linux time variable man page", last accessed in 2012, Jun.

[113] http://www.intel.sg/content/www/xa/en/processors/xeon/xeon-processor-e7-family.html?wapkw=xeon, "Intel Xeon", last accessed in 2012, Jun.

[114] http://www.m5sim.org/Main_Page, "GEM5", last accessed in 2012, Jun.

# Appendix A

# Multi-ARM Simulator for Power Profiling

As discussed in Section 1, power profiling can be carried out either offline or online using a simulator and the on-chip performance counters, respectively. However, after investigating the available literature and examining the code sources available on the Internet, we found that:

- No cycle-accurate power profiler is available for current high performance multi-core ARM architecture (e.g. ARM Cortex-A9). GEMS [40] is a popular and widely-used multi-core simulator but its core only simulates a simple in-order pipeline that is out of date for most modern high performance microprocessors. GEM5 [114] does not have the power model for ARM architecture. Additionally, the inter-core communications in GEM5 only considers network-on-chip (NoC) or bus architectures, based on a pure message passing interface (MPI) [97], without supporting the POSIX multi-threading library [96] (pthread [22]) which is mainly used in the current mainstream multi-threaded applications. The following table shows the strengths and weaknesses for both modes of inter-process communication.

| | **MPI** | **pthread** |
|---|---|---|
| Communication Architectures | NoC or Bus | Bus |
| Memory Model | Distributed Memory | Shared Memory |
| Programming Difficulty | Hard | Easy |
| Bandwidth of Inter-Core Communication | Low (Data Package) | High (Memory Transaction) |
| Number of Simultaneous Inter-Core Communication | High | Low |
| Latency | High | Low |
| Need for Cache Coherency | No | Yes |
| Scalability | High | Low |

Table A.1: Comparison between MPI and pthread

As seen from the comparison between the two inter-core mechanisms, both have their own advantages, and we can conclude that MPI is more suitable for future many-core architecture with full optimization of message passing

[22] POSIX thread library is widely used for multi-threading on the shared memory architecture.

techniques in the compiler (like network programming, e.g. the sending and receiving of messages between the cores), while pthread is more suitable for multi-core architectures, with traditional compilers and a much simpler programming model (e.g. using shared memory as a global database in a multi-threaded program, and using join, lock and mutex to achieve the synchronization and the mutual exclusion).

- No counter-based power estimator for the ARM architecture is implemented so far. So the validation of the utilization estimation for the ARM architecture is not able to be carried out and thus the correctness of the simulator-based offline power profiling cannot be determined.

In this appendix, we introduce an extension of SimpleScalar [41] that supports multi-threaded programming running on a shared-memory scheme on a multi-ARM architecture that we used for power profiling in our TAS research. While much of the multi-ARM simulator has been completed, there is still some work that needs to be done, in terms of the simulator-based and counter-based power estimation. These uncompleted extensions are listed in the future work chapter of this thesis.

## A.1 Multi-core ARM Simulator

The SimpleScalar simulator [41] is able to simulate real programs running on modern processors and systems, and provides a realistic implementation which is close to that of current commercial processors that support ILP, via the out-of-order execution of hardware dynamic instruction scheduling provided by SuperScalar. The critical issue of SimpleScalar (for any architecture, e.g. Alpha, ARM and PowerPC) is that it only exactly simulates a uniprocessor without any multi-thread (multi-process) programming allowed. In other words, a multi-threaded application cannot be executed on SimpleScalar. In this section, we extend the original SimpleScalar to support multi-core simulation and multi-thread applications.

To provide support for multi-core and multi-threaded applications, we must add the following necessary code blocks into the simulator:

- We need to duplicate the uniprocessor in the original SimpleScalar to produce a multi-core architecture. However, some parts (e.g. the Load/Store queue) of the uniprocessor need to be modified to satisfy the logical correctness of the inter-core data dependency.
- We need to add a snoop control unit (SCU) to the inter-core communication bus to implement cache coherency among cores.
- We need to implement a number of critical system calls used by the multi-thread library (pthread), such as thread creation (the clone system call), thread pending for event (the wait system call), and so on.

Before introducing our extension, we need to look at the basic structure of the uniprocessor simulation in the original SimpleScalar. That will help to understand the differences between the architectures in the uniprocessor and multi-core scenarios.

### A.1.1 Basic Simulation Procedure in SimpleScalar-ARM

SimpleScalar is widely used in academia to provide the cycle-accurate information of a uniprocessor at the micro-architecture level. It supports several complicated features in modern architectures for high performance computing, including: out-of-order execution provided by SuperScalar, instruction pre-fetch units, branch predictor and speculative execution.

In a uniprocessor, several critical functional units are simulated:

- Register File: This simulates the internal register set as an array REGS. The system registers that support the supervision mode needed by modern OSs (e.g. CP15 and CPSR register set in ARM processor) is not simulated as SimpleScalar is not a behavioural level simulator that allows an OS to run on it.
- Instruction fetch unit: This is the basic component that fetches the instruction from the L1 cache. The instruction queue used for pre-fetch is implemented, and the queue's length can be customized by the user (the default setting is 4). This pre-fetch logic is affected by the branch predictor.

- Branch predictor: Before the branch conditions are solved, additional instructions are predictively fetched according to historical results. SimpleScalar implements the predictor in two ways: a bimodal predictor and a two-level adaptive predictor (The default setting is the bimodal predictor).
- Reservation Station (Renaming Unit): This is used to assign the virtual register name (or dependent execution unit name) to resolve and encode the data dependency among the instructions. This encoding method is important and necessary for SuperScalar to dynamically schedule the instructions on multiple execution units and thus achieve instruction-level parallelism (ILP).
- Instruction dispatch and issue unit: The renamed instructions are decoded and issued to the corresponding execution unit in this stage. It determines and generates the control signal according to the identification of operands and the operator of each instruction. It checks whether the execution units are idle or busy and allocates the instructions to the idle ones. The issue width (the number of instructions to be issued at the same cycle) can be customized.
- Execution unit: SimpleScalar simulates the 5 classes of resources: integer adder and logic operator (bit operator, logic compare and shift operator), integer multiplier/divider, floating point adder, floating point multiplier/divider, memory ports (the Load/Store queues including the stack operator).
- Load/Store queue: Any memory access operations are passed to this queue for further L1 cache, or other level, memory accesses. For the uniprocessor, the instructions in the queue can be reordered or optimized according to their memory address dependency (e.g. if a memory address read appears in a prior write instruction in the queue, it can directly return the value to be written to the read instruction without needing to access the L1 cache). However, for multiprocessor systems, the memory value can be affected by another cores' access. We will discuss this later.
- L1, L2 cache and Memory: SimpleScalar has a self-customized multi-layer cache structure. The size, instruction or data used or shared between data and instruction can be set on the L1 and L2 cache. Memory is assumed to be big enough to accommodate any applications without considering a virtual

memory implementation on external storage (e.g. a hard disk). The cache simulation can be integrated with other cache simulators, e.g. CACTI.

- Memory management unit (MMU): This unit is used to translate the virtual address into a physical address in memory space. SimpleScalar implements a 2-level address translation. All the addresses in the cache lines are expressed as the physical pattern. SimpleScalar does not support multi-threading (context switching), and thus the page table base address swap used in context switching is not simulated.
- Translation look-aside buffer (TLB): This unit is used to accelerate the memory address translation by recording the latest history of a memory translation. The memory translation should firstly compare the addresses in the TLB before it actually carries out the translation in the MMU. The TLB should be flushed when context switching occurs.
- Instruction write-back and commit unit: This is to write back the result to the register file and solve the dependency among the instructions and predictor conditions. If a branch prediction is failed, it needs to roll back and recover to the correct execution path and notify the instruction fetch unit to flush. It also notifies the renaming and issuing unit to put the pending instruction into ready-to-run status. It releases the execution units used by the completed instructions and marks them as idle. The commit width can also be customized.

The above components are the main components simulated by SimpleScalar. As well as these components, SimpleScalar uses a number of data structures to support the simulation.

