UNIVERSITY OF CALIFORNIA, SAN DIEGO

Temperature and Cooling Management in Computing Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Engineering

by

Raid Ayoub

Committee in charge:

Professor Tajana Rosing, Chair
Professor Chung-Kuan Cheng
Professor Bill Hodgkiss
Professor Ryan Kastner
Professor Dean Tullsen

2011

The dissertation of Raid Ayoub is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2011

DEDICATION

To my family and memory of my father.

EPIGRAPH

Imagination is more important than knowledge.
For knowledge is limited, whereas imagination embraces the entire world,
stimulating progress, giving birth to evolution.
It is, strictly speaking, a real factor in scientific research.
–Albert Einstein

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere gratitude to my advisor, Prof. Tajana Rosing, for her guidance in research, unwavering encouragement and preparing me to carry on my research after school. I thank her for giving me the opportunity to be her student. I also thank my doctoral committee, Prof. Dean Tullsen, Prof. Bill Hodgkiss, Prof. Ryan Kastner and Prof. Chung-Kuan Cheng for their valuable feedback and contributions.

My internships at Intel research and Cisco during my PhD were a great experience. I sincerely thank Umit Ogras, Michael Kishinevsky, Eugene Gorbatov, Kam Timothy, Paul Diefenbaugh, and Inder Bhasin for their guidance, ideas and discussions that helped my research going forward. My research work was made possible by funding from NSF Project GreenLight Grant 0821155, NSF Grants 0916127 and 1029783, NSF CIAN EEC-0812072, MuSyC, DARPA, UC Micro, Oracle, Google, UCSD Center for Networked Systems and NSF Variability. I thank them for their generous support.

I am grateful to all my lab mates and colleagues for their valuable contribution through comments, discussions and guidance. I thank Gaurav Dhiman, Shervin Sharifi, Rajib Nath, Raju Indukuri, Yanqin Jin, Yen-Kuan Wu, Ayse Coskun, Giacomo Marchetti, Bryan Kim, Nima Nikzad, Yashar Asgarieh, Arup De, Priti Aghera, Vasileios Kontorinis and Edoardo Regini for their friendship and contributions to my research.

I want to express my gratitude to my family for their relentless support and encouragement. I thank my wife, Bushra Ayoub, for her unwavering support, love and encouragement, which made the tough times in my Ph.D seem easier. I am also thankful to my mother for her endless caring and support which made my path throughout the Ph.D. much smoother. I am grateful to my sisters, Daliah, Rana and Zena as they always encouraged me and kept me inspired. I feel very lucky to have such a wonderful family.

Chapters 1 and 2, in part, are a reprint of the material as it appears in Proceedings of the International Conference on Computer Design, 2007. Ayoub,

R. and Orailoglu, A. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 3, in part, are a reprint of the material as it appears in Proceedings of the International Symposium on Low Power Electronics and Design, 2009. Ayoub, R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 3, in part, are a reprint of the material as it appears in IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems, 2011. Ayoub, R.; Indukuri, K. R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 4, in part, are a reprint of the material as it appears in IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems, 2011. Ayoub, R.; Indukuri, K. R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 4, in part, are a reprint of the material as it appears in Proceedings of the International Symposium on Low Power Electronics and Design, 2009. Ayoub, R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 5, in part, are a reprint of the material that is in preparation to be submitted to IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems. Ayoub, R.; Nath R.; Indukuri, K. R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

VITA

B.S. in Electrical Engineering
University of Technology, IRAQ

M.S. in Electrical Engineering
University of Technology, IRAQ

Ph.D. in Computer Engineering
University of California, San Diego, La Jolla, CA

PUBLICATIONS

Ayoub, R.; Indukuri, K. R. and Rosing, T.S., Temperature Aware Dynamic Workload Scheduling in Multisocket CPU Servers. IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems (*TCAD*), vol. 30, no. 9, 2011.

Ayoub, R.; Ogras, U; Gorbatov, E.; Jin, Y.; Timothy, K; Diefenbaugh, P. and Rosing, T.S., Power Minimization Under Tight Performance Constraints in General Purpose Systems. In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (*ISLPED*), 2011.

Ayoub, R.; Induk R. and Rosing, T.S., Energy Efficient Proactive Thermal Management in Memory Subsystem. In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (*ISLPED*), 2010.

Ayoub, R. S.; Sharifi, S.; and Rosing, T.S., GentleCool: Cooling Aware Proactive Workload Scheduling in Multi-Machine Systems. In Proceedings of the Design, Automation and Test in Europe (*DATE*), 2010.

Ayoub, R. and Rosing, T.S., Cool and Save: Cooling Aware Dynamic Workload Scheduling in Multi-socket CPU Systems. In Proceedings of the Asia and South Pacific Design Automation Conference (*ASP-DAC*), 2010.

Ayoub, R. and Orailoglu A., Performance and Energy Efficient Cache Migration Approach for Thermal Management in Embedded Systems. In Proceedings of the Great Lakes Symposium on VLSI (*GLSVLSI*), 2010.

Ayoub, R. and Orailoglu A., Filtering Global History: Power and Performance Efficient Branch Predict, In the Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (*ASAP*), 2010.

Ayoub, R. and Rosing, T.S., Predict and act: dynamic thermal management for multi-core processors. In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (*ISLPED*), 2009.

Dhiman G.; Ayoub, R. and Rosing, T.S., PDRAM: A Hybrid PRAM and DRAM Main Memory System. In Proceedings of the 46th ACM/IEEE Design Automation Conference (*DAC*), 2009.

Ayoub, R. and Orailoglu A., Power efficient register file update approach for embedded processors. In Proceedings of the International Conference on Computer Design (*ICCD*), 2007.

Ayoub, R. and Orailoglu A., Low Power Branch Predictor for Application Specific Processors. In Proceedings of the Workshop on Application Specific Processors (*WASP*), 2005.

Ayoub, R. and Orailoglu A., A Unified Transformational Approach for Reductions in Fault Vulnerability, Power, and Crosstalk oise & Delay on Processor Buses. In Proceedings of the Asia and South Pacific Design Automation Conference (*ASP-DAC*), 2005.

Ayoub, R. and Orailoglu A., Instruction Memory Transformations for reductions in Power and Fault Vulnerability on Embedded Processors. In Proceedings of the International SoC Conference (*SOCC*), 2004.

ABSTRACT OF THE DISSERTATION

Temperature and Cooling Management in Computing Systems

by

Raid Ayoub

Doctor of Philosophy in Computer Engineering

University of California, San Diego, 2011

Professor Tajana Rosing, Chair

Temperature and cooling are critical aspects of design in today's and future computing systems. High temperature has a significant impact on reliability, performance, leakage power and cooling energy costs. State of the art temperature management techniques come with performance overhead and do not optimize for cooling energy costs. Energy management techniques usually focus on optimizing the computing energy without considering the impact on temperature or cooling system. In general, managing temperature, cooling and energy separately leads to suboptimal solutions. In this thesis we introduce a new hierarchical approach that manages the temperature, cooling and energy problems jointly and with low overhead. Our approach addresses microarchitecture, core, socket and system levels.

At the microarchitectural level we achieve temperature and energy optimizations by eliminating the redundant writes to the register file at minimal performance overhead. The experimental results show that our technique is able to achieve on average 22% energy savings in register file with 4^oC reduction in temperature. We next introduce a novel core level proactive thermal management technique that intelligently allocates jobs across cores of a single CPU socket to create a better thermal balance across the chip. We introduce a novel temperature predictor that is based on the band limited property of the temperature frequency spectrum where the prediction coefficients can be identified accurately at design time. Our results show that applying our algorithm considerably reduces the average system temperature, hottest core temperature, and improves performance by 6 ^oC, 8 ^oC and 72% respectively. At the CPU socket level, we propose a new algorithm which schedules the workload between sockets to minimize cooling energy by creating a better balance in temperature between the sockets. The reported results show that combining the socket level with the core level optimizations can result in cooling energy savings of 80% on average at performance overhead of less than 1%. Finally, we describe a combined temperature, cooling and energy management approach that significantly lowers the cooling energy costs of the system as well as the operational energy of memory. We introduce a comprehensive thermal and cooling model which is used for online decisions. This technique clusters the memory accesses to subset of memory modules in tandem with balancing the temperature between and within the CPU sockets. The experimental results show that our method delivers an average cooling and memory energy savings of up to 70% compared to the state of the art techniques at performance overhead of less than 1%.

Chapter 1

Introduction

Technology scaling coupled with the high demand for computation leads to increase in integration and performance of the modern computing systems. This trend, however, is facing serious challenges since it results in a substantial increase in power/energy consumption. Higher power consumption combined with high integration elevates the device power density which causes thermal hot spots. High temperature has a substantial effect on reliability, performance and leakage power. When temperature reaches thermal emergencies, the computations need to be throttled to keep the temperature in the safe zone, an issue that degrades performance. Dissipating the excess heat is a big challenge as it requires a complex and energy hungry cooling subsystem. This is because the power consumed by fan subsystem is cubically related to the air flow rate [47]. This indicates that thermal and cooling are important aspects of design in computing systems.

A common way to mitigate temperature problems in today's processors is to throttle computations to reduce power density. The obvious drawback of this technique is its performance overhead. One alternative approach is to reduce redundant activities within cores to lower their power density. This approach can be applied at the microarchitectural level. At the operating system level (OS) there are potential savings opportunities by optimizing the assignment of jobs to cores within CPU sockets. One possible approach is to dynamically balance the jobs across the cores to reduce the overall power density. Dynamic load balancing is usually implemented in modern operating systems to enhance the utilization of

the system resources. This policy performs thread migration to lower the difference in task queue lengths of the individual cores [21]. The operating system initiates dynamic load balancing every hundred milliseconds. Although this simple solution has a potential to reduce power density of the chip, it provides suboptimal solutions thermally. When hardware utilization is below 100%, dynamic load balancing policy does not exploit the idle cores to create a better thermal distribution in the chip by migrating the jobs among the cores in a thermally sensitive manner. The overhead of such migration is usually acceptable since the time constant of temperature is much higher than thread migration overhead. A challenge is how to decide on the effective migration during run time. For the case of multiple CPU sockets, balancing the jobs without accounting for their thermal profile lead to imbalances in the thermal distribution among the CPU sockets. This is because the thermal profile of the jobs is not the same due to the differences in the way the exercise the resources. For example, CPU intensive jobs tend to generate more heat then memory intensive ones. This thermal imbalance increases the cooling energy costs due to the nonlinearity between fan speed and its power. The proper approach is to schedule the workload between CPU sockets in a way that creates a better temperature balance between them to minimize cooling energy costs.

Management of temperature, cooling and computing energy problems at the system level is a big challenge since this involves heterogeneity in components in addition to the workload. One of the key disadvantages of state of the art work is that cooling, temperature and computing energy are considered separately. For example, clustering the computations to subset of the resources to save energy must account of potential temperature rise which may require more cooling energy that compromise savings. Moreover, thermal dependencies can also occur when components share the same cooling resources which need to be considered to optimize cooling energy. A holistic approach is required that integrates the workload assignments with thermal and cooling aspects to ensure efficiency and stability in order to mitigate the cooling and thermal problems at low overhead.

1.1 Related work on thermal management

Thermal management is vibrant area of research as temperature has a significant impact on a number of important aspects in computing systems. The increase in on chip temperature can result in severe reliability degradation since it causes stress migration, electromigration and dielectric breakdown which can lead to permanent damage [7, 48]. High temperature also impacts leakage power due to exponential dependency of leakage power on temperature [47]. The increase in temperature degrades the performance due to its impact on elevating the interconnect delay [9].

A number of core level thermal management techniques have been suggested in recent years. They can be broadly classified into two categories: reactive and proactive management techniques. The research in [55] proposes multiple reactive DTM techniques at the microarchitectural level. The first technique reduces the power density in the core by employing instruction fetch toggling to reduce the execution. The benefit of this technique is that it can manage temperature at fine grain granularity at the cost of performance. The authors in [55] also show that register file is the hottest component in the processor due to its high power density. To overcome the thermal problems in register file the research suggests adding an extra register file to reduce the power density using activity migration, only one register is active at any point in time. When one register file become hot, its contents are migrated to the cooler one and so on. The primary overhead in these this technique is the extra hardware and additional communications latency. A class of techniques aims at reducing the power density of register file through dynamically reducing the size of the register file through the use of banking [23] and by reducing the number of register ports [34, 46]. However, applying such approaches come at loss in performance. The research in [21] proposes thread migration at the operating system level to manage the excess temperature by moving the computations across duplicated units. However, this technique misses savings opportunities since it reacts only upon reaching thermal emergencies. Recently, dynamic thermal management has become an integral part of actual processor designs. For example, the Xeon processor employs clock gating and dynamic voltage-scaling to manage

thermal emergencies.

To overcome the inefficiencies with the reactive techniques, a class of proactive thermal management techniques have been suggested that try to avoid the thermal emergencies while the processor is still running below the temperature threshold. The authors in [22] propose use of ARMA model that is based on the serial autocorrelation in the temperature time series data. The model is updated dynamically to adapt to possible workload changes. Although the ARMA model is accurate, it requires a training phase that could impact the performance, and the predictor could miss some of the prediction opportunities during training. The authors in [35, 63] suggest proactive thermal management techniques that utilize regression-based thermal predictors. However, these techniques also require run-time adaptation. These solutions are limited to a single socket, hence they are not efficient to minimize cooling energy since some sockets may generate much more heat than others, which requires a better heat balance between them.

To handle thermal problems in memory subsystem, a few techniques have been suggested [36, 37]. The work in [37] mitigates overheating in memory system by adjusting memory throughput to stay below the emergency level. Research in [36] manages memory overheating by grouping threads in the system in such a way that each group is mapped to certain dual-in-line memory modules (DIMMs) assuming that all the threads in a group can be active simultaneously. Only one group of DIMMs is active at any point in time while the rest stay inactive to cool. The authors assume that the bandwidth of each thread is known in advance through static profiling which is normally not the case in dynamic systems. Additionally, this approach is restricted to the cases when number of threads is a multiple of the number of cores. Recent research has sought a number of techniques to handle the energy consumption in memory subsystems. The work in [25] lowers DRAM power by changing memory mode to low power state when there are no accesses to the memory. This approach is beneficial only when there are frequent idle periods. Research in [28] manages memory power by clustering heavily accessed pages to a subset of the memory modules, in particular DIMMs, and putting the rest of the DIMMs in a self-refresh mode. However, such consolidation

is likely to generate thermal problems due to clustering. None of techniques on CPU or memory consider cooling costs.