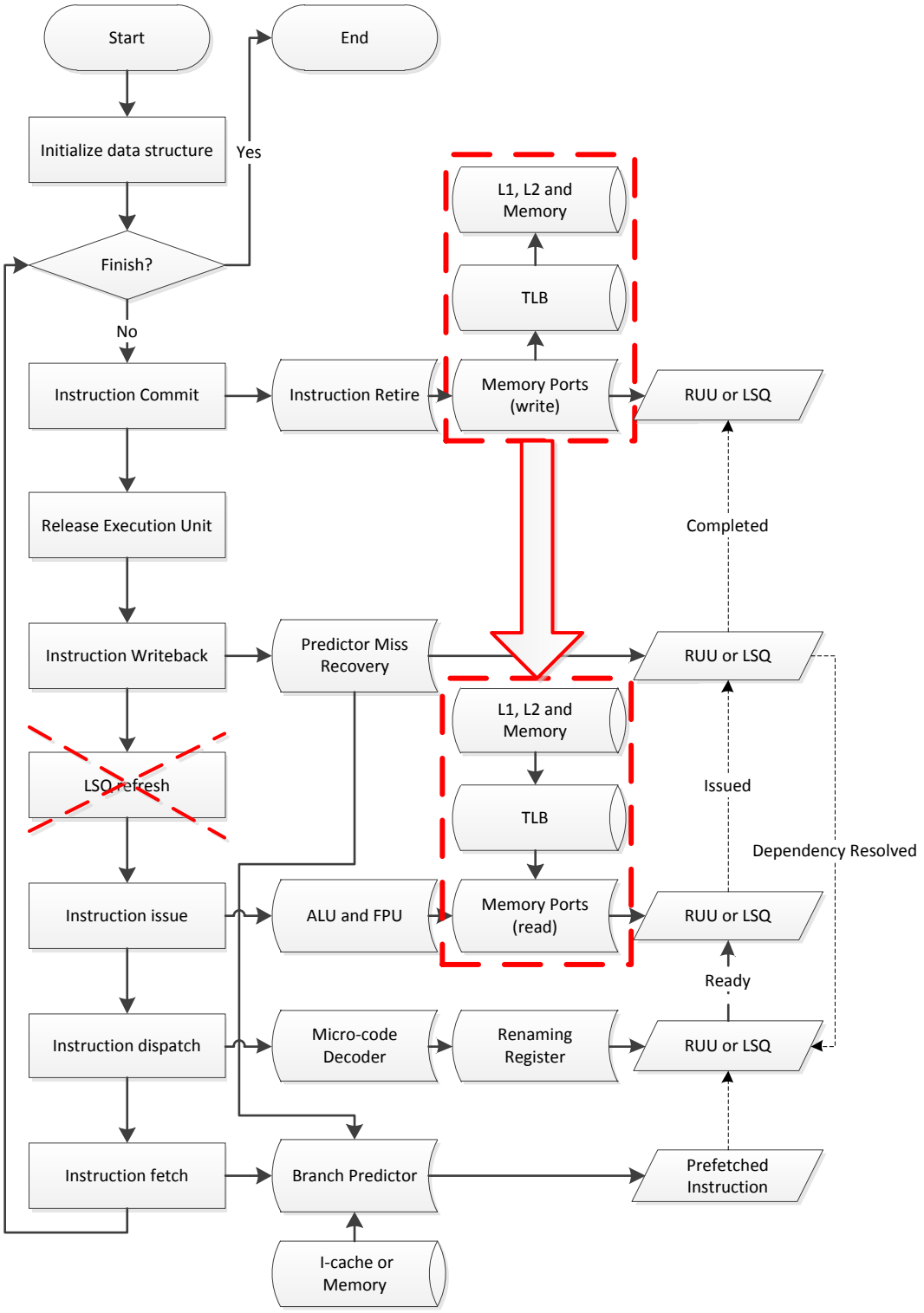



Figure A.1: The flow chart of SimpleScalar

Two data structures are used for tracking instruction status, interdependency, execution unit category, etc. after decoding. All the instructions in ARM can be classified into 3 categories based on their operands (source-to-destination): 1) register-to-register; 2) register-to-memory; 3) memory-to-register. The first class of

instructions are abstracted to the RUU_station (referred to as RUU in subsequent sections) data structure, while the second and third classes (the memory instructions) are abstracted to the LSQ_station data structure (referred to as LSQ in subsequent sections). However, in SimpleScalar, the basic integer and floating point execution units are only simulated for register-to-register style instructions, while memory ports support the register-to-memory or memory-to-register style. As a result, any memory access instruction or multiple memory access instruction (LDR and STR instructions[23] can support a set of registers reading\writing from\to a block of memory) can be converted into several equivalent single memory access instructions in the decoding stage of the dispatch unit. For instance, "*ADD mem, R1, R2;*" can be converted into "*ADD Rt, R1, R2; STR mem, Rt;*". This conversion is referred to as micro-code decoding. Therefore, each micro-code instruction is purely atomic for an execution unit and can be assigned to a RUU (register instruction) or a LSQ (memory instruction) to resolve the dependency. There are several linked lists that are used to link RUU and LSO together to track the instruction state (e.g. ready to be issued since all inputs are available or completed after execution to notify the pending instruction).

After introducing the basic components and data structure in SimpleScalar, we detail how the simulator works. Figure A.1 shows the flow chart in each simulated cycle. The main loop appears in the left side of Figure A.1. To facilitate resolving the dependency among the instructions, the simulation flow is reversed against the normal order of the pipeline in the processor (the completed instruction in the write-back stage can be to notify the pending instructions in the dispatch stage). Therefore, we introduce each sub-procedure in this reversed order.

- Instruction commit: The completed RUU or LSQ micro-code instruction should be retired[24] in this stage. The corresponding elements in RUU or LSQ should be deleted from the station array (a global RUU and LSQ array is used to track which instruction is being processed in current cycle). The completed write memory access[25] instruction (store instruction) denoted by LSQ should

[23] LDR and STR instructions are the load and store instructions in ARM instruction set.
[24] The commit width decides the number of instructions that are retired in a single cycle.
[25] It is note that in execution stage, a write memory instruction is not really write to the memory, and it only submit the write request to the load/store queue (but mark as completed). The true writing happens in commit stage to reduce the number of writing to same address (only commit the lasted writing value to one address).

actually commit the data in load/store queue to the L1 cache. It should be noted that for the uniprocessor, this late store instruction commit is feasible because the stored value needed by other instructions can be directly sent to these instructions before sending it to the memory. However, in the multi-core case, the stored value can be changed or rewritten by store instructions from other cores. As such, a store instruction in the multi-core case cannot be implemented as a two stage (late commit) process and must be atomically completed and synchronized by cache coherency before the next read.

- Release execution units: Each execution unit's busy cycle should be self-subtracted by 1. If busy cycle reaches 0, it means the execution unit is idle.
- Instruction writeback: In this stage, the simulator searches the issued instructions in the event queue (the event queue stores the RUU and LSQ information in completion order form according to the number of cycles needed by an instruction, e.g. ADD needs 1 clock cycle, and MUL needs 3 clock cycles) and changes the instruction status from "issued" to "completed". If the result of branch condition does not match the prior prediction, the correct execution path should be recovered by rolling back the index in RUU and LSQ array to where the branch occurs. The branch predictor is notified and it updates the program counter (PC) to fetch the instructions in another path. Lastly, once this instruction is complete, it should be detached from the dependency vector which links to other RUUs (or LSQs) which are dependent on this instruction. The newly resolved RUUs (or LSQs) are added into the ready queue for instruction issue in the next cycle.
- LSQ refresh: The dependency among the memory access instructions is resolved in this stage. A load instruction can directly read the value written by a prior instruction from the load/store queue. This is referred to as fast load, but is not suitable for the multi-core scenario where the value might be changed by a different core's store instruction, as explained in the instruction commit stage. Another optimization implemented in the LSQ refresh is that multiple store instructions can be combined and only latest store instruction is actually committed, since earlier values written to same address in the load/store queue would be overwritten. This is referred to as store reduction,

which again is not feasible in a multi-core scenario as the earlier value may be used by other cores.

- Instruction issue: The issued RUUs (and LSQs) in the ready queue are assigned to the idle execution units[26] (with *busy cycle* equal to 0). The number of cycles needed by every issued instruction needs to be calculated. For RUU this is fairly easy as each register-to-register operation has a predefined number of cycles for each execution unit. However, for LSQ the latency is difficult to determine due to the different level caches (either hit or miss), TLB and MMU address translation and any necessary memory access. It is noted that for a memory access instruction, only a load instruction is simulated because the store instruction is simulated in a later commit stage, as mentioned previously. After knowing the number of cycles needed, the issued instructions can be added into an event queue in the completion order, that is the earliest one should be stored at the head of the queue. For the multi-core case, the memory access latency should include the cycles needed to ensure cache coherence among the cores.
- Instruction dispatch: The pre-fetched instructions in the pre-fetched array are decoded in this stage. Some complex instructions, as mentioned earlier (e.g. register-to/from-memory instructions), are also converted into several equivalent micro-code instructions in the decoder. After generating the micro-code instruction, each micro-code instruction can be allocated to a RUU (or a LSQ) in the station array, and then the dependency among these instructions is generated by analysing the input and output registers in the renaming unit, with a dependency vector being used to link the inter-dependent instructions together. Therefore, the RUUs and LSQs in the station array appear in fetched order, while the dependency vector stores their inter-dependency that is resolved in the writeback stage. The instructions whose input registers are all resolved should be put into the ready queue.
- Instruction fetch: the instructions are fetched from the L1 cache[27] (or higher level memory if a cache miss is generated) to the pre-fetched array. The fetch simulation is similar to the memory load instruction. As the variable latency of the memory access (due to a cache or memory miss) might lead to a stall of

[26] The issue width decides the number of instructions that are issued to execution units in a single cycle.
[27] The pre-fetch width decides the number of instructions that are loaded from memory in a single cycle.

the pre-fetch unit, all cache levels as well as the main memory must be simulated. The pre-fetch unit also needs to interact with the branch predictor[28] to decide which branch the execution should follow upon meeting a branch instruction. The predictor might update the new fetch address after being notified by the writeback stage due to a missed prediction.

On a final note, this simulation procedure is carried out by the original SimpleScalar. However, we have stated that due to the concurrent access of a shared memory in the multi-core scenario, the relevant memory access simulation, in the instruction commit and issue stages with LSQ refresh, is not feasible anymore. In other words, if we just simply duplicate and integrate the original cores without any modification, the multi-core simulation will behave incorrectly at the logic level. In the following sections, we address these problems in order to fit the original uniprocessor model to the multi-core scenario.

### A.1.2 Transactional Load/Store Instructions

To solve the problems of accessing shared memory, we require the following criteria to implement the memory access instruction:

- Any store instruction should be completed atomically, i.e. a store operation should actually write the value to the cache and then complete the cache coherency[29] without being interrupted by any other operations on the same address. This is to prevent a write-after-write (a dirty write) among the cores: we need to be aware that the value in an earlier issued store instruction may be rewritten by a later store instruction to the same address. In other words, the user should see the latest written value to that address.
- Any load instruction should be completed atomically, i.e. a load operation should actually read the value in the cache, rather than read the latest value in the load/store queue. This is to prevent a read-after-write (a true dependency) among the cores: we need to be aware that the value at the address may not

[28] Branch condition is regarded as a normal micro-code instruction handled by ALU or FPU.
[29] Whether the value is directly written back to main memory is not important and decided by the cache mode: write-back or write-through.

only be changed by the core itself, but it may also be changed by the other cores.

- Any store instruction executed on one core should be visible to the other cores. This is done by cache coherency, and is discussed in the next section.

In this section, we address these issues in the uniprocessor and adapt them to the multi-core scenario.

- To implement a transactional store instruction, the store instruction cannot be completed in the previously mentioned two stage way: that is, 1) marking the store instruction status as "completed" directly in the writeback stage, and then 2) carrying out the real write operation in the commit stage. Therefore, the late commit for a store instruction should be moved and integrated into the issue stage where the number of cycles needed by a load instruction is calculated by the cache, TLB, MMU and memory simulation. This modification is shown as the red dashed boxes in Figure A.1. In other words, all the load/store instructions should be atomically issued (within a single stage) to the load/store queue (memory port) that actually accesses the L1 cache.
- To implement a transactional load, a load instruction cannot directly fetch the value directly from the load/store queue. The load instruction needs to actually access the L1 cache. Hence, the fast load in the LSQ refresh stage should be removed.
- To make all store instructions visible to the others cores, the store reduction in LSQ refresh is no longer feasible, as it deletes some store operations on the same address. Hence, those value changes do not appear in the cache and the main memory, and are invisible to the other cores. Therefore, the store reduction in the LSQ refresh stage should be removed. This means that the optimizations for memory access in the uniprocessor cannot be applied to the multi-core, and thus the entire LSQ refresh stage should be removed, as shown by the red cross in Figure A**.1**.

After modifying the uniprocessor as stated above, we can duplicate our modified uniprocessor to build a multi-core simulator. The duplication is not detailed here since it is relatively easy to duplicate all the necessary components and their corresponding data structures introduced in the earlier section, using "*core_id*" to identify the

individual cores. An inter-core communication structure is needed to connect the cores together. This is described in the next section.

### A.1.3 Inter-Core Communication

The inter-core communication discussed in this section is only applicable for the shared memory architecture in current multi-core chips, such as multi-core ARM processors. The key component of the inter-core communication infrastructure is the snoop control unit (SCU) that has the following main functions:

- Complete the cache coherency among the cores
- Complete the communication between cache and the shared memory via bus and arbitrator.

Before examining our implementation of the SCU, we introduce the basic cache models and protocols of cache coherency. In terms of the 2-level cache model in most modern ARM processor, there are two styles of hierarchy in multi-core architecture: private L2 cache (left in Figure A.2) and shared L2 Cache (right in Figure A.2).

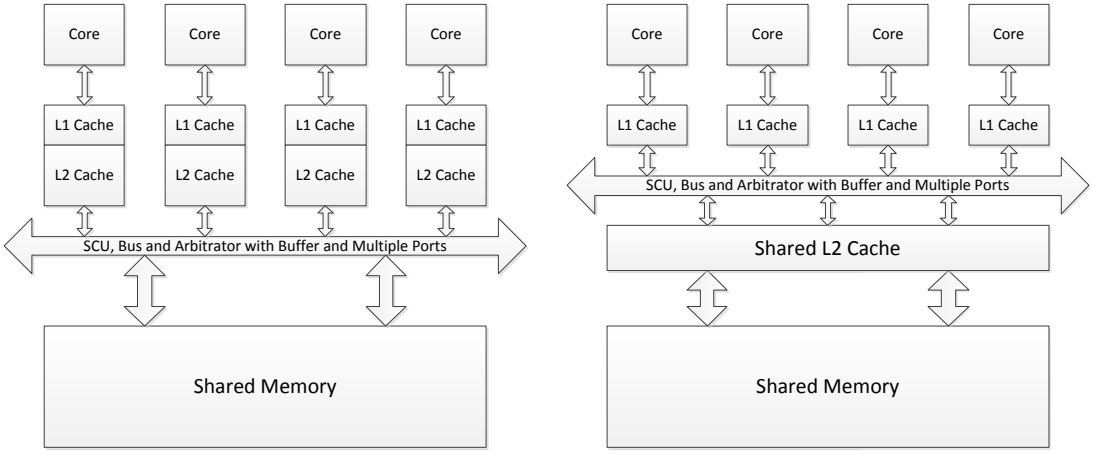



Figure A.2: SCU and Cache Hierarchy

In terms of the private L2 cache model, the L1 and L2 cache in a single core can be seen as a single cache because most modern architectures (x86 and ARM) are using the inclusive cache scheme, i.e. any cache line in L1 must exist in L2, in other words, L2 includes all the contents of L1 (but also has the cache lines that are not in L1, for example, one cache line is retired from L1 but still in L2). Therefore, the cache

coherency operations must take care of these two levels: the write operation occurring on one core's L1 not only needs to update its L2 but also needs to update the cache line's status in the other core's L1 and L2 caches. In terms of the shared L2 cache model, one can assume the shared L2 cache and main memory can be combined as a unified memory. This makes the L2 cache and the main memory the same as the normal cache scheme in a uniprocessor. As a result, these two cache models can be similarly designed and treated at the logic level. For simplicity, we adopt the shared L2 cache model in this thesis.