1.2 Related work on cooling management

Cooling in high-end servers usually relies on the forced convection phenomena to enhance the heat transfer between the heat sink and the ambient. As the CPU power density escalates, the air flow rate must enlarge at a similar rate. Unfortunately, the fan power increases substantially due to the cubic relationship between fan speed and its power [47]. The fan system power in high-end servers is as much as 80 Watts in 1U rack servers [47] and 240 Watts in 2U rack servers or more [1]. In addition to the increase in fan power, high fan speed introduces large noise levels. It is shown that the acoustic noise level increases by 10 dB as air flow rate increases by 50% [38]. This indicates that minimizing air flow to just what is needed for cooling is an essential metric to deliver appreciable energy efficiency at lower acoustic noise levels.

Recently, a few fan control algorithms have been suggested which operate based on closed loop control schemes to optimize the fan controller [20,62]. The research in [62] suggests an optimal fan speed mechanism for the blade servers where the optimal speed is determined through convex optimizations. The work in [54] suggests a fan control mechanism that also considers the leakage power. However, these techniques provide suboptimal solutions since they are not integrated with workload scheduling. A class of techniques have been suggested to improve cooling efficiency at the cluster and data center levels [31, 52, 59]. The authors in [52, 59] suggest the use of workload scheduling to mitigate the air circulation problem in data centers. The research in [31] implements a thermal-aware load balancer which models the temperature within the servers using workload utilization. However, these techniques are not so effective for minimizing the temperature and cooling costs within the servers. In [2, 47] methodologies are proposed for modeling the convective thermal resistance between the heat sink and ambient temperature as a function of the air flow rate, which we use in this thesis.

1.3 Thesis contributions

Temperature, cooling and computational energy optimizations need to be performed in an integrated fashion to maximize efficiency due to the dependencies between them. To address these challenges we developed a hierarchical management approach that allows the optimizations to span multiple components in the system simultaneously while keeping the complexity at manageable level. In the following we provide the summary of our primary contributions:

- We introduce thermal and power management approach at the microarchitectural level. We target application specific processors where the tasks are known in advance. The temperature and power optimizations are achieved by eliminating the redundant writes to the register file at virtually no performance overhead. The redundant write information is encoded in the registers name space using post compiler register renaming algorithms. We developed a cost efficient hardware support to capture these redundant writes. The experimental results show that this technique is able to achieve on average 22.3% energy savings in register file and 4^{o}C reduction in temperature.

- We introduce a novel proactive thermal management technique that schedules the workload intelligently between the cores to reduce temperature and cooling energy. Our proactive thermal management algorithm is able to prevent thermal emergencies in many instances, thus avoiding performance overhead. In order to predict temperature accurately at low cost, we present a novel temperature predictor that is based on the band limited property of the temperature frequency spectrum. The important feature of our predictor is that the prediction coefficients can be computed at the design stage which makes it workload independent. Our results show that applying our algorithm considerably reduces the average system temperature, hottest core temperature, and improves performance by 6 ^{o}C, 8 ^{o}C and 72% respectively.

- We introduce a novel multi-tier algorithm that schedules the workload at the core as well as the socket levels to minimize for cooling energy and the

occurrence of thermal emergencies. We schedule the workload between CPU sockets in a way that mitigates hot spots across them and reduces cooling energy. We developed a control theoretic framework for the socket level scheduling that guarantees the desired objectives, in terms of energy savings and stability, are met. For the core level we use our proactive thermal management technique to reduce the hot spots across the cores and improve cooling savings within a given socket. The reported results show that our multi-tier scheme is able to deliver cooling energy savings of 80% on average while keeping performance overhead below 1%.

- We propose a combined energy, thermal and cooling management technique that significantly lowers the energy consumption and cooling costs of CPU sockets and memory. Our analysis shows that decoupling these optimizations leads to suboptimal solutions due to thermal dependencies between CPU and memory, and non-linearity in cooling energy costs. We propose a comprehensive thermal and cooling model which is used for online decisions. This technique reduces the operational energy of the memory by clustering pages to a subset of memory modules while accounting for thermal and cooling aspects. At the same time this technique tries to remove hot spots between and within the sockets, and reduces the effects of thermal coupling with memory to save cooling costs. We designed our technique using formal control to ensure stability. The experimental results show that our technique delivers total cooling and memory energy savings of up to 70% on average compared to the state of the art techniques at performance overhead less than 1%.

Chapters 1 in part, is a reprint of the material as it appears in Proceedings of the International Conference on Computer Design, 2007. Ayoub, R. and Orailoglu, A. The dissertation author was the primary investigator and author of this paper.

Chapters 1 in part, is a reprint of the material as it appears in Proceedings of the International Symposium on Low Power Electronics and Design, 2009. Ayoub, R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 1 in part, is a reprint of the material as it appears in IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems, 2011. Ayoub, R.; Indukuri, K. R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 1 in part, is a reprint of the material that is under preparation to be submitted to IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems. Ayoub, R.; Nath R. ;Indukuri, K. R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 2

Temperature Aware Microarchitectural Design

Functional units of the processor vary in terms of their thermal profiles. Register file is known to be the hottest component in the core due to its high power density [55]. High power dissipation come from the frequent accesses to the register file coupled with the use of multiple ports [11,27,34]. In Motorola M.CORE architecture, the register file contributes 16% of the total processor power and 42% of the data path power. Consequently, handling the high power dissipation in a register file not only leads to a cheaper and smaller packaging but also lower energy costs, an issue that improves the usability and quality of the product.

In pipelined processors, data forwarding is one of the standard hardware approaches for enhancing performance [32]. In a processor that supports data forwarding, the outcome of the instruction can be forwarded to its dependent instructions before it commits. Forwarding the data improves performance by eliminating unnecessary stalls in the pipeline since the dependent instruction does not need to wait for the source instruction to write its results in the register file for it to start the operation. For a class of in-order processors, unnecessary writes to the register file occur when the instruction passes its results to all of its dependent instructions before it commits. The reported results in [46, 51] show that a significant portion of the writes are redundant due to forwarding. In the light of this fact, elimination of unnecessary writes can reduce the temperature

while deliver energy savings.

Utilizing application-specific knowledge can be the keystone to a set of techniques aimed at improved power savings and performance in application specific processors [12,49,50]. Application specific processors are attractive since they can be optimized to perform certain tasks efficiently and they can be part of high end or mobile chips. We introduce an application specific scheme to mitigate the temperature and high power dissipation in register files for the class of in-order processors. The straightforward approach for precluding the unnecessary accesses to the register file is through marking each instruction that needs to write to the register file [51]. However, the power savings in such a paradigm are limited due to the need to store nontrivial amounts of information, an issue that diminishes the power savings due to the associated power expenditure in storing such information. Furthermore, some of this information is redundant since a portion of the ISA instructions do not even write to the register file in the first place, an issue that further lowers the applicability of such a design paradigm.

Instead of encoding the write permission information per instruction, we propose using the register name space for encoding by marking a small subset of the registers to preclude the writing to the register file. Only a few bits are required in such encoding. In the approach we introduce, static liveness analysis is performed to determine the liveness distance of the registers. Based on this analysis we decide whether the register needs to be written to the register file. In order to increase the skew in the set of preserved registers, a static register renaming approach is suggested in this work. The information regarding the preserved registers is extracted at the granularity of the application major loops. This information is preloaded into the processor to be utilized during program execution. Our solution also handles the unpredictable dynamic events, e.g. instruction cache misses and exceptions. The microarchitectural support is not only highly cost efficient but is also dynamically programmable, thus preserving the generality of the processor while applying the encoding methodology that we propose. The experimental results show that the proposed technique is able to achieve on average 22.26% energy savings in register file and 4^oC reduction in its temperature.

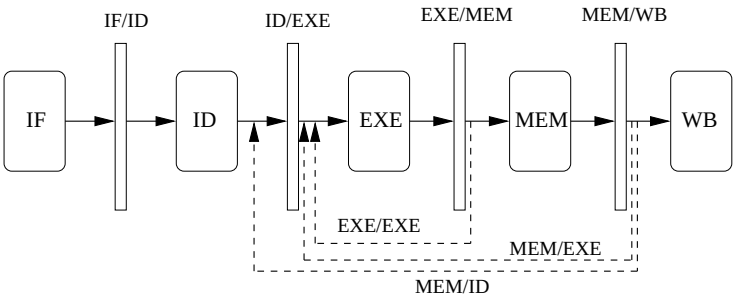


Figure 2.1: Five stage pipeline with forwarding

2.1 Design flow

Our approach utilizes application-specific knowledge and specialized hardware support to eliminate the redundant writes in the register file. It encodes the redundant write information in the application code which can be decoded easily by the hardware. This solution is restricted to application specific processors since it handles the cases when the tasks are known a priori.

Our design flow starts with generation of the application assembly code by the compiler followed by the identification of the optimal application-specific low power code. Subsequent to the identification of the efficient encoding for the registers, the encoding is attained by renaming the original register assignment. Then the modified assembly code is assembled and linked by the compiler to generate the binary code. The resultant binary code is loaded to the instruction memory. A specialized low-cost hardware unit on the processor side captures the registers that do not need to be written to the register file to attain the power savings.

2.2 Forwarding based pipelined processors

In this section we review the basics of forwarding across the pipeline stages. Figure 2.1 show an illustrative example of a basic five stage pipeline processor. The five stages are: (1) IF: fetch the instruction from the memory system, (2) ID: instruction decode and operand read from the register file, (3) EXE: execute stage, ALU, (4) MEM: memory access stage, (5) WB: write back the results to the

register file. The forwarding network is composed of three forwarding paths. In the following we give a brief discussion for this forwarding network:

- **EXE/EXE**: This path allows forwarding the operand of instruction I_k from its predecessor instruction I_{k-1}.
- **MEM/EXE**: This path allows forwarding the operand of the instruction at the EXE stage, I_k, from an older instruction at the MEM stage, I_j, where the *forwarding distance* $FD = k - j$ is limited to at most 2.
- **MEM/ID**: This path allows the forwarding distance to be at most 3.

From this illustration we can conclude that when the distance between instructions is within the forwarding span, then there is no need for the dependents of the producer instruction to wait until the producer commits. As the forwarding network is capable of passing the source operand across the pipeline stages then there is no need for the instruction to write to the register file if its outcome can be forwarded to all its dependents. Eliminating unnecessary writes contributes to lowering the power dissipation in the register file.

2.3 Preventing unnecessary writes to register file

In order to preclude the unnecessary writes to the register file, information regarding whether a particular register should write to the register file needs to be identified *a priori*. As we are exploiting the forwarding hardware to prevent the unnecessary accesses to the register file, information regarding the liveness of the register is necessary to determine whether the destination register should be written to the register file. In the following sections we discuss issues related to register liveness and then we lay out our approach to handle the unnecessary accesses to the register file.

2.3.1 Register liveness analysis

This section outlines the register liveness analysis to be used in evaluating the necessity for the instruction to write its destination register to the register file.

```
I0:   add   r1, r2, r3
I1:   add   r4, r1, r5
I2:   load  r6, 0 (r10)
I3:   add   r9, r6, r8
I4:   add   r1, r6, r4
I5:   sub   r6, r7, r9
```

Figure 2.2: Example of segment of instructions

As forwarding is limited by the forwarding span, we use register liveness as a metric in ascertaining the necessity for writes to the register file. We denote this metric as *register liveness distance*, *RLD*. This metric represents the distance between the definition of the register and its last use in terms of number of instructions and stalls that can be precisely determined at compilation time. We use an example to clarify the register liveness distance concept. Figure 1 shows a fragment of code for 6 instructions. We assume that this code would be executed on the 5 stage pipeline (assuming that all the ALU instructions can be executed in one cycle and that accessing the data cache takes one cycle as well). In this example the destination of instruction I0, r1, has an RLD of 1 since I1 is the last instruction that uses its value before its new assignment in I4. Regarding the destination register of I2 which is r6, it has an RLD equal to 3 since the load instruction induces one stall. In both of these cases the forwarding network is capable of bypassing the results of I0 and I6 to all their dependent instructions. Consequently, no need for these instructions to write to the register file in the WB stage exists. The observation that can be made here is that unnecessary writes take place in the cases when RLD fails to exceed the forwarding distance, FD. However, this rule could be violated due to possible cache misses, exception events or the presence of conditional branches in the path between the instruction and its dependents. In the following discussion, we illustrate the impact of these issues:

- **Instruction and data cache misses:** The occurrence of such events could induce stalls that could increase the RLD beyond the value of FD, an issue that leads to possible correctness problems if the register file is not updated.

An illustrative example can be found in the I-cache miss at I3 in the code in Figure 2.2. The occurrence of such an event causes the *RLD* of r6 in I2 to increase to 4. In this case prohibiting I2 from writing its results in the register file leads to an erroneous execution since I4 would read the wrong value of r6. The simple solution to this problem is to ignore the write prohibition in such a class of cache misses. This solution is effective in light of the fact that cache misses are typically low.

- **Exception events:** Similar to the case of cache misses, the occurrence of exception events increases the *RLD* value of the register. Since the occurrence of the exception is a rare event, the simple solution is to ignore the write prohibition in the cases of exception.

- **Conditional branches:** The presence of conditional branches between the instruction and its dependents impacts the register liveness in two ways. The first impact is that the *RLD* for the case of the branch being *taken* could be quite different from the case when it is *not taken* which could lead to a nondeterministic situation. Another problem that is associated with branches is the use of branch prediction to improve performance. In cases of branch misprediction, the pipeline needs to be flushed and filled with the correct path. This issue impacts the *RLD* of the registers that belongs to the instructions that precede the branch. As the complexity that is associated with handling register writes across basic blocks is nontrivial, handling such cases is likely to result in diminishing returns. Considering that, we intend to focus on the cases when the liveness of the register is within the basic blocks only.

Overall, utilizing register liveness analysis is an efficient way to identify unnecessary writes. The dynamic behavior in terms of cache misses and exceptions needs to be considered to ensure the correctness of the execution.

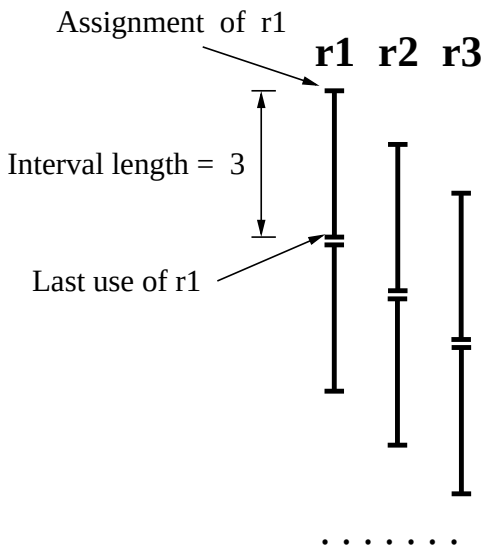


Figure 2.3: Three registers are enough to cover all possible assignments with FD=3

2.4 Register based encoding scheme

The analysis that we have provided in the previous section indicates that unnecessary writes could be eliminated in the cases when $RLD \leq FD$. We denote these cases as *short interval* cases. Instead of assigning a write permission bit to each instruction, we propose preserving a subset of registers to cover all short live variables in the code. We denote such a class of registers as *short interval registers*. In the following we show that the set of short interval registers is typically quite small. It can be inferred then that the overhead of applying such a scheme in terms of power and complexity is negligible.

In order to determine the upper bound for the number of registers that need to be preserved for forwarding, we use static liveness analysis for this purpose. We denote the set of short interval registers as *SSIR*. From the inequality $RLD \leq FD$ that we provided earlier, we can see that the upper bound for the RLD is equal to FD. Considering that, one can conclude that $|SSIR| \leq FD$. In the following we outline the rationale for this inequality through an illustrative example. The example in Figure 2.3 shows the assignment of registers that leads to the worst case in terms of the number of registers that need to be preserved for a case when

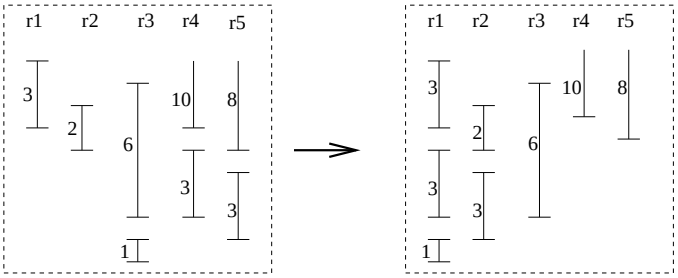


Figure 2.4: Minimizing the size of preserved registers through renaming for RLD=3

$FD = 3$. The vertical axis corresponds to a sequence of instructions in a basic block (we assume all the stalls that can be precisely determined at compilation time are changed into *NOOP* instructions). Each vertical line in the figure represents the live interval for a certain register. In order to represent the worst-case scenario, we need to have an interleaved interval each with length equal to FD. As can be seen in the figure, the worst-case scenario necessitates 3 registers since each register can be reused after its prior interval expires. As the value of FD is small in practice, one can expect that all unnecessary writes could be eliminated.

In a typical case, the compiler assigns the registers to the variables from the pool of free registers. Consequently, arbitrary intervals could be assigned to each register. In order to apply this approach efficiently we need to compact the short live registers into a small subset of size FD at most. Interestingly, such a compaction can be attained efficiently at the compiler level through the use of a renaming scheme approach. To clarify the idea of renaming, we use an illustrative example. Let's assume the live interval distribution for a certain code shown in the left of Figure 3. Also let's assume that r1 and r2 belong to $SSIR$. Considering that, one can observe that the short intervals in r3, r4, and r5 can not be exploited for low power despite the fact that they satisfy the inequality $RLD \leq FD$. In the right part of Figure 2.4, we use a renaming to move these intervals into r1 and r2. Interestingly, performing renaming at the compiler level comes at no overhead in terms of power, performance and hardware, an issue that enhances the relevance of this approach.

In order to perform renaming, two cases need to be considered. The first

one handle the cases when there is sufficient number of registers to cover all short intervals assignments while the other manage the cases when there is insufficient number of registers for all short intervals assignments. In the following, we outline the details of the renaming algorithms for these two cases.

2.4.1 Full register renaming algorithm

In this section we solve the renaming problem when there are enough registers that can be dedicated for the $SSIR$ to cover all short intervals in the given code. In order to attain such a renaming we suggest the use of the Left-Edge algorithm, LEA, that has been proposed in [30]. Although this algorithm has originally been proposed for channel routing, it fundamentally solve an analogous problem where the channels correspond to the short interval registers and the trunks of the net correspond to the live interval of registers. Following the basic ideas in LEA we sort the given short term intervals in ascending order relative to the y coordinate at the top point of the intervals. Subsequently, the algorithm allocates a short term register to each of the intervals, one at a time, based on the sorted order. In allocating an interval to a register, the algorithm follows the concept of first fit. This algorithm exhibits a linear time complexity.

2.4.2 Partial register renaming algorithm

The aim in this section is to handle the cases when there is an insufficient number of registers in covering all possible short time intervals in the given code. Considering the constraints in the problem, one needs to optimize the use of the available registers. In order to attain the renaming under such constraints we propose extending LEA to handle the possible conflicts in the assignments. In the new algorithm, we assign a weight to each interval that reflects the power savings. The difference between the new algorithm and LEA is in handling the cases when the algorithm determines that there is no room for a certain interval. In such cases we compare the weight that is associated to the unallocated interval to the weights that are associated with the intervals that are creating the conflict.

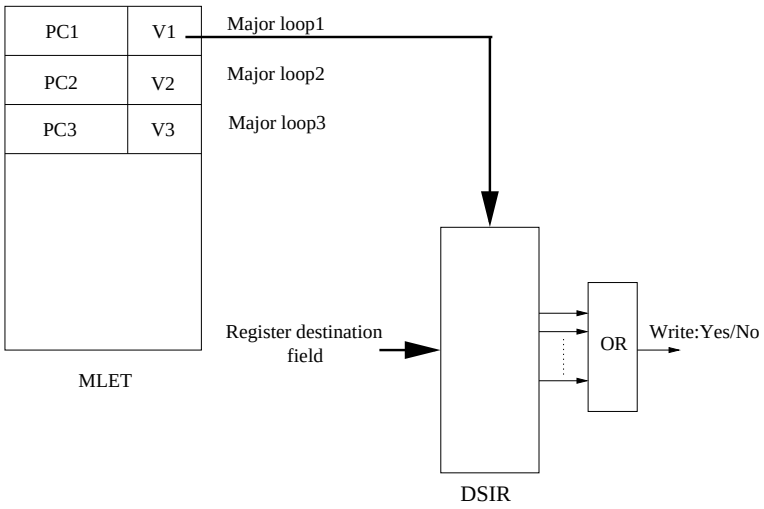


Figure 2.5: Hardware support

The interval with the least amount of weight would be evicted and returned to its original allocation. The complexity of the new algorithm is also linear. The weight function is shown in the following:

$$w_i = W_i P_i / (D_i + 1) \tag{2.1}$$

where w_i corresponds to the assigned weight for interval i, W_i, corresponds to the power dissipation for writes to the register file, P_i represents the probability of no occurrence of cache misses or exceptions that lead to the assertion of the writes for short interval registers, and D_i corresponds to the distance from the conflict point to the end of the interval. The numerator in the weight function represents the amount of power savings when the write is prohibited. As the allocation of long intervals leads to lower utilization of the registers, we divide the amount of power savings by the length of intervals to optimize the allocation that is generated through LEA.

In order to improve power savings in the cases when there are not enough registers for $SSIR$, one can utilize the fact that a typical application consists of relatively few major loops [12, 49, 50] and apply the renaming algorithms at the granularity of these loops. Thus the loops that are not the bottleneck can allocate more registers in $SSIR$ which leads to improved power savings.

2.5 Hardware support

The hardware architecture of the proposed implementation is presented in Figure 2.5. The major loop encoding table (MLET) stores the PC's for the entries of the major loops and the vector, V, that has the write permission information for the set of short interval registers. As the maximum size of the short live register span is known a *priori*, this set of registers can also be determined in advance. In this case, each bit in the vector V corresponds to a prespecified register. The size of each vector is FD which is typically quite small, e.g. 4 bits for a 5 stage pipeline. The elements of the vector V can be stored in a latch circuit to avoid the associative access to MLET. In cases where all the major loops exhibit enough registers to be dedicated for the set $SSIR$, it suffices to access MLET only once at the beginning of the application. Otherwise, MLET could be accessed only at the beginning of the major loops. The access to MLET could be signaled through executing a certain sequence of instructions. In order to capture whether the incoming destination register belongs to the set $SSIR$, we use a decoder to capture the elements of $SSIR$. We denote this decoder as the Decoder of the Short Interval Registers, $DSIR$. The size of this decoder is small since it has only FD outputs. In this decoder the elements of the vector V are AND'ed with the decoder primary outputs to embed the write permission information. The set of FD outputs of the decoder are OR'ed to determine whether the incoming register should write to the register file. In this case, if the output of the OR gate is *True*, then the write would be prohibited. In the case of cache misses or exceptions, the register *should* write to the register file. In the next sections we show that the power overhead that is associated with this scheme is insignificant, an issue that improves the applicability of our work.

Table 2.1: Processor parameters

Execution model	inorder
Issue width	1
Register file	32 registers
Branch predictor	2048 entries, *gshare*
BTB	256 entries, 1 way
L1 I-cache	8KB, 2 way
L1 D-cache	8KB, 2 way
L2 unified cache	128KB, 4 way

Table 2.2: Technology and packaging

Processor clock	1.5GHz
Technology	130nm
Processor die size	4.4mm × 2.5mm
Die thickness	0.2mm
Heat spreader	5mm × 5mm × 1mm
Heat sink	3cm × 3cm × 0.5cm
Local ambient temperature	45 ^{o}C

2.6 Experimental Evaluation

2.6.1 Methodology

In this section we present a set of experimental results to assess the effectiveness of using the proposed scheme in delivering power savings and reducing temperature of the register file. We use an illustrative example of a 5 stage single issue MIPS like processor. The register file is composed of 32 registers of 32 bits each. It has three ports, two for read and one for write. The details of the processor configuration is given in Table 2.1. We utilized three simulators, SimpleScalar [10], Wattch [18], and HotSpot-4.0 [57]. SimpleScalar simulator is used to obtain the architectural level performance simulation. SimpleScalar results are fed into Wattch to obtain the power values of the processor functional units. The power values are then used to estimate the energy consumption in register file as well as the temperature through the HotSpot simulator. Figure 2.6 shows the floor plan of the processor we study. The details of the processor size and packaging are given in Table 2.2.

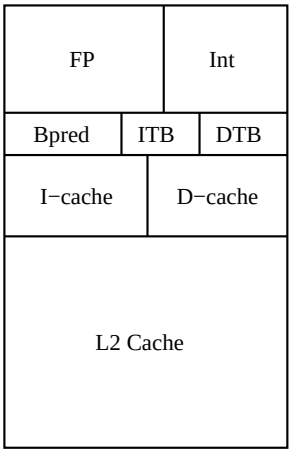


Figure 2.6: CPU floor plan

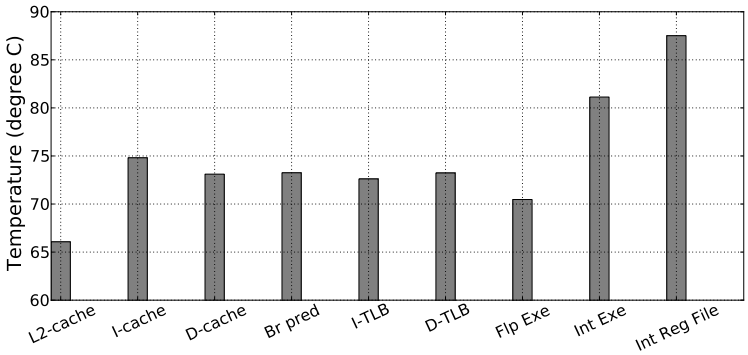


Figure 2.7: Temperature distribution of functional units

The workload that we use includes a set of five benchmarks from various DSP and numerical applications, ones typically used in application specific processors. The benchmarks used are: fast discrete cosine transform ($fdct$) DSP kernel; matrix multiplication ($mmul$) on a matrix of size 50x50; successive over-relaxation (sor); a tridiagonal linear system solver (tri) on a matrix of size 128x128; and extrapolated Jacobi-iterative method (ej) on a 128x128 grid.

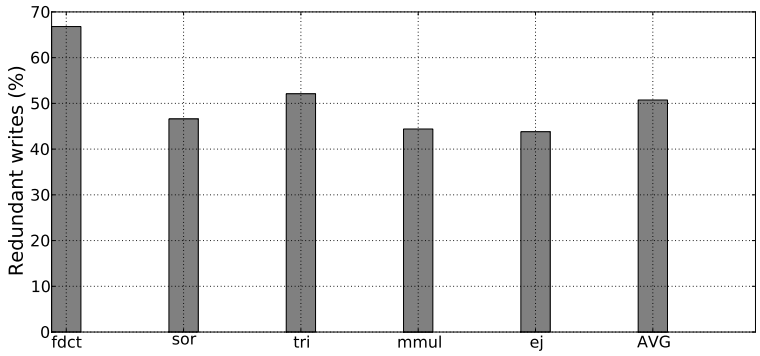


Figure 2.8: Percentage of short live registers: RLD (1-3)

2.6.2 Results

Figure 2.7 shows the temperature distribution of the functional units. Temperatures value are collected by averaging the resultant temperatures from running the set of benchmarks one at a time. The results clearly show that the register file is the hottest component in the processor. Figure 2.8 depicts the percentage of the registers that have a liveness distance in the range of 1-3. The results show that 50% of the register assignments on average have liveness distance ≤ 3. These results indicates that a significant percentage of writes can be eliminated by utilizing our approach. The power overhead of the supported hardware is negligible, less than 1.0%, as the decoder DSIR can be built using only few gates. Additionally, the latch circuit that stores the vector V, also consumes negligible power since its state rarely changes. The resultant net energy savings in register file are shown in Figure 2.9. These results depict appreciable savings could be attained by applying the proposed technique which reaches 22.3% on average. The savings in the case of *fdct* is the highest since this workload has the largest amount of redundant writes.