There are several available protocols for cache coherency [98]: MSI, MSEI, MOSI, MOESI, etc. In our multi-core simulator, we only concentrate on the two basic cache coherency models, MSI and MSEI, which are practically used in ARM Cortex series multiprocessors [99]. MSI and MSEI donate the status of a cache line as their names, and are explained as below:

- Modified (M): the cache line is modified by the cores and this cache line is "dirty" (meaning its content is different with the content in next level memory and needs to written back). For MSEI, the "modified" status is changed to "exclusive" when the write back is completed, that means this cache line only exists in this core's cache. For MSI, the "modified" status is changed to "shared" when the write back is completed, triggered by the other cores' read and write on their cache line with the same address.
- Shared (S): the cache line is shared among the cores (or exists in at least one core in the MSI protocol) and its content is never modified by any core (it is identical to the content in main memory). In any implementation, if the write operation occurs on this "shared" cache line, the written cache line's status is changed to "modified", and the cache lines with same address on other cores is changed from "shared" to "invalid".
- Invalid (I): the cache line neither exists nor is "dirty" in the current cache. Thus, it needs to be fetched from the main memory or the other core's cache line, with a status change from "invalid" to "shared/exclusive" (for a read operation from another cache line or from main memory, respectively) or "modified" (for a write operation).

- Exclusive (E): the cache line only exists in one core's cache and its content is "clean" (identical with the main memory). Another core's read on the same address can make the status on this core change from "exclusive" to "shared". Another core's write to this cache line can change the status from "exclusive" to "modified".

After introducing the basic protocol, we list the added pseudo code for the transactional load/store (read/write) instruction simulation. The initial status for all cache lines in our simulator are set to "invalid".

**Algorithm A.1: Behaviour for Read Hit**

**Input**: $addr$ is physical address translated by TLB or MMU, $core\_id$ is current core.
**Output**: $lat$ is latency of this operation

**IF** $cache[core\_id][addr].status = M\ or\ S\ (or\ E)$ **THEN** //read hit
    Directly fetch the content $cache[core\_id][addr].value$;
    **RETURN** $lat = cache.read\_lat$;
**END IF**

**Algorithm A.2: Behaviour for Read Miss**

**Input**: $addr$ is physical address translated by TLB or MMU, $core\_id$ is current core.
**Output**: $lat$ is latency of this operation

**IF** $cache[core\_id][addr].status = I$ **THEN** //read miss
    **FOR** $i = other\ core's\ id$ **DO**
        **IF** $addr$ is existed in $cache[i]$ **THEN**
            **IF** $cache[i][addr].status = S$ **THEN**
                $cache[core\_id][addr].value = cache[i][addr].value$;
                $cache[core\_id][addr].status = S$;
                **IF** $cycle_{earliest\_bus\_idle} \geq cycle_{current}$ **THEN**
                    $lat = cycle_{earliest\_bus\_idle} + lat\_cache\_trans - cycle_{current}$;
                    $cycle_{earliest\_bus\_idle} += lat\_cache\_trans$;
                **ELSE**
                    $lat = lat\_cache\_trans$;
                    $cycle_{earliest\_bus\_idle} = cycle_{current} + lat\_cache\_trans$;
                **END IF**
                **RETURN** $lat$;
            **END IF**
            **IF** $cache[i][addr].status = M$ **THEN**
                $cache[core\_id][addr].value = cache[i][addr].value$;
                Writeback $cache[i][addr].value$ to lower level memory;
                $cache[core\_id][addr].status = S$;
                $cache[i][addr].status = S$;
                **IF** $cycle_{earliest\_bus\_idle} \geq cycle_{current}$ **THEN**
                    $lat = cycle_{earliest\_bus\_idle} + lat\_cache\_trans + lat\_write\_mem - cycle_{current}$;
                    $cycle_{earliest\_bus\_idle} += lat\_cache\_trans + lat\_write\_mem$;
                **ELSE**
                    $lat = lat\_cache\_trans + lat\_write\_mem$;
                    $\text{cycle}_{\text{earliest\_bus\_idle}} = \text{cycle}_{\text{current}} + \text{lat\_cache\_trans} + \text{lat\_write\_mem}$;
                **END IF**

```
                RETURN lat;
            END IF
            IF cache[i][addr].status = E THEN
                cache[core_id][addr].value = cache[i][addr].value;
                cache[core_id][addr].status = S;
                cache[i][addr].status = S;
                IF cycle_earliest_bus_idle ≥ cycle_current THEN
                    lat = cycle_earliest_bus_idle + lat_cache_trans − cycle_current;
                    cycle_earliest_bus_idle += lat_cache_trans;
                ELSE
                    lat = lat_cache_trans;
                    cycle_earliest_bus_idle = cycle_current + lat_cache_trans;
                END IF
                RETURN lat;
            END IF
        END IF
    END FOR
    Read from low level memory, cache[core_id][addr].value = mem[addr];
    IF MSEI is used THEN
        cache[core_id][addr].status = E;
    ELSE
        cache[core_id][addr].status = S;
    END IF
    IF cycle_earliest_bus_idle ≥ cycle_current THEN
        lat = cycle_earliest_bus_idle + lat_read_mem − cycle_current;
        cycle_earliest_bus_idle += lat_read_mem;
    ELSE
        lat = lat_read_mem;
        cycle_earliest_bus_idle = cycle_current + lat_read_mem;
    END IF
    RETURN lat;
END IF
```

**Algorithm A.3: Behaviour for Write Hit**

**Input**: $addr$ is physical address translated by TLB or MMU, $core\_id$ is current core.
**Output**: $lat$ is latency of this operation

```
IF cache[core_id][addr].status = S THEN
    Directly write cache[core_id][addr].value;
    cache[core_id][addr].status = M;
    FOR i = other core's id DO
        IF addr is existed in cache[i] THEN
            cache[i][addr].status = I;
        END IF
    END FOR
    IF cycle_earliest_bus_idle ≥ cycle_current THEN
        lat = cycle_earliest_bus_idle + cache.write_lat + lat_broadcast − cycle_current;
        cycle_earliest_bus_idle += cache.write_lat + lat_broadcast;
    ELSE
        lat = cache.write_lat + lat_broadcast;
        cycle_earliest_bus_idle = cycle_current + cache.write_lat + lat_broadcast;
    END IF
    RETURN lat;
END IF
IF cache[core_id][addr].status = M THEN
    Directly write cache[core_id][addr].value;
```

```
    IF cycle_earliest_bus_idle ≥ cycle_current THEN
        lat = cycle_earliest_bus_idle + cache.write_lat − cycle_current;
        cycle_earliest_bus_idle += cache.write_lat;
    ELSE
        lat = cache.write_lat;
        cycle_earliest_bus_idle = cycle_current + cache.write_lat;
    END IF
    RETURN lat;
END IF
IF cache[core_id][addr].status = E THEN
    Directly write cache[core_id][addr].value;
    cache[core_id][addr].status = M;
    IF cycle_earliest_bus_idle ≥ cycle_current THEN
        lat = cycle_earliest_bus_idle + cache.write_lat − cycle_current;
        cycle_earliest_bus_idle += cache.write_lat;
    ELSE
        lat = cache.write_lat;
        cycle_earliest_bus_idle = cycle_current + cache.write_lat;
    END IF
    RETURN lat;
END IF
```

## Algorithm A.4: Behaviour for Write Miss

**Input**: $addr$ is physical address translated by TLB or MMU, $core\_id$ is current core.
**Output**: $lat$ is latency of this operation

```
IF cache[core_id][addr].status = I THEN    //write miss
    Directly write cache[core_id][addr].value;
    l = 0;
    FOR i = other core's id DO
        IF addr is existed in cache[i] THEN
            IF cache[i][addr].status = S THEN
                cache[i][addr].status = I;
                l += lat_broadcast;
            END IF
            IF cache[i][addr].status = M THEN
                Writeback cache[i][addr].value to lower level memory;
                cache[i][addr].status = I;
                l += lat_write_mem;
                BREAK;
            END IF
            IF cache[i][addr].status = E THEN
                cache[i][addr].status = I;
                BREAK;
            END IF
        END IF
    END FOR
    IF write-through mode is enable and MSEI is used THEN
        Write cache[core_id][addr].value to lower level memory;
        cache[core_id][addr].status = E;
        l += lat_writeback;
    ELSE
        cache[core_id][addr].status = M;
    END IF
    IF cycle_earliest_bus_idle ≥ cycle_current THEN
        lat = cycle_earliest_bus_idle + l − cycle_current;
```

```
        cycle_earliest_bus_idle += l;
    ELSE
        lat = l;
        cycle_earliest_bus_idle = cycle_current + l;
    END IF
    RETURN lat;
END IF
```

Algorithms A.1 to A.4 describe the cache line status transition process and the latency calculation in case of a cache hit/miss induced by a transactional load/store. These 4 algorithms should complete atomically and cannot be interrupted or disturbed by other transactional load/stores.