Figure 2.10 shows the temperature of the register file with and without our technique. The results show that preventing redundant writes can reduce the temperature of register file by 4^oC on average and reduces the occurrence of hot spots by 60% with thermal threshold of 85^oC. The maximum reduction in temperature of 6^oC occurs in the case of *fdct* since it has the highest number of

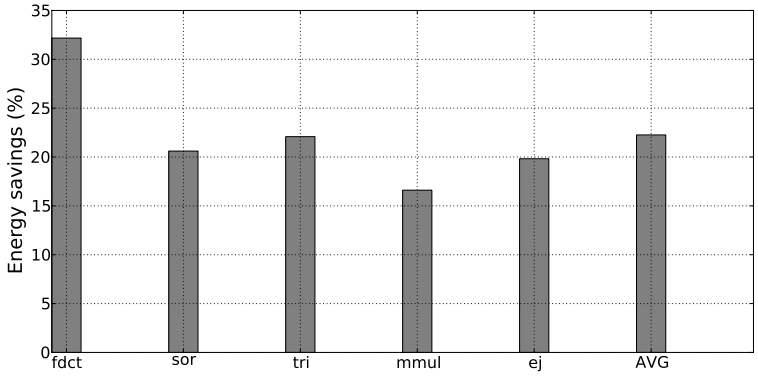


Figure 2.9: Percentage of energy savings in register file

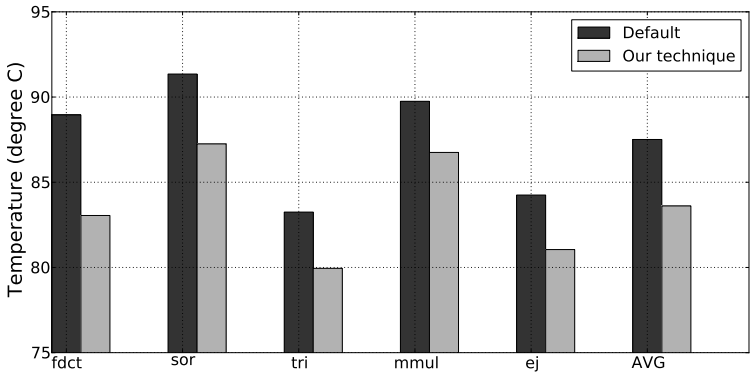


Figure 2.10: Temperature reduction in register file

redundant writes. On the other hand, the temperature reduction of in the case of *mmul* is lowest since it has fewest redundant writes. These results indicate that we can reduce the temperature appreciably at almost no performance overhead.

2.7 Conclusion

In this chapter we have presented an approach for reducing the temperature of register file while delivering energy savings. We achieve this by minimizing unnecessary writes to the register file. This work shows that a significant portion of the writes are redundant due to forwarding network. We embed the redundant writes information in the application code which can be decoded easily by the

hardware. The encoding scheme skews the redundant writes information into a small set of registers that are prespecified *a priori* at the hardware through the use of renaming at the compiler level. We show that a small number of registers suffice to eliminate all redundant writes. Efficient algorithms are provided for the purpose of renaming. The size of the prespecified registers can be adjusted dynamically through the use of programmable hardware to handle any register pressure, thus preserving the generality of the of processors while applying the encoding methodology that we propose. The reported experimental results show that the proposed technique is able to reduce the occurrence of hot spots by 60% while achieving energy savings of 22.3% on average at almost no performance overhead.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the International Conference on Computer Design, 2007. Ayoub, R. and Orailoglu, A. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Proactive Thermal Management for General Purpose Multicore Processors

In the previous chapter we showed that the microarchitectural level provides opportunities for reducing temperature by eliminating redundant activities in the register file. However, microarchitectural solutions have limitations as they are restricted to local optimizations and they are not suitable for implementing software like algorithms that are required for thermal management. This motivates us to introduce a software solution at the operating system layer that manages the temperature of the entire cores within a CPU socket.

To manage the high temperature within a single CPU socket, a number of reactive dynamic thermal management, DTM, techniques have been proposed. These techniques resolve the temperature problems upon reaching undesirable levels. However, this approach is not efficient in creating a better thermal balance in the chip and reducing leakage power. In contrast, when thermal management can predict temperature then it can act before reaching thermal emergencies to prevent them when possible. The proactive approach has the potential to improve the temperature distribution and at the same time reduce the leakage power.

In this chapter we introduce an novel proactive thermal management scheme to reduce the thermal hot spots between cores. In order to predict temperature

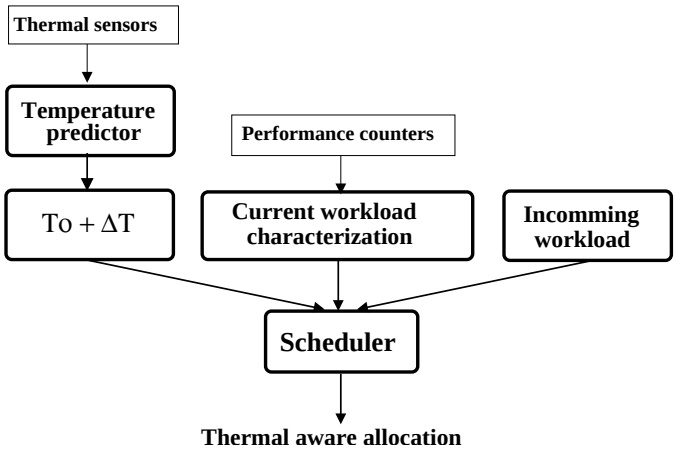


Figure 3.1: Thermal aware scheduling for the core level

accurately at negligible cost we present a new temperature predictor, called Band-Limited Predictor (BLP) that is based on the band limited property of the temperature frequency spectrum [39,41,42]. The important feature of our predictor is that the prediction coefficients can be identified accurately at the design stage which makes it workload independent. Our results show that applying our algorithm considerably reduces the average system temperature, hottest core temperature, and improves performance by 6 ^{o}C, 8 ^{o}C and 72% respectively.

3.1 Overview of proactive thermal management

Forecasting thermal problems and managing them ahead of time can dramatically reduce performance overhead and temperature. In this chapter we target an SMT multicore processor. Figure 3.1 depicts the operational framework of the proposed approach. The OS scheduler employs proactive thermal management to minimize the impact of temperature. To predict the core temperature efficiently we propose a fundamentally new temperature predictor that is based on the concept of predicting band-limited signals which requires no training phase. The predictor coefficients are estimated at the design time based on the temperature spectral limited bandwidth. Moving history of the temperature is fed to the suggested predictor to estimate the future temperature, $T_0 + \Delta T$ where T_0 represents the current

temperature. Temperatures of the cores are collected using thermal sensors that are commonly available in the state-of-the-art processors and can be accessed easily by the operating system. Workload characterization is done to estimate the contribution of the individual threads to the core heating. To extract the activity of the individual threads we use performance counters that are typically available in current processors. The last input to the OS scheduler are queues that contain the tasks that are waiting to be executed (incoming workload). The OS scheduler calculates the predicted temperature of the individual cores at each scheduling period. If predicted temperature of one of the cores surpasses a given temperature threshold, the scheduler migrates a portion or all of the workload of the hot core to the core that is predicted to be the coolest. The migration overhead is only a few microseconds which is much lower than the cost of putting the processor in a low power mode [40]. Migration cost comes primarily from warming up the caches and OS overhead [40]. Workload characterization is used to improve the efficiency of migrations in the cases when only portion of the running threads are allowed to be migrated. In handling the incoming threads, the scheduler assigns them to the core that is predicted to be the coolest. In the subsequent sections we elaborate on the design of our suggested approach.

3.1.1 Temperature prediction

Our new temperature predictor is used as an input to the core level scheduler as depicted in Figure 3.1. The basic idea of the temperature predictor is that the band-limited signals can be predicted from the previous samples using prediction coefficients that are *independent* of the particular signal or autocorrelation function [39,41,42]. Prediction coefficients are function of the signal bandwidth only. Before going into the details of our band-limited predictor, we show that the temperature frequency spectrum is band-limited in nature.

The thermal model that present show that the temperatures of the individual die components (e.g. cores, L2 cache) is a simple *low pass RC* filter with horizontal thermal resistances connecting the adjacent units. The bandwidth of temperature frequency spectrum can be computed using standard RC network

analysis or using CAD tools, e.g. HSPICE. For high end CPUs, we can neglect the effect of the horizontal thermal resistances to simplify the bandwidth computations since their values are much higher than the vertical ones [33]. This indicates that the temperature signal has a limited frequency bandwidth, the frequency spectrum of the individual components can be modeled as follows:

$$\frac{T(w)}{T(0)} = \frac{1}{\sqrt{1 + (w\tau_c)^2}} \tag{3.1}$$

where $T(w)$ represents the temperature value as a function of the angular frequency w, and τ_c is the core temperature time constant that equals RC. Given that the temperature is a low pass filter, it satisfies the band-limited condition. The author in [39] shows that band limited signals can be predicted using the following linear formula:

$$x(t) = \sum_{n=1}^{N} a_n x(t_n) \tag{3.2}$$

where a_n are the prediction coefficients, t_n represents the nth time sample and N is the total number of samples. For the case of uniform sampling, the value of t_n can be written as $t_n = t - nd_s$ where d_s is the sampling period. For this predictor to be applicable, the error needs to be bounded and sufficiently small. The absolute error is expressed as $\epsilon = |x(t) - \sum_{n=1}^{N} a_n x(t_n)|$. Using Paley-Wiener theorem and Schwarz formula, a bound on the error ϵ^2 can be found as follows [39]:

$$\epsilon^2 \leq \{\int_{-W}^{W} |X(f)|^2 df\}\{\int_{-W}^{W} |ds(f)|^2 df\} \tag{3.3}$$

where $ds(f) = e^{i2\pi ft} - \sum_{n=1}^{N} a_n e^{i2\pi f t_n}$, f is the frequency and W is the frequency bandwidth in Hertz. The first part of this equation represents the signal energy while the second part is the amount of prediction error. The equation (3.3) can be rewritten as:

$$\epsilon^2 \leq \|x\|^2 \cdot \epsilon_I \tag{3.4}$$

where ϵ_I is the error component while $\|x\|$ is the signal energy. The error integral represents an *N-dimensional* function $E(a_1, a_2, ..., a_N)$ of prediction coefficients.

The BLP coefficients can be obtained by minimizing ϵ_I. To obtain the optimal coefficients we use the method of *eigenvector optimization* [39]. The work in [43] shows that the error integral part ϵ_I can be rewritten as:

$$\epsilon_I = v^T \Omega v \tag{3.5}$$

where v is a vector that have $N+1$ elements. To satisfy this equation, it is imperative for the vector v to have its first entry equals to 1 ($v_1 = 1$). The matrix Ω is expressed as:

$$\Omega = \begin{pmatrix} 1 & \mathbf{b}^T \\ \mathbf{b} & D \end{pmatrix}$$

where $\mathbf{b}$ is a vector of N components and D is a matrix of $N \times N$. The vector $\mathbf{b}$ and the matrix D can be constructed from the results of applying the standard method of minimization for $E(a_1, a_2, ..., a_N)$, [42], to the error that gives the following system of equations:

$$\sum_{n=1}^{N} a_n sinc(2W(t_j - t_n)) = sinc(2W(t_0 - t_j)) \tag{3.6}$$

where j = 1,...,N and $sinc(t) = sin(\pi t)/(\pi t)$. The matrix D equals the system matrix in (3.6) and the vector $\mathbf{b}$ equals the vector that is in the right hand side of (3.6). As a result, the matrix Ω can be expressed as:

$$\Omega = \begin{pmatrix} s(t_0 - t_0) & s(t_1 - t_0) & s(t_2 - t_0) & ... & s(t_N - t_0) \\ s(t_0 - t_1) & s(t_1 - t_1) & s(t_2 - t_1) & ... & s(t_N - t_1) \\ s(t_0 - t_2) & s(t_1 - t_2) & s(t_2 - t_2) & ... & s(t_N - t_2) \\ ... & ... & ... & ... & ... \\ s(t_0 - t_N) & s(t_1 - t_N) & s(t_2 - t_N) & ... & s(t_N - t_N) \end{pmatrix}$$

where $s(t) = sinc(2Wt)$. The matrix Ω is symmetric since $sinc(t)$ is an even function. In addition, this matrix is positive definite which makes all the eigenvalues positive. The optimal prediction coefficients that minimize the prediction error

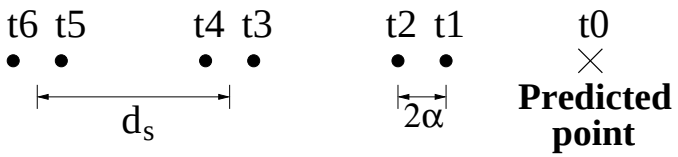


Figure 3.2: Position of interlaced sampling points

ϵ_I can be obtained by minimizing the value of $v^T \Omega v$ across all the possible vectors with $v_1 = 1$. For the set of the normalized eigenvectors with length 1, the minimum value of $v^T \Omega v$ is determined by the smallest eigenvalue of matrix Ω. In other words, we need to find the eigenvector that is associated with the smallest eigenvalue, λ_{min}, then normalize this eigenvector to make the first entry equals to 1. Lets assumes that V is the eigenvector that is associated with the smallest eigenvalue with $\|V\| = 1$. For this selected eigenvector, the value of $V^T \Omega V$ is equal to λ_{min}. The normalized eigenvector, V_{norm}, can be obtained simply by dividing the eigenvector by the vector's first element, $V_{norm} = \frac{V}{V_1}$. The important result is that we can extract the set of prediction coefficients, $\{a_1, a_2, ..., a_N\}$, directly from V_{norm} as [39]:

$$a_i = V_{norm_{i+1}} \tag{3.7}$$

The significance of equation (3.7) is that the optimal prediction coefficients depend only on the signal bandwidth, W. For the case of uniform sampling, the upper bound for prediction distance, d_p, can be obtained using Nyquist condition, $2d_pW < 1$. The temperature spectral bandwidth depends on the value of τ_c as shown in (3.1). Extracting τ_c can be simply accomplished at the design time using die layout and thermal package parameters [56].

The temperature typically changes slowly as a function of its thermal time constant. As a result, extending the prediction window allows for better thermal management as more thermal emergencies could be captured and prevented ahead of time. The prediction window can be extended by employing nonuniform sampling. The theoretical work in [45] shows that if a signal is sampled at $1/m$ times the Nyquist rate, but in each sampling interval not one but m samples are used,

the signal can be reconstructed completely. In [39], the author show that this concept can be extended to increase the prediction window. To apply this concept, the samples need to be placed in interlaced pattern. For $m = 2$, the placement of samples should follow a pattern of two close samples that is followed by a uniform delay, then two more close samples are taken and so on. Figure 3.2, shows the locations of interlaced sampling points. The theoretical proof is given in [39]. The prediction distance is computed as $d_p = d_s - 2\alpha(m-1)$ where α is the half distance between the two close samplings.

In summary, to predict the temperature, we first compute the coefficient factors at the design time using (3.7). At run time, we collect the temperature samples and apply the simple polynomial given in (3.2) to predict the temperature. The overhead of our prediction is negligible since computing the prediction polynomial can be achieved in a few CPU cycles.

3.1.2 Proactive Thermal Management

In this work we are targeting a multicore platform where each core can execute multiple threads e.g. SMT core that we show in Figure 3.1. Our proactive policy performs thermal aware scheduling through either migrating the hot threads across the system cores or scheduling the incoming jobs in a thermally aware manner. To initiate the migration process, the predicted temperatures of the hot cores must surpass the temperature threshold and there has to be a sufficient difference in the predicted temperature between the hot and coolest cores. We use the temperature time constant to calculate the temperature rise that corresponds to one scheduling period. The temperature gap should be greater than or equal to this temperature rise value. When the coolest core is idle, we migrate the most active thread from the hot core to the colder one. If the coolest core is executing, we select the thread with modest activity from the hot core and migrate it to balance the temperature. We determine the activity of the individual threads (e.g. hot or cold) running in the same core by using *fetch rate* as a run time workload characterization mechanism. The fetch rate can be measured directly through the use of processor performance counters (e.g. Model Specific Registers MSR). The

Table 3.1: Processor simulation parameters

Parameter	Value
Issue width	4
Number of threads	2
ROB	128
Functional units	4 IntALU, 1 IntMult/Div
	1 FPALU, 1 FPMult/Div
Branch predictor	Tournament 2048 local predictor
	8192 global predictor
BTB	2K entries, 1 way
LSQ	32
L1 I-cache	32KB, 4 ways, 32B blocks, 1 cycle
L1 D-cache	32KB, 4 ways, 32B blocks, 1 cycle
L2	4MB, 8 ways, 64B blocks, 12 cycles
Memory latency	200 cycles

fetch rate statistics are collected for the duration of one sampling period. When assigning the incoming jobs, we use the result of prediction to find the coolest cores with available resources.

3.2 Evaluation

3.2.1 Methodology

For the experimental evaluation we assume a platform that consists of 6 cores, where each core is a 4-issue SMT ALPHA like processor that can run at most 2 threads. Table 3.1 gives the simulation parameters that we have used in our simulations. We utilized three simulators, M5 [44], Wattch [18], and HotSpot-4.0 [57]. The M5 simulator is used to obtain the architectural level performance simulation. The M5 results are fed into Wattch to obtain the power values of the processor functional units. The power values are then used to estimate the temperature through the HotSpot simulator. Figure 3.3 shows the floorplan of our processor.

To account for CPU cores leakage power temperature dependency, we used

the second-order polynomial model that is proposed in [29]. We extracted the model coefficients empirically based on the given normalized leakage values. To estimate the leakage values for 65nm technology we incorporate the reported value of leakage power density ($0.5W/mm^2$ at 383K [17]) in the second-order polynomial model. In our results we also account for the temperature warm up, as the heat sink has a longer time constant than the die.

We use benchmarks from the SPEC2000 suite for our workload. We selected a set of benchmarks that exhibit various levels of thermal stress to represent real life applications. We ran each benchmark for a representative interval of 4 seconds. Such time is sufficient to evaluate our policies since it is orders of magnitude larger than the die thermal time constant. As we are assuming an SMT multicore platform we constructed a representative set of benchmarks that are shown in Table 3.3. The selected benchmark combinations varies in terms of CPU utilization and thermal stress, to better evaluate our policies under various workload conditions.

Our *Proactive Thermal Management that uses Band Limited Predictor PTM-BLP*, reduces the occurrence of hot spots between cores. The thread migration in this policy is allowed to be initiated only when the temperature difference between the hot core and colder core is 5 ^{o}C to avoid ping-pong scenarios. For the predictor parameters we use $\alpha = 0.135$, $m = 3$ and N=3 based on the analysis in section 3.1.1. We compare PTM-BLP algorithm against the following set of state of the art policies:

Dynamic Load Balancing, DLB: It is usually implemented in modern operating systems to enhance the utilization of the system resources. The DLB performs thread migration to minimize the difference in task queue lengths of the individual cores [21]. The operating system initiates dynamic load balancing every hundred milliseconds. The balancing threshold is set to one to keep the difference between queues equal to zero as possible. In this work we implement the DLB as our *default policy* for the purposes of comparison.

Reactive Thermal Management, RTM: This policy migrates the threads from the cores that reach the temperature threshold to colder ones when possible. The temperature threshold for activating migration is set to slightly below

Table 3.2: Technology and package characteristics

Technology	65nm
Processor clock	2.0 GHz
Local ambient temperature	45 ^{o}C
Core area	3.3 × 3.3 mm
CPU thermal threshold	85 ^{o}C
Convective resistance	O.1 K/W,
Die thickness	0.2 mm
Heat spreader thickness	1.0 mm

the thermal emergency threshold (2 ^{o}C) to give this policy some time to react before reaching thermal emergencies. Same threshold is used for other policies that perform migration. Upon reaching the temperature threshold, the RTM selects randomly one of the threads and migrates it away.

Proactive Thermal Management using ARMA, PTM-ARMA: We also implement another proactive policy that utilizes the ARMA based temperature predictor proposed in [22] for the comparison purpose. Our implementation, PTM-ARMA, uses ARMA to predict the future temperature. If the predicted temperature is higher than the temperature threshold, the scheduler selects randomly one of the threads from the hot core and migrates it to the core that is predicted to be the coolest and has available resources. For forming the ARMA(p,q) model, we used the iterative approach as described in [22] and set the maximum order for p and q to 5 (p and q represent the order of the autoregressive and moving average part of the model respectively). At run time, we check if the prediction error exceeds 5 ^{o}C, and if it does we initiate adaptation phase and form a new model. To avoid noncritical adaptations, the error is calculated by averaging the prediction errors in a window of 20 samples.

3.2.2 Results

The reported results show that our policy outperforms the other policies in minimizing the system average temperature as well as the temperature of the hot cores while minimizing performance overhead. In the following we discuss the

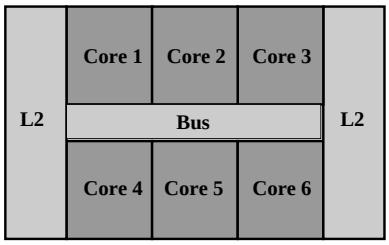


Figure 3.3: Processor floorplan

Table 3.3: Benchmark combination list

Test group	Benchmarks
1	4× bzip2
2	5× bzip2
3	3× bzip2 + 3×gzip
4	3× crafty + 3× swim
5	3× bzip2 + 3× gzip + 1× gcc
6	3× crafty + 3× swim + 1× gcc
7	3× bzip2 + 3× gzip + 2× gcc
8	3× bzip2 + 3× gzip + 3× gcc

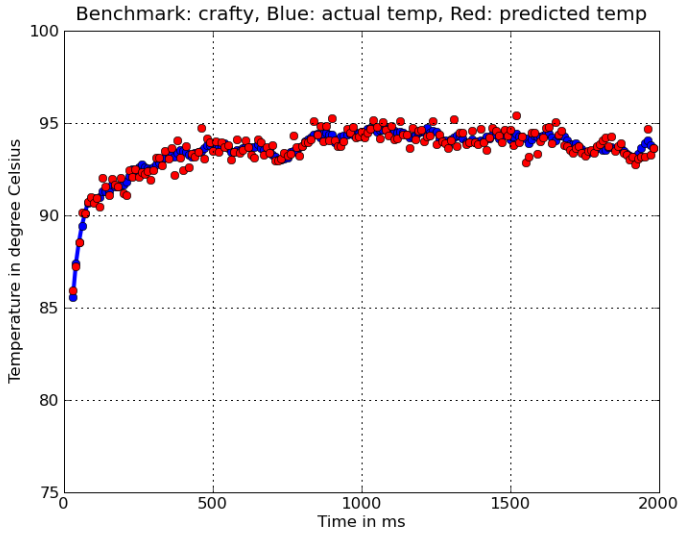


Figure 3.4: Temperature prediction of crafty

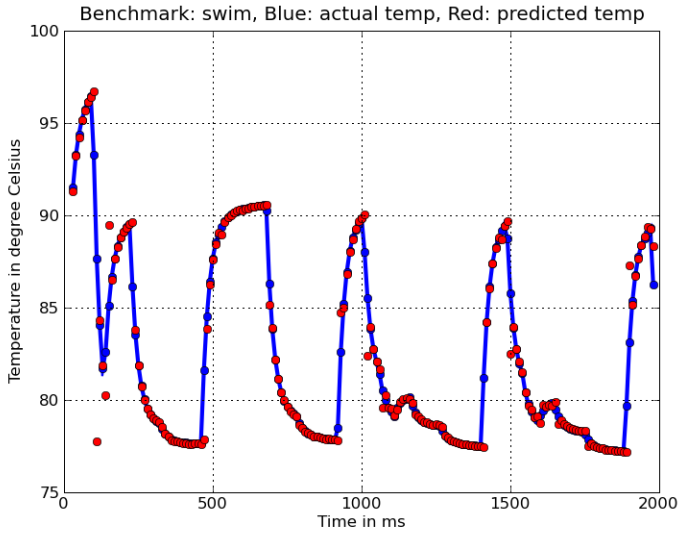


Figure 3.5: Temperature prediction of swim

details of the results.

We start with evaluating our band-limited predictor. We fit coefficients of our predictor based on the design parameters and don't change the predictor for the following experiments. In order for the predictor to be feasible, we need to show that the prediction window is in the range of operating system scheduling period. In Figures 3.4, 3.5 we show an illustrative examples of predicting the temperature using our predictor with executing crafty and swim benchmarks respectively. In these experiments we run one instance of the benchmark in each core and set the prediction distance to 10ms. It is clear from the results that our predictor is fairly accurate in predicting most parts of the signals despite their variations; the average of errors are 0.33 ^{o} C and 0.35 ^{o} C for crafty and swim respectively.

Figure 3.6 shows the die *average temperature* when using the DLB, RTM, PTM-ARMA, and PTM-BLP policies for all the workloads given in Table 3.3. It is evident from the results that using PTM-BLP results in a considerable improvement over the DLB, RTM and PTM-ARMA; the enhancement is as high as 8.8 ^{o}C, 8.4 ^{o}C, and 8.16 ^{o}C respectively. It provides appreciable savings over the DLB since it is able to prevent the overheating at lower temperatures and distribute the heat

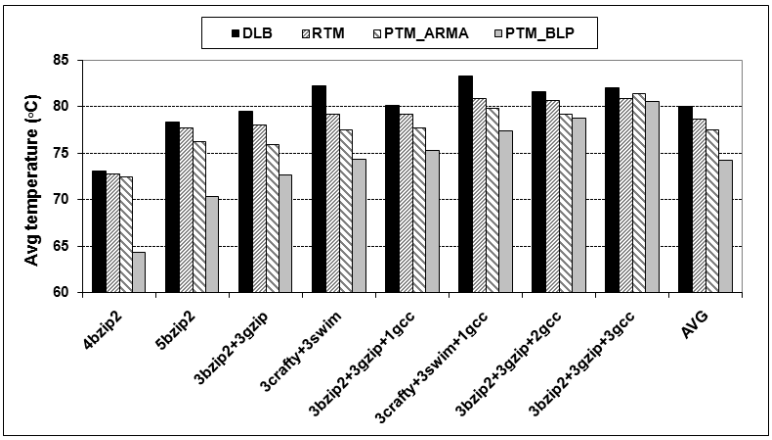


Figure 3.6: Lowering average system temperature

more evenly across the cores. The additional source of savings comes from lowering the impact of exponential dependency of leakage power on temperature since preventing the overheating at earlier point reduces the effect of the dependency positive loop. As can be seen from the results, the savings in the case of $4bzip2$ and $5bzip2$ are higher than $\{3bzip2 + 3gzip + 2gcc\}$ since there are more opportunities for migrating the threads across the system in the prior cases. Interestingly, our scheme is capable of providing savings even when the number of threads exceeds the number of cores, (e.g. $\{3bzip2 + 3gzip + 1gcc\}$). The PTM-ARMA outperforms the reactive policy (RTM) since the RTM acts only when the temperature becomes close to the overheating point. More interestingly, our scheme shows considerable improvement even over the proactive policy, PTM-ARMA. This could be attributed to the impact of prediction inaccuracy during training phases as the PTM-ARMA defaults to its current model until the training phase is complete during such events. It should be noted that training phase may be a frequent event. In addition, the ARMA training phase can take up to several hundred milliseconds [22], which is expected to increase the overall power consumption and the temperature in turn. Beside that, periods with frequent thread migrations can cause the temperature dynamics to vary and reduce the prediction accuracy. The other factor that contributes to the reduction in the average temperature is the use of runtime workload characterization that identifies activity of the individual

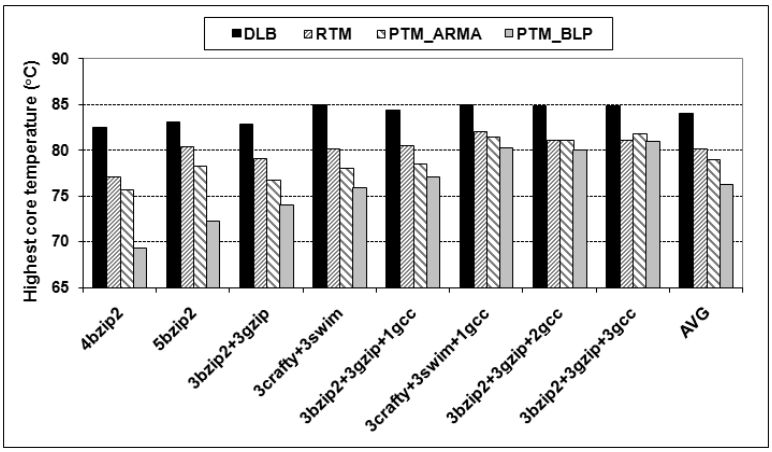


Figure 3.7: Minimizing core's highest temperature

threads and allocates them in a thermally sensitive manner.