The latency $lat$ is expressed as the number of cycles from the current cycle to the completion cycle of the transactional read/write. Since the bus and SCU are shared resources among the cores, a transitional load/store (including all the operations described in any one of the above four algorithms) should exclusively occupy the bus and SCU during the entire transaction. Therefore, we use a global variable $cycle_{earliest\_bus\_idle}$ to track the earliest idle cycle on bus, and this value indicates when a transaction can occupy the bus, and should be updated at the end of each transaction by adding the number of bus cycles needed by a transaction. To calculate the number of bus cycles needed by a transaction, the following latencies needed to be pre-calculated or simulated:

- $cache.read\_lat$ (and $cache.write\_lat$), denote the number of cycles for reading (writing) a cache line directly from the core's own cache. These values are not used by updating $cycle_{earliest\_bus\_idle}$ since it does not access the bus, and is only needed for calculating the latency of a transaction.
- $lat\_broadcase$, denotes the number of cycles needed for broadcasting "invalid" on the bus to each core, and is a pre-determined constant since the broadcasting and snooping behaviours are defined by the bus protocol (e.g. AMBA).
- $lat\_cache\_trans$, denotes the number of cycles needed for transferring a cache line between the caches of two cores, and is a pre-determined constant determined by the size of a cache line and the number of cycles needed for transferring data on the bus.

- $lat\_read\_mem$ (and $lat\_write\_mem$), denote the number of cycles needed for reading (writing) a cache line from (to) the lower level memory, and is a variable evaluated by simulating the lower level memory behaviour (i.e. the shared L2 cache hit/miss and the data transfer between the L2 cache and the main memory). This value is dependent upon the amount of data being transferred and the bus working mode (pipelined or burst).

These constants and variables are evaluated by adopting the AMBA bus protocol [100] in our implementation, and not detailed here. After knowing the transactional load/store latency, the memory access instruction can be integrated into the instruction issue stage mentioned in Section A.1.1.

### A.1.4 System Calls and Instructions Needed By Multi-Threading

In the last two sections, we have detailed the simulated hardware: the core and the inter-core communication, but if we want to run a multi-thread program using the pthread library, we still need the simulated software components required by pthreads. The original SimpleScalar cannot does not support the multi-threading (even multi-threading on a uniprocessor is impossible) since it only implements several the simple system calls needed by a single thread. All the critical system calls for multi-threading, such as thread creation, thread blocking, thread communication and so on, are not implemented in the original SimpleScalar. As a result, if we want to allow a multi-thread program to run on our simulator, these missing system calls must be added.

In fact, the system calls' implementation[30] is dedicated for a certain operating system. But we can assume if there is no OS running on the simulator, the critical system calls can also be simulated and behave as the system calls in a specific OS (e.g. Linux). In this section, we introduce our implementation of the system calls that support multi-thread program execution on the multi-core simulator. Our simulated system calls have similar logical behaviours and results as in Linux, but the detailed implementation and codes are not copied from Linux since some supervisor mode and

[30] The SWI or SVC instruction can trigger a soft IRQ that is called as system service or system call. The soft IRQ handler represents the system call's implementation and can be customized and programmed by operating system designer.

CP15 instructions are not supported in our current version. The system calls in the original SimpleScalar (e.g. file system calls: create/delete, open/close, read/write, link/unlink, lseek, etc.; system function calls: time, chmod, chown, geteuid, getgid, settimer, writeev, usleep, etc.; network sockets and system statistic calls) are unchanged, and are not discussed here. Table A.**2** lists all our added or modified system calls that are needed by multi-threading in the pthread library.

Before introducing our implemented system calls, we added the following global data structures to support multi-threading: 1) $core[core\_id].status$ is to record the status of a core: IDLE, BUSY and SUSPENDED, that can affect the thread allocation and instruction fetching and the execution, or execution pending, for a core (e.g. the core with SUSPENDED status must be pending on the instruction fetching, dispatching or issuing stage and the execution of this core is suspended, while the core with IDLE status can be allocated to a new created thread); 2) signal related structures (e.g. $sigmask[core\_id]$, $pending\_sig[core\_id]$ and $sigaction[cord\_id][num\_sig]$) are used to record the signal masks, the pending signals and signal handler structures of the threads running on all cores, and signal is widely used in inter-process communication; 3) pipe file descriptor (e.g. $pipe\_fd[core\_id][0..1]$) are used to store the pipe file descriptor for the current thread running on the core, where 0 denotes the read endpoint and 1 denotes the write endpoint, and the pipe in pthread is mainly used to send and receive the management information between parent and children; 4) supervision mode indicator (e.g. $in\_kernel[core\_id]$) shows if the thread running on $core\_id$ is trapped inside a system call or not, and this status is used to identify a thread is waiting for another thread or pending on some signal. There is a common memory block read/write function appearing in most system calls: $mem\_bcopy(mem\_fn, mem, \frac{read}{write}, simulated\ memory\ address, data\ pointer, size)$, which is used to copy a block of data (e.g. the content of our internal data structures pointed by data pointer) from/to simulated main memory (pointed by the simulated memory address). In other words, this function can move the data between our simulator and the simulated memory. The pseudo code for the added system calls is given below. Comprehensive inline comments, and system calls where only slight modifications have been made are omitted.

| System Calls Added or Modified | Function |
|---|---|
| getpid (modified) | This is to allow the calling process to get its own process id. |
| getppid (added) | This is to allow the calling process to get its parent process id. |
| rt_sigaction and sigaction (added) | This it to allow the calling process to examine and/or specify the action to be associated with a specific signal. |
| rt_sigpromask and sigpromask (added) | This is to set a mask that indicates which signal events can be blocked by the calling process. The blocked signal is not responded. |
| rt_sigreturn and sigreturn (modified) | This is to allow users to atomically unmask, switch stacks, and return from a signal context. |
| clone (added) | This is to create a child process that shares parts of its execution context with the parent. This is also used to create a thread that is a light-weight process in Linux. |
| kill (added) | This is to send a signal to a specified process. |
| wait4 (added) | This is to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. |
| rt_sigsuspend and sigsuspend (added) | This is to suspend the thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. |
| nanosleep (added) | This is to suspend the execution of the calling thread until the specified time has elapsed. |
| poll (modified) | This is to make a calling process wait for the status changes on a file descriptor. This is usually used by block-reading/writing data from/to a pipe. |
| pipe (modified) | This is to create a bidirectional message queue that supports inter-process communication. It returns two file descriptors, allowing the calling process to send and receive data. |
| exit (modified) | This is to stop the execution of the calling process or thread. |
| mmap (modified) | This is to map a physical memory address to a virtual memory address. |
| read (modified) | This is to read data from a file (described by the file descriptor) to memory. This is also used by reading from a pipe. |
| write (modified) | This is to write data from memory to a file (described by the file descriptor). This is also used by sending to a pipe. |

Table A.2: System calls needed by multi-threading

## Algorithm A.5: clone

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is clone flag, $core[core\_id].regs{->}\ regs\_R[1]$ is child (created) stack pointer, $core[core\_id].regs{->}\ regs\_R[SP]$ is stack pointer of current thread.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ error code