Figure 3.7 shows the benefit of applying our scheme to minimizing the temperature level of the hottest cores in the system. It can be seen from the results that using PTM-BLP results in a strong improvement over the DLB, RTM and PTM-ARMA; the improvement can reach as high as 13.3 ^{o}C, 7.8 ^{o}C, and 6.37 ^{o}C respectively. The strong reduction in the hottest core temperature reflects a great temperature prediction accuracy of our method. The maximum benefit occurs in the case of $4bzip2$ since it has the highest opportunity of finding available hardware threads. For the case when the number of running threads is similar, the improvement of using our scheme over the RTM and PTM-ARMA is shown to be higher in the case of $\{3bzip2 + 3gzip\}$. This can be related to the higher temperature average of $\{3crafty + 3swim\}$ which lowers the chance of passing the temperature difference threshold that is required to avoid the ping-pong effect. Our scheme outperforms the PTM-ARMA by 2.9 ^{o}C degrees on average.

Figure 3.8 illustrates the percentage of performance improvement over the DLB when using the DTM policies that are discussed earlier. Our policy significantly outperforms DLB, RTM and PTM-ARMA; the enhancement is as high as 84%, 72% and 74% respectively. The DLB delivers the worst performance as it suffers the most from powering down overhead due to the frequent overheating. PTM-BLP surpasses RTM appreciably since the RTM could lead to more frequent

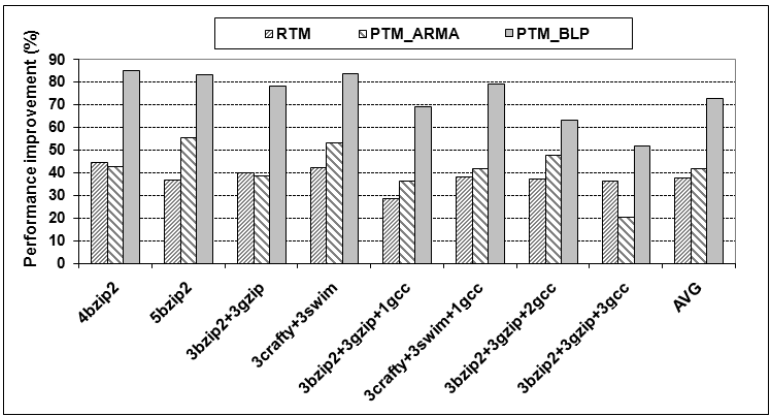


Figure 3.8: Performance improvement over DLB

power down events since the average temperature in the case of the RTM is higher than PTM-BLP (see Figure 3.6). The other reason is due to many migrations that are not helpful since RTM makes thermal management decisions based only on the current temperature. The results show that applying PTM-ARMA could result in high performance overhead that exceeds the level of the RTM. Example of that can be seen in the case of the workload $\{3bzip2 + 3gzip + 3gcc\}$. This is due to the training overhead and resulting inaccuracy due to frequent migrations. Including workload characterization in the scheduler as the PTM-BLP does, assists in improving the performance due to the enhancement in migration efficiency. Interestingly, our scheme outperforms the other techniques even in the case of high utilization. These results indicate that PTM-BLP is highly effective in handling thermal hot spots at various levels of system utilization.

3.3 Conclusion

3.3.1 Conclusion

In this chapter, we have introduced a new proactive dynamic thermal management technique for multicore system. Our algorithm incorporates continuous temperature prediction information and runtime workload characterization to guide the OS scheduler in allocating the workload in a thermally sensitive man-

ner. We also introduced a new temperature predictor that not only requires no runtime adaptation, but is also highly cost efficient. We provide detailed analysis on how to calculate the prediction coefficients at design time. We implemented our scheme and compared it against other state-of-the-art polices. The reported results show that our predictor is accurate where the average temperature error is below 0.5 ^{o}C despite the large variations in the temperature signal. Our results show that applying our algorithm considerably reduces the average system temperature, hottest core temperature, and improves performance by 6 ^{o}C, 8 ^{o}C and 72% respectively.

The limitation of our PTM-BLP is that it can not deliver savings in the cases when all cores are experiencing high thermal stress. The primary reason for this limitation is that the scope of managing the temperature is limited to a single CPU socket and does not take the opportunities of managing the temperature between multiple sockets. In the next chapter we study hierarchal approach where we perform core level as well as socket level thermal management to improve savings.

Chapters 3, in part, is a reprint of the material as it appears in Proceedings of the International Symposium on Low Power Electronics and Design, 2009. Ayoub, R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 3, in part, is a reprint of the material as it appears in IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems, 2011. Ayoub, R.; Indukuri, K. R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Thermal and Cooling Management in Multisocket CPU Servers

At any given speed setting the fan can dissipate only a limited amount of heat from a CPU. Increasing the speed requires exponential increase in fan power. This indicates that temperature solutions that act only within a socket are not sufficient to minimize cooling energy since some sockets may generate much more heat than others, resulting in dramatically higher cooling costs.

In this chapter we introduce a multi-tier algorithm that schedules the workload at the core and socket levels to minimize cooling energy and the occurrence of thermal emergencies. We schedule the workload between CPU sockets in a way that mitigates hot spots across them and reduces cooling energy. We developed a control theoretic framework for the socket level scheduling that guarantees the desired objectives, in terms of temperature, cooling energy savings and stability, are met. We add core level workload management that we discussed in chapter 3 to reduce the hot spots across the cores and improve cooling savings within a given socket. The reported results show that our multi-tier scheme is able to achieve cooling energy savings of 80% on average. The reported results also show that our formal technique maintains stability while heuristic solutions fail in this aspect.

In this study we address the limitations of the previously suggested thermal

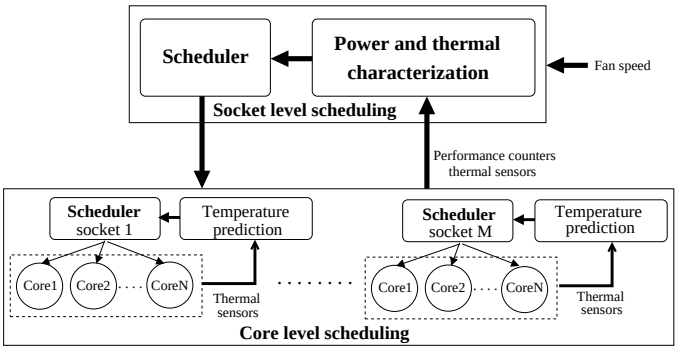


Figure 4.1: Overview of multi-tier scheduling

management techniques in single machines. Firstly, the scope of the prior techniques is limited to single sockets, hence they can not minimize hot-spots between sockets. Second, the temperature management within the single socket come with drawbacks that require improvements. The other important limitation is the lack of a realistic model of the cooling subsystem. Our work focus on mitigating the limitation in the prior work to reduce operational costs and to enhance performance. The main contributions of our work are summarized below:

- We design a new cooling aware multi-tier dynamic workload scheduling technique within a control theoretic framework to deliver high energy savings and ensure stability.
- We present a thorough evaluation and discussion of the proposed techniques that results in a substantial cooling costs savings of 80%.

4.1 Multi-tier thermal management overview

In general, the heat removed from the CPU by the fan is proportional to the fan speed. However, removing more heat requires exponential increase in cooling power. This means it is better to have the heat dissipation balanced between the CPUs to have a more uniform fan speed distribution and thus reduce the cooling energy. The other possible way to reduce cooling energy is to lower the average

combined fan speeds. This can be achieved by eliminating some of the hot spots and maximizing the use of fan capacity. Implementing these optimizations require intelligent job allocation between sockets. We can also reduce temperature by decreasing the power density on the die through migrating the workload from hot cores to cooler cores. This class of optimizations requires core level scheduling. This indicates that utilizing both core level and socket levels is necessary in order to maximize cooling savings.

These concepts motivate us to develop a multi-level thermal management (MTTM) technique where temperature is managed at socket as well as at core level. Figure 5.8 depicts the operational framework of our proposed approach.

Socket level: The scheduler at the socket level manages the jobs between sockets. It takes temperature, performance and fan speed information as an input. They are collected every during scheduling period. Cooling savings can be achieved by balancing fan speed or lowering the average combined fan speed. The techniques that achieve these two objectives are: (1) *Spreading*: This method focus one the cases when power is significantly unbalanced between sockets. It tries to balance the thermal hot spots across the sockets to generate a more uniform distribution in fan speed. (2) *Consolidation*: This technique is complementary to spreading where it optimize the situations where the sockets have balanced power distributions. It schedules the workload in a way that eliminates subset of the hot spots, an issue that reduce the average of combined fan speeds and deliver cooling savings. We design our scheduler within a control theoretic framework to ensure stability and to avoid exhaustive tunning as in the case of heuristic solutions. The scheduling period is on the order of seconds which incurs negligible overhead to the performance.

Core level: Core level is complementary to the socket scheduling where it reduces the temperature within each individual socket. The core level scheduler employs proactive thermal management that we discussed in chapter 3 to minimize the temperature. The scheduler calculates the predicted temperature of the individual cores and migrates the jobs from the hot cores to those that are predicted to be the coldest. This strategy helps reduce the occurrence of hot spots by

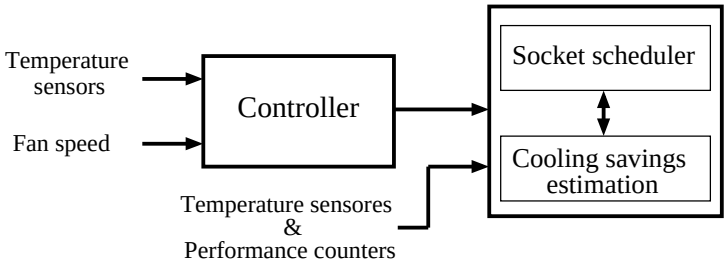


Figure 4.2: Overview of socket level scheduling

providing a better thermal balance between cores. The core level algorithm is invoked during the OS scheduling periods which is in the order of milliseconds. This core level scheduler should not impact the stability of the system since it reaches steady-state within a small fraction (several 10s of milliseconds) of the high level scheduler interval (several seconds). We show that the overhead of our technique is trivial due to the low time overhead of both predictions and migrations compared to the time scale of temperature changes. In the subsequent Sections we discuss both core and socket level algorithms in more details.

4.2 Socket level scheduling

We introduce our socket level scheduling to provide a better thermal distribution across different CPU packages. This provides additional savings on top of those obtained via core level scheduling. Figure 4.2 shows the overview scheduling at this level. It is composed of two primary stages: controller and scheduler. The controller determines how much power needs to be moved to/from sockets to minimize the cooling energy. The scheduler takes the inputs from the controller, thermal sensors and performance counters and tries to assign the workload based on the controller decisions. The scheduler communicates with a cooling savings estimator to quantify the savings of each decision and thus avoid ineffective scheduling events. To better understand this design we start with description of thermal and cooling models we used.

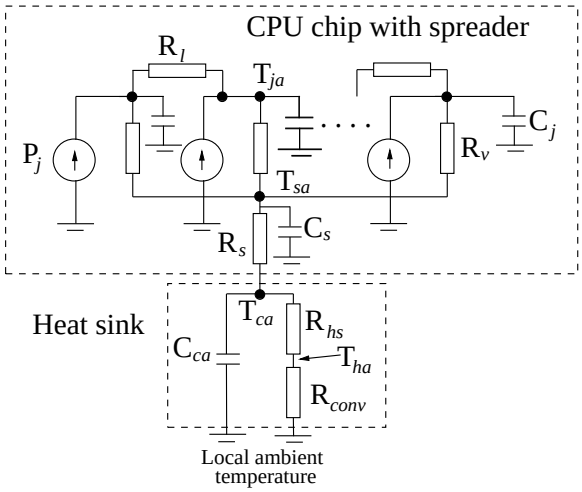


Figure 4.3: Single socket thermal model

4.2.1 Thermal and cooling model for CPU socket

Thermal modeling of the individual sockets can be done using an RC network similar to what is used in HotSpot [57]. Figure 4.3 shows the thermal model of the CPU chip with a thermal package. The thermal model of the CPU chip includes the die and the heat spreader. R_v is the die component's vertical thermal resistance including the thermal interface resistance. The lateral heat flow between the die components is modeled using a lateral thermal resistance R_l. However, the impact of the lateral heat flow can be ignored due to the high ratio of core area to die thickness in high performance processors [33]. C_j is the thermal capacitance of the components in the die and P_j corresponds to the power dissipated by the individual components. For the spreader, R_s and C_s refer to the thermal resistance and capacitance of the spreader respectively. The heat spreader is simplified to a single node because it behaves as an isothermal layer due to its high thermal conductivity. The time constant of the core is about 60 times smaller than that of the heat spreader [19].