**FOR** $next\_core = 1\ to\ NUM\_CORE$ expect $core\_id$ **DO**
  **IF** $core[next\_core].status = IDEL$ **THEN**
    $core[next\_core].status = BUSY$; //allocate new thread to *next_core*
    core[next_core]$.regs{->}\ regs\_R[SP] = core[next\_core].regs{->}\ regs\_R[1]$; //stack assign: R1 is an argument of calls to indicate the child's stack base
    $core[next\_core].regs{->}\ regs\_PC = core[core\_id].regs{->}\ NPC$ ; //set the start address for new thread, that is the next instruction of parent thread
    $core[next\_core\ ].regs{->}\ regs\_NPC = core[next\_core].regs{->}\ regs\_PC + sizeof\ (md\_inst\_t)$; //set the next pc for new thread
    $core[next\_core].regs{->}\ regs\_R[PC] = core[next\_core].regs{->}\ regs\_PC$;
    $core[next\_core].regs{->}\ regs\_R[0] = 0$; //the return value in child thread is 0
    $ppid[next\_core] = 1024 + core\_id$; //record the child thread's parent pid
    $core[core\_id].regs{->}\ regs\_R[0] = 1024 + next\_core$; //the return value in parent thread is the child's pid
    $pipe\_fd[next\_core][0..1] = pipe\_fd[core\_id][0..1]$; //pipe file descriptor duplication: child has the same pipe with its parent
    **FOR** i=1 to NUM_SIG **DO**
      $sigaction[next\_core][i] = sigaction[core\_id][i]$; // signal handler duplication: child and parent share the same signal handler at the creation
    **END FOR**
    **RETURN** $core[core\_id].regs{->}\ regs\_R[0]$;
  **END**
**END FOR**
core[core_id]. regs$->\ regs_R[0] = -$errval; //cannot find available core to run new thread
**RETURN** $core[core\_id].regs{->}\ regs\_R[0]$;

## Algorithm A.6: sigaction

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is signal number, $core[core\_id].regs{->}\ regs\_R[1]$ is the pointer of a new signal handler struct, $core[core\_id].regs{->}\ regs\_R[2]$ is the pointer of the old signal handler struct.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ error code

$signum =\ core[core\_id].regs{->}\ regs\_R[0]$;
**IF** $core[core\_id].regs{->}\ regs\_R[2] != 0$ **THEN**
  $mem\_bcopy(mem\_fn, mem, Write, core[core\_id].regs{->}\ regs\_R[2],\ \&(sigaction[core\_id][signum]),\ sizeof(struct\ sigaction))$; //write the old sigaction struct to the simulated memory pointed by R2
**END IF**
**IF** $core[core\_id].regs{->}\ regs\_R[1] != 0$ **THEN**
  $mem\_bcopy(mem\_fn, mem, Read, core[core\_id].regs{->}\ regs\_R[1],\ \&(sigaction[core\_id][signum]),\ sizeof(struct\ sigaction))$ ; //read the new sigaction struct from the simulated memory pointed by R1 and store this struct in our internal data structure sigaction.
**END IF**
$core[core\_id].regs{->}\ regs\_R[0] = 0$;
**RETURN** $core[core\_id].regs{->}\ regs\_R[0]$;

## Algorithm A.7: sigprocmask

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is signal flag, $core[core\_id].regs{->}\ regs\_R[1]$ is the pointer of a new signal mask, $core[core\_id].regs{->}\ regs\_R[2]$ is the pointer of the old signal mask.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ error code

$sigflag = core[core\_id].regs{->}\ regs\_R[0];$
**IF** $core[core\_id].regs{->}\ regs\_R[2] != 0$ **THEN**
    $mem\_bcopy(mem\_fn, mem, Write, core[core\_id].regs{->}\ regs\_R[2],$
        $\&(sigmask[core\_id]), 4)$ ; //write the old signal mask to the simulated memory pointed by R2
**END IF**
**IF** $core[core\_id].regs{->}\ regs\_R[1] != 0$ **THEN**
    $qword\_t\ *new\_mask = malloc(4);$
    $mem\_bcopy(mem\_fn, mem, Read, core[core\_id].regs{->}\ regs\_R[1],\ new\_mask,$
        $sizeof(struct\ sigaction));$ //read the new signal mask from the simulated memory pointed by R1 and store this mask in our internal data structure sigmask.
    **SWITCH** $core[core_{id}].regs{->}\ regs\_R[0]$ //set the mask according to flag
    **CASE** OSF_SIG_BLOCK:
        $sigmask[core\_id] \mid= *new\_mask;$
        **BREAK**;
    **CASE** OSF_SIG_UNBLOCK:
        $sigmask[core\_id]\ \&= \sim(*new\_mask);$
        **BREAK**;
    **CASE** OSF_SIG_SETMASK:
        $sigmask[core\_id] = *new\_mask;$
        **BREAK**;
    **DEFAULT**:
        core[core_id].regs$->\ regs_R[0] = -$errval;
    **END SWITCH**
**END IF**
$core[core\_id].regs{->}\ regs\_R[0] = 0;$
**RETURN** $core[core\_id].regs{->}\ regs\_R[0];$

## Algorithm A.8: wait4 (partially implemented)

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is the pid of a thread that is waited for.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ the pid of a thread that is waited for.

**IF** $core[core\_id].regs{->}\ regs\_R[0] = -1$ **THEN**
    $core[core_{id}].regs{->}\ regs\_R[0];$
    **RETURN**;
**END IF**
**IF** $core[core[core\_id].regs{->}\ regs\_R[0] - 1024].status = IDLE$ **THEN**
    $in\_kernel[core\_id] = 0;$ //waited thread is not running
    **RETURN**;
**END IF**
$in\_kernel[core\_id] = 1;$ //suspended in system calls, make PC trap into current instruction to simulate the wait
$core[core\_id].regs{->}\ regs\_R[PC] = core[core\_id].regs{->}\ regs\_PC;$
$core[core\_id].regs{->}\ regs\_NPC = core[core\_id].regs{->}\ regs\_R[PC];$
**RETURN**;

## Algorithm A.9: sigsuspend

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is replaced signal mask, $core[core\_id].regs{->}\ regs\_R[1]$ is the signal mask size.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ is the error code.

$static\ qword\_t\ *repl\_mask;$
**IF** $in\_kernel[core\_id] = 0$ **THEN** //current thread is not pending in system call
  $repl\_mask = malloc(core[core\_id].regs{->}\ regs\_R[1]);$
  $mem\_bcopy(mem\_fn, mem, Read, core[core\_id].regs{->}\ regs\_R[0], repl\_mask, 4);$
**END IF**
**IF** $pendingsig[core\_id]! = 0$ **THEN** //current thread has pending signals to process
  **FOR** i=1 to 32 **DO** //check every bit on pending signal
    **IF** $((pendingsig[core\_id] >> i)\ \&\ 1)$ **AND** $\sim(((*repl\_mask) >> i)\ \&\ 1))$ **THEN**
      //check which signal is pending to be processed
      $core[core\_id].status = BUSY;$
      $core[core\_id].regs{->}\ regs\_R[0] = i + 1;$ //set signal number
      $core[core\_id].regs{->}\ regs\_R[LR] = core[core\_id].regs{->}\ regs\_PC$ ; //store the current pc as the later return point after signal handling (like interrupt)
      $core[core\_id].regs{->}\ regs\_R[PC] = sigaction[core\_id][i + 1].sa\_handler$ ; //the current thread will execute in signal handler
      $core[core\_id].regs{->}\ regs\_NPC = core[core\_id].regs{->}\ regs\_R[PC];$
      $pendingsig[core\_id]\& = \sim(1 << i);$ //clear pending signal
      $in\_kernel[core\_id] = 1;$ //trap to signal handler that is still a part of system call
      **RETURN**;
    **END IF**
  **END FOR**
**ELSE IF** $in\_kernel[core\_id] = 0$ **THEN** //no pending signal and not trapped in kernel
  $core[core\_id].status = SUSPENDED;$ //suspend the current core to wait for signal
  $in\_kernel[core\_id] = 1;$ //the suspended core is stopped its own simulation
  **RETURN**;
**END IF**
$core[core\_id].regs{->}\ regs\_R[0] = -errval;$
$in\_kernel[core\_id] = 0;$
**RETURN** $core[core\_id].regs{->}\ regs\_R[0];$

## Algorithm A.10: kill

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is the pid of a thread that is sent a signal to, $core[core\_id].regs{->}\ regs\_R[1]$ is the signal number.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ the pid of a thread that is waited for.

$wake\_core = core[core\_id].regs{->}\ regs\_R[0] - 1024$ ; //select the core that has suspended thread according to pid
$pendingsig[wake\_core]| = (1 << (\ core[core\_id].regs{->}\ regs\_R[1] - 1));$ //sending signal
$core[wake\_core].status = BUSY;$ //wake up the thread specified by pid
$core[core\_id].regs{->}\ regs\_R[0] = 0;$
**RETURN** $core[core\_id].regs{->}\ regs\_R[0];$

## Algorithm A.11: nanosleep

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is time struct that indicate how long to sleep.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ is the error value.

$struct\ timespec * temp;$
$static\ long\ sleep\_inv = 0;$
**IF** $in\_kernel[core\_id] = 0$ **THEN**

```
        temp = malloc(sizeof(struct timespec));
        mem_bcopy(mem_fn, mem, Read, core[core_id].regs-> regs_R[0],
                temp, sizeof(struct timespec)); //read time struct from simulated memory
        sleep_inv = temp-> tv_sec * 10000; //can be calculated by simulated cycle
        free(temp);
END IF
IF sleep_inv > 0 THEN
        in_kernel[core_id] = 1; //make PC trap in current instruction to simulate the sleep until the
                sleep_inv is equal to 0
        sleep_inv − −; //each cycle, the count is self-reduced by 1
        core[core_id].regs-> regs_R[PC] = core[core_id].regs-> regs_PC;
        core[core_id].regs-> regs_NPC = core[core_id].regs-> regs_R[PC];
        RETURN;
END IF
core[core_id].regs-> regs_R[0] = 0; //the sleep is over
in_kernel[core_id] = 0;
RETURN core[core_id].regs-> regs_R[0];
```

**Algorithm A.12: pipe**

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is the pointer of file descriptors of a pipe.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ is the error number.

```
word_t fd_addr = core[core_id].regs-> regs_R[0];
core[core_id].regs-> regs_R[0] = pipe(pipe_fd[core_id]); //generate a real pipe and store the
        pipe's descriptor in our internal data structure pipe_fd
mem_bcopy(mem_fn, mem, Write, fd_addr, pipe_fd[core_id] , 2 * sizeof(word_t)) ;   //write
        pipe_fd into the simulated memory pointed by R0
RETURN core[core_id].regs-> regs_R[0];
```

**Algorithm A.13: poll (partially modified)**

**Input**: $core[core\_id].regs{->}\ regs\_R[0]$ is the pointer of file descriptors of a pipe, $core[core\_id].regs{->}\ regs\_R[1]$ is the number of the file descriptor, $core[core\_id].regs{->}\ regs\_R[2]$ is the specified time out of poll operation.
**Output**: $core[core\_id].regs{->}\ regs\_R[0]$ is the error number.

```
FOR i = 1 to core[core_id].regs-> regs_R[1] DO
        mem_bcopy(mem_fn, mem, Read, core[core_id].regs-> regs_R[0] + i *
                sizeof(pollfd), &fds[i] , sizeof(pollfd)) ; //read the pipe descriptor from
                simulated memory pointed by R0+offset
END FOR
poll(fds, core[core_id].regs-> regs_R[1], core[core_id].regs-> regs_R[2]); //poll data from
        the real pipe created in pipe system call and change the status of pipe descriptor
FOR i = 1 to core[core_id].regs-> regs_R[1] DO
        mem_bcopy(mem_fn, mem, Write, core[core_id].regs-> regs_R[0] + i *
                sizeof(pollfd), &fds[i] , sizeof(pollfd)); //write the pipe descriptor back to
                simulated memory
END FOR
RETURN core[core_id].regs-> regs_R[0];
```

For the important modifications, we indicate what function we modified to provide the support for multi-threading in Table A.3:

| Modified System Call | Highlight of our modification |
|---|---|
| exit | The status of the core (where a thread exits) is set back to “IDLE”. |
| mmap | We add the MAP_PRIVE and MAP_ANONYMOUS mapping mode to correct the original wrong heap allocation if using malloc() function in the simulated codes. |
| read | We modify the read to support the reading from the modified pipe. |
| write | We modify the write to support the writing to the modified pipe. |
| getpid and getppid | We re-define the pid number of a thread as $pid = 1024 + core\_id$. |

Table A.3: The slight modifications to some existing system calls

In addition to the above system calls, a critical instruction (not implemented in the original SimpleScalar) also needs to be added into our simulator. The most important object in resource competition is the "lock" which allows multiple threads to access a shared resource exclusively. To access a shared resource, we need to firstly check the lock (read the lock value from memory), then if it is free, change the lock value (to declare the resource belongs to this thread), and then write back the new value to memory. These consecutive operations (i.e. read-change-write) need to be completed atomically without any interruption. In a specific instruction set architecture, there is an atomic instruction for this purpose. In the ARM architecture, the instruction is SWP [101], which can swap a value between a register and the specified memory address in a single step. We have made the following modifications to add the SWP instruction into our simulator.

- To make the simulator recognize the SWP instruction, we add SWP to the decoder and split SWP into three micro-code instructions (*MOV Rt, R0; LDR R0, mem; STR mem, Rt;*).
- To prevent the order of three instructions being affected by other instructions running on the same core, these three instructions must be dispatched and issued[31] in a single cycle and added into the ready queue in their original order (i.e. *MOV*, *LDR* and *STR* are added into RUU and LSQ station consecutively without interruption).

[31] Once three instructions' dependencies are all resolved, these three instructions can be issued together. This is done by slightly modifying the issue stage.

- To prevent the other cores from performing a read/write to the same memory address and disturbing the three instructions, we define a global variable, $atomic\_address[core\_id]$, to record the memory address *mem*. All load/store instructions on the other cores should check this memory address before they are issued to their load/store queues. Once the same (conflict) address is verified, the other core's load/store instruction is made pending in its issue stage [32]. After SWP instructions finishes the three micro-code instructions, $atomic\_address[core\_id]$ is reset to 0 so as to allow the resumption of the other pending core's execution.

After adding (modifying) the necessary data structures (e.g. global variables, arrays), system calls and the SWP instruction[33], we can run a multi-thread program (invoking pthread functions) on our multi-core simulator. The common pthread library functions tested by us are listed in Table A.4:

| **pthread Funtions** | **System Calls Used** |
|---|---|
| pthread_attr_init/pthread_attr_destroy | mmap, free |
| pthread_mutex_init/pthread_mutex_destroy | mmap, free |
| pthread_spin_init/pthread_spin_destroy | mmap, free |
| pthread_cond_init/pthread_cond_destroy | mmap, free |
| pthread_create | mmap, clone, sigprocmask, pipe, exit, sigaction, sigsuspend, kill, write, read, getpid, getppid, SWP |
| pthread_join/pthread_exit | sigsuspend, wait4, kill, write, exit |
| pthread_kill | kill |
| pthread_mutex_lock | sigsuspend, kill |
| pthread_mutex_unlock | kill |
| pthread_spin_lock | sigsuspend, kill |
| pthread_spin_unlock | kill |
| pthread_cond_wait | sigsuspend, kill |
| pthread_cond_signal | kill |
| pthread_cond_broadcast | sigsuspend, kill |
| signal related functions | sigaction, sigprocmask, sigreturn, sigsuspend, kill, write |
| semaphore related functions | sigsuspend, kill, pipe, read, write |

Table A.4: Tested pthread functions

[32] We set the value of PC and NPC to the same current instruction's address to trap the core, similar to that of the wait4 system call.
[33] SWP is included in the function compare_and_swap(), which appears in pthread functions that use __pthread_lock().

## A.2 Summary

In this section, we introduced our implementation of a multi-core simulator for the ARM architecture based on the original SimpleScalar simulator. There are 3 important modifications that were required: 1) modify the uniprocessor for the multi-core scenario by implementing transactional memory access; 2) implement the inter-core communication infrastructure (SCU and bus) to support the full MSEI cache coherency protocol; 3) add the critical system calls and SWP instructions that are needed by the pthread library. We have successfully tested our simulator using a multi-threaded program covering most of the pthread main functions.