In state of the art servers, CPU sockets are equipped with fans for cooling. For example, the Intel s5400sf server has two sockets where each socket has two sets of fans [3] that blow air toward its heat sink. The heat flow between the CPU case to ambient can be modeled by a combination of conduction and convection

heat transfers, [57]. The heat sink is assumed to be an isothermal layer due to its high thermal conductance [57]. R_{hs} represents the thermal conductive resistance of the heat sink. The convective part is modeled by a convective resistance, R_{conv}, connected in series with R_{hs} where their sum represents the case to ambient resistance, R_{ca}. R_{ca} is connected in parallel with the thermal capacitance of the heat sink, C_{hs}, to form a simple RC circuit that has a time constant that is in the orders of magnitude larger than the time constant of the heat spreader [19]. The reference temperature, *local ambient temperature*, measured inside of the server enclosure is normally higher than the room's ambient temperature by 20^oC typically [58]. We call the temperature between the CPU case to the local ambient as T_{ca}.

In [2,47], it is shown that the value of R_{conv} changes with the air flow rate. Unfortunately, HotSpot uses a fixed R_{conv} since it assumes a fixed fan speed, which does not represent real systems. Using the results in [2,47], the value of R_{conv} can be computed as :

$$R_{conv} \propto \frac{1}{AV^{\alpha}} \tag{4.1}$$

where A is the heat sink effective area, V is the air flow rate and α is a factor with a range of (0.9 - 1.0) for heat sinks in high end servers. To estimate the cooling costs we use the results from [47] to relate the fan speed, F, with the air flow rate as: $V \propto F$. The cooling costs for changing the air flow rate from V_1 to V_2 can be computed as [47, 54]:

$$\frac{P_{V2}}{P_{V1}} = (\frac{V_2}{V_1})^3 \tag{4.2}$$

where P_{V1} and P_{V2} represent the fan's power dissipation at V_1 and V_2 respectively. Next we calculate the amount of fan power that is required to reduce T_{ca} from T_{ca_1} to T_{ca_2}. For a given CPU power, and using (4.1), we can write $\frac{V_2}{V_1} = (\frac{T_{ca_1}}{T_{ca_2}})^{\frac{1}{\alpha}}$. Using this result and (4.2) we get,

$$\frac{P_{V2}}{P_{V1}} = (\frac{T_{ca_1}}{T_{ca_2}})^{\frac{3}{\alpha}} \tag{4.3}$$

This shows that optimizing the fan speed is crucial for power savings since reducing T_{ca} requires an increase in the fan power on the order of $\frac{3}{\alpha}$.

Table 4.1: Temperature profile of a CPU with heat sink

Workload	T_{ja}^{max}	T_{sa}	T_{ca}	T_{ha}
{9W, 9W, 9W, 9W}	71.6 ^{o}C	54.0 - 54.8 ^{o}C	52.6 - 53.3 ^{o}C	50.1 - 50.8 ^{o}C
{13W, 9W, 7W, 7W}	79.3 ^{o}C	53.9 - 54.8 ^{o}C	52.5 - 53.2 ^{o}C	50.2 - 50.9 ^{o}C

Verification of thermal and cooling models: We start with showing that the value of T_{ca} is a function of the total power dissipated in the CPU socket rather than specific temperature distribution of the cores. We illustrate our ideas using HotSpot simulator which we extend to include the dependency of R_{conv} on the air flow rate. We assume a 4 core CPU socket with a floor plan and thermal package similar to a quad core Intel Xeon [2], an air flow of 20 CFM and local ambient temperature inside server enclosure of 42 ^{o}C. We run two cases as shown in Table 4.1. In the first case we simulate execute four threads where each thread dissipates 9W of total power. For the second case, we run four threads that accumulate similar total power to the first case but with large variation in the power distribution {13W, 9W, 7W, 7W}. The results show that T_{ca} of the two cases stay almost the same despite the large difference between their peak temperatures, T_{ja}^{max}. The results also show similar behavior for the heat spreader temperature, T_{sa}, and temperature between heat sink surface to local ambient, T_{ha}.

The other important feature that we explore is the correlation between T_{ja}^{max} and T_{ca}. Figure 4.4 shows how T_{ja}^{max} and T_{ca} changes with fan speed and executing various workloads that have different dynamic power distribution. Leakage power is calculated using a model that accounts for temperature effect on leakage that we give later in Figure 4.9. It can be noticed that T_{ja}^{max} changes at a rate similar to T_{ca}. This is because T_{ca} is connected in series with the junction to case temperature. The value of T_{ja}^{max} in the case of {12W, 9W, 9W, 9W} is higher than the one in {12W, 12W, 9W} despite the fact that the later has two hot threads at 12W while the former has only one. The reason is that the later workload has lower total power which is manifested by its lower T_{ca}. Increasing the fan speed reduces the

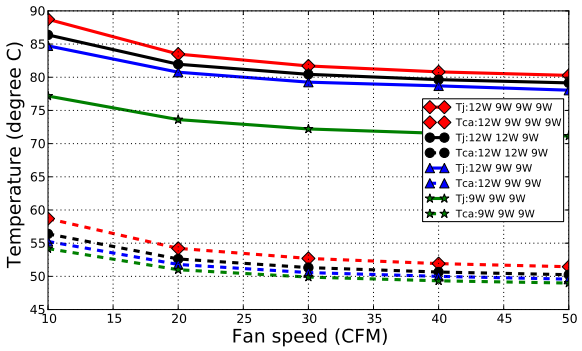


Figure 4.4: Impact of fan speed on core and case to ambient temperature

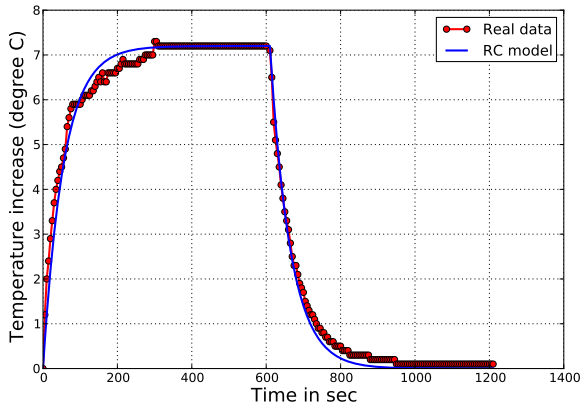


Figure 4.5: T_{ha} transient behavior (referenced to idle temperature)

value of T_{ca} due to the reduction in R_{conv}. These results indicate that T_{max} can be used to control T_{ja}^{max} and the cooling rate.

We extend the evaluation of the cooling model by running experiments on a real 45nm Intel Quad Core dual socket Xeon E5440 machine. The aim of the first experiment is to show that modeling the heat transfer of the heat sink as an RC circuit that is comprised of R_{ca} and C_{ca} is accurate enough. To perform this experiment we inserted external thermal sensor at the middle of the heat sink surface which measures, T_{ha} (temperature across the convective resistance, refer to Figure 4.3). We run workload in a way that can capture the transient behavior of the package by starting at the idle state then at time 0 we execute 2 threads of *perl*

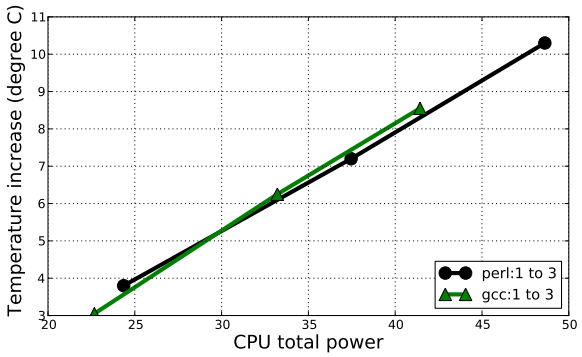


Figure 4.6: T_{ha} vs. CPU total power. T_{ha} is referenced to idle temperature

(refer to Table 4.2) for 600 seconds followed by another 600 seconds of idleness. To get representative measurements, the fans are kept at a default speed, which is about 25% of the max speed in our server. We set the local ambient to 24^{o}C to keep the machine cool enough so the fan speed stays fixed at the default value. Figure 4.5 show the measured and the modeled values of T_{ha} (referenced to its value when idle). The results clearly show a strong match between the real data and our RC circuit model. The transient behavior of T_{ca} is similar to T_{ha} except it has a higher amplitude due to the extra temperature across R_{hs} (refer to Figure 4.3).

In the next experiment we validate our assumptions and simulations that show the temperature of the heat sink is a function of the total power consumed in the CPU. We measure steady state temperature of T_{ha} (referenced to its value when idle) using the external thermal sensor as before. We execute 1 to 3 threads of two benchmarks, *perl* and *gcc* where *perl* have higher core power than *gcc* (refer to Table 4.2). We use the same fan speed setup as in the previous experiment. The results in Figure Figure 4.6 clearly show that T_{ha} changes linearly with the total power despite us running two different workloads and varying the number of threads.

4.2.2 Sources of cooling savings at the socket level

Energy of cooling subsystems could be reduced by intelligently distributing the workload across the CPU sockets. To illustrate this, we use dual 4 core Intel Xeon sockets where each is associated with a fan. Two types of threads are executed, one highly active that consumes 14W total power, and the other moderately active with 9.5W. Temperature threshold is 85^{o}C, and local ambient temperature inside server is set to 42^{o}C. We use HotSpot for thermal simulation. Figure 4.7 shows the impact of workload assignment on cooling cost savings at the socket level. The left part of the figure shows the thread assignments by state of the art schedulers while the right part shows their assignments by our scheduling model. As shown in this figure, when there is a high imbalance in the total power between the sockets, we can intelligently balance power across the sockets and save on the cooling costs. The savings are 60%. We call this class of assignment *spreading*.

In the second scenario the air flow rate of socket 1 in the original assignment is about twice of that of socket 2. To minimize the cooling costs we can swap the hot thread from socket 2 with two moderate threads from socket 1. Moving the hot thread to socket 1 does not increase the peak temperature since its power is similar to the power of hottest thread that is already there. In fact, the new assignment lowers the maximum temperature in socket 1 due to the reduction in total power by 5 Watts. The savings are 68%. We denote this class of assignments as *consolidation*.

However, a number of challenges need to be addressed to adopt a combination of spreading and consolidation. Workload migration across sockets comes with performance overhead of about 50-100us. We need to show that the time scale for cooling aware scheduling is orders of magnitude higher than the migration latency to ensure negligible performance overhead. The other challenge is to ensure a stable solution since the fan is a mechanical device and large variations in the fan speed impact its lifetime. Stable solution also helps to minimize the number of migration events between sockets. To solve this problem we use a control theoretic approach since ad-hoc solutions do not provide stability guarantees.

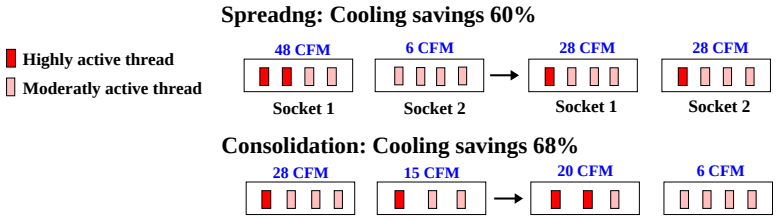


Figure 4.7: Cooling aware scheduling at the socket level

4.2.3 State-space controller and scheduler

The first stage of our socket level management is the controller. It determines the amount of power that needs to be added/removed from the sockets to reduce cooling energy by balancing the fan speed when there is a large difference in the total power dissipation between CPU sockets. Such imbalance in fan speed is the source of energy inefficiency due to the cubic relation between fan power and its speed. The extra cooling cost consumed by a given socket, C_{cost_i}, can be computed as:

$$C_{cost_i} \sim (F_i^3 - F_{avg}^3) \tag{4.4}$$

where F_i and F_{avg} are the fan speed of the given CPU socket and target speed respectively.

To design this controller we use state-space control since it is robust and scalable. We first extract the sate-space thermal model for the case to ambient temperature. The instantaneous value of case to ambient temperature for a given CPU, i, $T_{ca_i}(t)$, can be written as,

$$\frac{dT_{ca_i}(t)}{dt} = -\frac{T_{ca_i}(t)}{\tau_{ca_i}} + \frac{P_{cpu_i}(t)}{C_{ca_i}} \tag{4.5}$$

where, τ_{ca_i} is the heat sink time constant, $\tau_{ca_i} = R_{ca_i} * C_{ca_i}$. The $P_{cpu_i}(t)$ represents the instantaneous power dissipated in the CPU socket. For the case of n number of CPUs, the vector of case to ambient temperatures can be written as $T_{ca} = [T_{ca_1}(t), T_{ca_2}(t), \ldots, T_{ca_n}(t)]^T$ and the vector of CPUs power can be expressed as $P_{cpu} = [P_{cpu_1}(t), P_{cpu_2}(t), \ldots, P_{cpu_n}(t)]^T$. For the case of n CPUs, the equation (4.5) can be written as:

$$\frac{dT_{ca}(t)}{dt} = YT_{ca}(t) + ZP_{cpu}(t) \tag{4.6}$$

where Y and Z are diagonal matrices that can be written as:

$$Y = \begin{pmatrix} -\frac{1}{\tau_{ca_1}} & & 0 \\ & \ddots & \\ 0 & & -\frac{1}{\tau_{ca_n}} \end{pmatrix} \quad Z = \begin{pmatrix} \frac{1}{C_{ca_1}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{C_{ca_n}} \end{pmatrix}$$

The continuous system given in (4.6) can be discretized using the transformations given in [26] as:

$$T_{ca}(k+1) = \Phi T_{ca}(k) + \Gamma P_{cpu}(k) \tag{4.7}$$

where $\Phi = e^{Y\Delta t}$ and $\Gamma = Z\int_0^{\Delta t} e^{Yu}\,du$. The value of Δt is the sampling time. Since Y and Z are both diagonal matrices, Φ and Γ can be computed fairly easily as follows:

$$\Phi = \begin{pmatrix} e^{-\frac{\Delta t}{\tau_{ca_1}}} & & 0 \\ & \ddots & \\ 0 & & e^{-\frac{\Delta t}{\tau_{ca_n}}} \end{pmatrix}$$

$$\Gamma = \begin{pmatrix} \frac{1}{C_{ca_1}}(\int_0^{\Delta t} e^{-\frac{u}{\tau_{ca_1}}}\,du) & & 0 \\ & \ddots & \\ 0 & & \frac{1}{C_{ca_n}}(\int_0^{\Delta t} e^{-\frac{u}{\tau_{ca_n}}}\,du) \end{pmatrix}$$

where the integrals in Γ have a simple analytical solutions as, $\frac{1}{C_{ca_i}}(\int_0^{\Delta t} e^{-\frac{u}{\tau_{ca_i}}}\,du) = R_{ca_i}(1 - e^{-\frac{\Delta t}{\tau_{ca_i}}})$.