We then use our multi-core simulator for determining core power consumption for a number of benchmarks, which we use in subsequent chapters.

# Appendix B

# An Example of Online Thermal Estimation

To illustrate our thermal estimation technique, we present an example, using a simple 2×2 CMP. A 2x2 structure is chosen to appropriately simplify the explanation and the presentation of results. The single LUT of Table 3.1 (in Chapter 3, and repeated here as Table B.1) is used for all calculations due to the thermal symmetry of the 2×2 CMP. In this example, we assume that the core has already been heated above ambient temperature. That is we assume that the processor has been operational for some time. We make an arbitrary assumption that the core temperature is at an initial temperature of 45°C, that is $T_{t=0ms}$= 45°C. However, any initial temperature can be chosen.

| TDL A | Core1 | Core2 | Core3 | Core4 |
|---|---|---|---|---|
| 0ms | 0 | 0 | 0 | 0 |
| 10ms | 0.1553 | 0.0007 | 0.0007 | 0.0000 |
| 20ms | 0.1788 | 0.0012 | 0.0012 | 0.0000 |
| 30ms | 0.1844 | 0.0016 | 0.0016 | 0.0001 |
| 40ms | 0.1880 | 0.0019 | 0.0019 | 0.0001 |
| 50ms | 0.1912 | 0.0021 | 0.0021 | 0.0001 |
| 60ms | 0.1943 | 0.0024 | 0.0024 | 0.0001 |
| 70ms | 0.1971 | 0.0028 | 0.0028 | 0.0002 |
| … | … | … | … | … |
| 500ms | 0.2013 | 0.0648 | 0.0648 | 0.0446 |
| 520ms | 0.2014 | 0.0649 | 0.0649 | 0.0447 |
| 540ms | 0.2015 | 0.0650 | 0.0650 | 0.0448 |
| … | … | … | … | … |
| 1000ms | 0.3233 | 0.0854 | 0.0854 | 0.0639 |
| 1050ms | 0.3235 | 0.0856 | 0.0856 | 0.0640 |
| … | … | … | … | … |
| 2000ms | 0.3504 | 0.1116 | 0.1116 | 0.0891 |
| 2100ms | 0.3506 | 0.1118 | 0.1118 | 0.0893 |
| … | … | … | … | … |
| Steady | 0.3819 | 0.1428 | 0.1428 | 0.1200 |

Table B.1: Look-Up Table for a 2×2 CMP

1. At $t = 10ms$, the first task with $ae.P = 20$ is allocated at $Core1$. Since $\mathbb{Q}$ is empty at the start, the temperature of each core at $t = 10ms$ is not modified. Then the atomic power event $ae(10, 20, Core1)$ is inserted into $\mathbb{Q}$ with $t_p = 10$.

2. To estimate the temperature at $t_n = 40$ , we use $T_{40} = T_{10} + 20 \times trans(\Re^{LUT\,A}_{40-10} - \Re^{LUT\,A}_{10-10},\ Core1)$. The $40 - 10 = 30ms$ row and $10 - 10 = 0ms$ row are fetched from Table 0.1 and subtracted. A transformation is not needed as $trans(, Core1) = 0$. Therefore, the estimated temperature at $40ms$:

$$T_{40} = (45,45,45,45) + 20 \times \{(0.1844,0.0016,0.0016,0.0001) - (0,0,0,0)\} = (48.688,45.032,45.032,45.002)$$

3. At $t = 50ms$, a second task with $ae.P = 30$ is assigned to $Core2$. Now, $t_c = 50$, $t_p = 10$ and there is one event in $\mathbb{Q}$, the thermal map should be updated as:

$$\begin{aligned} T_{50} &= T_{10} + 20 \times trans(\Re^{LUT\,A}_{50-10} - \Re^{LUT\,A}_{10-10}, Core1) \\ &= (45,45,45,45) + 20 \times \{(0.1880,0.0019,0.0019,0.0001) - (0,0,0,0) \\ &= (48.76,45.038,45.038,45.002) \end{aligned}$$

The second event $ae(50, 30, Core2, 1)$ is then queued into $\mathbb{Q}$ and $t_p = 50$.

4. To estimate the temperature at $t_n = 70$, we can accumulate the temperature increment induced by the two events in $\mathbb{Q}$.

$$\begin{aligned} T_{70} = T_{50} &+ 20 \times trans(\Re^{LUT\,A}_{70-10} - \Re^{LUT\,A}_{50-10}, Core1) + 30 \\ &\times trans(\Re^{LUT\,A}_{70-50} - \Re^{LUT\,A}_{50-50}, Core2) = \\ &= (48.76,45.038,45.038,45.002) + 20 \\ &\times (0.0063,0.0005,0.0005,0.0000) + 30 \\ &\times (0.0012,0.1788,0.0000,0.0012) = (48.922,50.312,45.048,45.038) \end{aligned}$$

where $trans(, Core2)$ corresponds to mid-y mirroring, when calculating the temperature increment induced by the second event in $\mathbb{Q}$. This procedure is illustrated in Figure B.1.

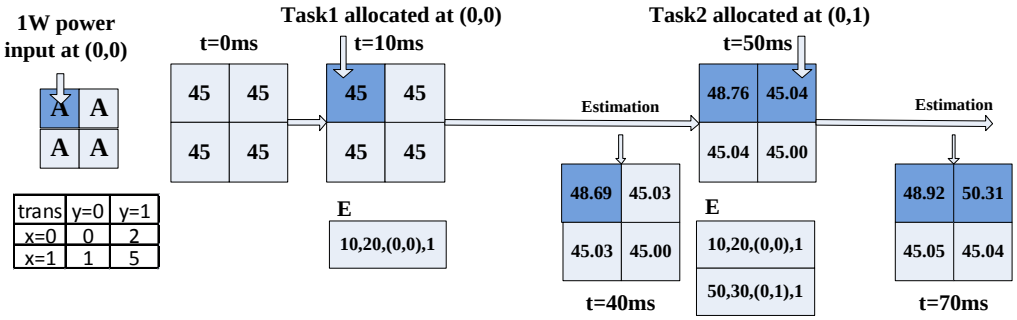



Figure B.1: Example of online event driven estimation

# Appendix C

# Author's Publications List

## Journal:

1 Jin Cui, Douglas L. Maskell: A Framework for Multiprocessor Schedulability Analysis with Thermal Constraints. Submitted to ACM Trans. on Embedded Computing Systems.

2 Jin Cui, Douglas L. Maskell: A Fast High-Level Event-Driven Thermal Estimator for Dynamic Thermal Aware Scheduling. Published by IEEE Trans. on CAD of Integrated Circuits and Systems 31(6): 904-917 (2012).

3 Zonghua Gu, Weichen Liu, Jiang Xu, Jin Cui, Xiuqiang He, Qingxu Deng: Efficient Algorithms for 2D Area Management and Online Task Placement on Runtime Reconfigurable FPGAs. Published by Microprocessors and Microsystems - Embedded Hardware Design 33(5-6): 374-387 (2009).

## Conference:

4 Jin Cui, Douglas L. Maskell: High Level Event Driven Thermal Estimation for Thermal Aware Task Allocation and Scheduling. In Proceeding of ASP-DAC 2010: 793-798.

5 Jin Cui, Douglas L. Maskell: Dynamic Thermal-Aware Scheduling on Chip Multiprocessor for Soft Real-Time System. In Proceeding of ACM Great Lakes Symposium on VLSI 2009: 393-396.

## Others before Enrolment:

6 Jin Cui, Qingxu Deng, Xiuqiang He, Zonghua Gu: An Efficient Algorithm for Online Management of 2D Area of Partially Reconfigurable FPGAs. In Proceeding of DATE 2007: 129-134.

7 Jin Cui, Zonghua Gu, Weichen Liu, Qingxu Deng: An Efficient Algorithm for Online Soft Real-Time Task Placement on Reconfigurable Hardware Devices. In Proceeding of ISORC 2007: 321-328.