To design the controller we use standard control law that is based on the feedback of the system states [26]. Applying the control law to the state-space model given in (4.7), yields:

$$P_{cpu}(k) = -GT_{ca}(k) + G_oT_{ref}(k) \tag{4.8}$$

where G and G_o are diagonal matrices of state feedback gain and input gain respectively. T_{ref} corresponds to the vector of target temperatures. Using (4.7) and (4.8), the closed loop heat sink temperature can be written as:

$$T_{ca}(k+1) = (\Phi - \Gamma G)T_{ca}(k) + \Gamma G_o T_{ref}(k) \tag{4.9}$$

This system is stable when the eigenvalues of $(\Phi - \Gamma G)$ are within a unit circle. We can find G by setting the eigenvalues to the desired values where the criteria for their selection are discussed shortly in this Section. To compute the elements of G_o, we use steady-state analysis which requires the value of $G_{o_{ii}}$ to be equal to $(R_{ca_i}^{-1} + G_{ii})$, so the closed-loop system can settle at T_{ref_i}.

The T_{ref_i} can be computed based on the difference between the current fan speed and the target speed as described earlier. We need to translate the speed difference into change in case to ambient temperature, $\Delta T_{ca_i}(k)$, so as $T_{ref_i}(k) = T_{ca_i}(k) + \Delta T_{ca_i}(k)$. The value of T_{ca_i} can be estimated through thermal sensors attached to the heat sink. To estimate $\Delta T_{ca_i}(k)$, we calculate the change in the CPU case to ambient resistance, $\Delta R_{ca_i}(k)$ that corresponds to the difference between the current fan speed and the target speed. The $\Delta T_{ca_i}(k)$ can be estimated easily based on Ohm's as $\Delta T_{ca_i}(k) = \Delta R_{ca_i}(k)\frac{T_{ca_i}(k)}{R_{ca_i}(k)}$. The final value of T_{ref_i} can be calculated as:

$$T_{ref_i}(k) = T_{ca_i}(k) + \Delta R_{ca_i}(k)\frac{T_{ca_i}(k)}{R_{ca_i}(k)} \tag{4.10}$$

We compute the transient response time of the controller by converting (4.9) to z-domain as: $zT_{ca_i}(z) = \frac{z\gamma_i T_{ref_i}(z)}{z-(1-\gamma_i)}$, where $\gamma_i = 1 - \Phi_{ii} - \Gamma_{ii}G_{ii}$. The value of zT_{ca_i} is equivalent to $T_{ca_i}(k+1)$. Based on this equation we obtain the time constant of the state-space controller for each CPU as:

$$\tau_{ssc_i} = -\frac{-\Delta t}{ln(1-\gamma_i)} = -\frac{-\Delta t}{ln(\lambda_i)} \tag{4.11}$$

This equation shows that the transient time is a function of the sampling time and the controller eigenvalue, λ_i. Using this equation, we can set Δt to a few seconds while keeping τ_{ssc_i} in the range of seconds which is sufficient in our case since the temperature of the heat sink changes very slowly. It is not a good idea to make τ_{ssc_i} too short since it can cause undesirable overshoot in the system.

Algorithm 1 Socket level scheduling

Calculate ΔP_{cpu} and the set of P_{thr} for each CPU. Set Q as an empty queue
*** Spreading ***
for i in set of unmarked CPUs **do**
  for j in unmarked threads in CPU i **do**
    dest $\Leftarrow$ CPU index with $max(\Delta P_{cpu})$
    if ($P_{thr_j} < -\Delta P_{cpu_i}$ **and** $P_{thr_j} < \Delta P_{cpu_{dest}}$) **then**
      if CPU *dest* has idle core **and** no core level migrations in CPU *dest* **then**
        Calculate cooling savings of migrating thread j to CPU *dest*
      else
        Calculate cooling savings of swapping thread j with the coolest unmarked thread from CPU *dest*
      end if
      if cooling savings $> S_{min}$ **then**
        Enqueue this migration event Q and mark migrated threads
        Update ΔP_{cpu} and threads assignment
        Mark CPU i and *dest*
      end if
    end if
  end for
end for

*** Consolidation ***
while (unmarked CPUs > 1) **do**
  $H \Leftarrow$ index of unmarked CPU with max fan speed
  if (fans have different speed) **then**
    $L \Leftarrow$ index of unmarked CPU with min fan speed
  else
    $L \Leftarrow$ index of any unmarked CPU that is different from H
  end if
  Find the hottest thread in CPU L (h^L)
  Starting from the coolest thread in H, find the smallest set of threads (S_{cool}) with total power $\geq P_{thr_{h^L}}$ (P_{thr} of each thread in H is $< P_{thr_{h^L}}$), then mark CPU H
  if (S_{cool} is not empty) **then**
    Calculate cooling savings of swapping the threads in S_{cool} with h^L
    if cooling savings $> S_{min}$ **then**
      Enqueue this migration event in Q, update threads assignment and then mark CPU L
    end if
  end if
end while
Execute all migration events in Q

4.2.4 CPU Socket Scheduling:

The details of the socket scheduling are given in *Algorithm* 2. The ΔP_{cpu} in this algorithm refers to the vector of the differences between the requested power by the controller and current power of the CPUs while P_{thr_i} corresponds to the power of the i^{th} thread in a given CPU. Individual scheduling events are allowed only when the predicted cooling savings are higher than a given threshold, S_{min}. The concepts of this algorithm are discussed below.

Spreading step (steps 3-19): Intuitively, since the fan power increases exponentially with cooling capacity it is important to balance the heat generation between the sockets in order to lower their fan speed and obtain cooling savings. Spreading is effective as long as there is sufficient imbalance in power consumption between CPU sockets such that the additional spreading migrations can lead to a better balance in socket power and fan speed. When this difference become sufficiently small, then the fans of the respective CPUs have reached their minimum speed from the spreading point of view. The controller decides on the amount of power that needs to be added to/removed from the sockets at each scheduling tick. We spread the workload starting with cooler threads to have finer grain control of the total power on each socket. Before each migration we evaluate the cooling savings as described in Section V.E to prevent ineffective scheduling decisions.

Consolidation step (steps 21-37): On the other hand, consolidation focuses on the cases when the total power consumption between sockets is comparable. Intuitively, since there are only relatively few fan speed settings, it is possible that at a particular speed the fan is actually capable of cooling a bit higher on die power density than is currently present on the socket. In a situation where there is a slight power density imbalance between the two sockets, it is possible that by switching cool and hot threads the increase in power density on one socket is not too high, thus keeping the fan speed constant, but the decrease on the other socket is just big enough enabling it to lower its fan speed, and as a result saving energy. Thus our strategy is to identify a set of cool threads on one socket whose power density is similar to a hot thread running on another socket, and then to swap them if our estimates (see Section 4.2.5) show that we will be able to save cooling

energy.

4.2.5 Estimating cooling savings

We can estimate cooling savings for a particular scheduling decision by predicting the resultant fans speeds. Let's assume we need to migrate a thread that consumes power, P_{thr}, from CPU i to CPU j. To approximate the resultant fan speed of the source CPU we apply Ohms's law to estimate the new R_{ca_i} which can be translated into fan speed. Let's assume that the increase in R_{ca_i} due to the migration equals ΔR_{ca_i}. The value of ΔR_{ca_i} can be estimated as follows assuming the migrated thread is not the hottest in CPU i:

$$\Delta R_{ca_i} = \frac{P_{thr} R_{sa_i}}{P_{cpu_i} - P_{thr}} \tag{4.12}$$

where $R_{sa_i} = R_{ca_i} + R_{s_i}$. If the migrated thread is the hottest in CPU i, then $\Delta R_{ca_i} = \frac{P_{thr} R_{sa_i} + R_{core_i} \delta p_{thr_i}}{P_{cpu_i} - P_{thr}}$. where δp_{thr_i} represents the core power difference between the hottest thread that we migrate and the second hottest thread in the CPU i. For the socket that is receiving the extra thread, there are few cases that need to be considered.

Case A: The first case is when the migrated thread dissipates less core power than the hottest thread in the destination socket and the destination's fan is not idle. Using Ohm's law, the value of its ΔR_{ca_j} can be computed simply as:

$$\Delta R_{ca_j} = -\frac{P_{thr} R_{sa_j}}{P_{cpu_j} + P_{thr}} \tag{4.13}$$

Case B: This case addresses the scenarios when the newly migrated thread dissipates more core power than the hottest thread in the destination socket and the destination's fan is not idle. When adding a thread that consumes more power than the current hottest thread, we expect maximum temperature increases not only in the heat sink and heat spread but also in the core. The value of ΔR_{ca_j} is as follows,

$$\Delta R_{ca_j} = -\frac{P_{thr} R_{sa_j} + R_{core}(P^{core}_{thr} - P^{max}_{thr_j})}{P_{cpu_j} + P_{thr}} \tag{4.14}$$

where $P_{thr_j}^{max}$ corresponds to the core power of the hottest thread in the destination CPU j. P_{thr}^{core} refers to the core power component of the migrated thread.

Case C: The next case we need to consider is when the destination fan is idle. If the thread to be migrated has lower core power than the hottest thread T_{max_j}, then adding the extra thread will increase the temperature of already existing hottest thread by $P_{thr}R_{sa_j}$. If the predicted core temperature after migration, $T_{max_j}^{new} = T_{max_j} + P_{thr}R_{sa_j}$, is found to exceed the fan trigger temperature, T_c^{fan}, then we can estimate ΔR_{ca_j} as follows:

$$\Delta R_{ca_j} = -\frac{T_{max_j}^{new} - T_c^{fan}}{P_{migr} + P_{cpu_j}} \tag{4.15}$$

Case D: The last case is when the power of the thread to be migrated exceeds the power of the hottest thread in the destination socket that has its fans idle. In this case the maximum temperature in the destination CPU would be higher due to the increase in T_{ca_j} as well as the core and heat spreader temperatures. We can calculate the resultant ΔR_{ca_j} based on Ohm's law as before. We next present how we estimate the induced power by the active threads at low cost.

4.2.6 Power estimation

The CPU power can be divided into two basic components: dynamic power and leakage power [48]. The dynamic power is a function of the induced activity by the application on the components. On the other hand, the leakage power is function of components area and the current temperature. In the following we describe how we estimate these values at small overhead. We need power traces for each core in our system in order to evaluate temperature and cooling savings.

To model the dynamic power for the CPU cores we need a metric that is directly correlated with the level of activity in the core. For this, we use the number of retired instructions per cycle or IPC which is shown to have a good correlation with the CPU power [24]. Executing instructions cover almost all of the dynamic activities in the core. Additionally, the rate of the instruction execution is expected to have a sufficient correlation with the induced activity since the faster the execution the more activity there is.

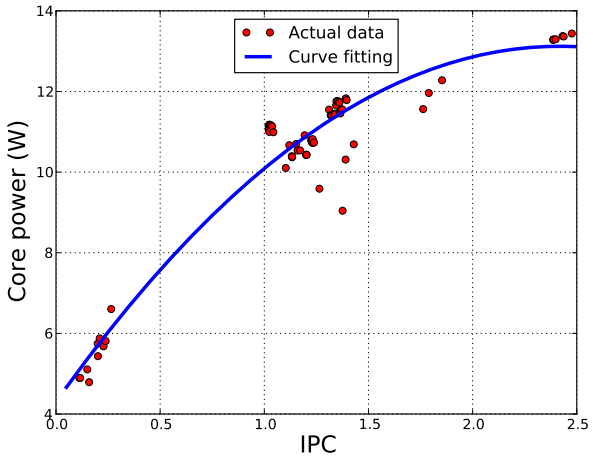


Figure 4.8: Dynamic power model

Before we generate the model we collect real time power and IPC measurements data on state of the art quad core dual socket Xeon E5440 server. The IPC traces are collected using processor performance counters (Model Specific Registers, or MSR). We generate the power traces as follows. We run the benchmarks in one of the cores and keep the rest in idle mode to get an accurate power trace per benchmark. We measure the power of the CPU package by inserting current sensors in the power connector of the CPU socket and collect power traces using data acquisition system. This power is composed of three main components: CPU baseline power, core power and L2 cache power. To exclude the CPU baseline power we measure the CPU power when it is idle and then subtract this baseline power from the CPU socket power trace. We deactivate the CPU C-states during the baseline power measurements as otherwise the CPU socket would go to a deep C-state when it is not executing. The power of the idle cores is already accounted for in the baseline power measurement. The resultant trace represents the core and L2 cache power components, which we call *core+L2* trace. Subsequently, we estimate the L2 power based on its access rate since it is a regular structure. We convert the L2 access rate to power using Cacti tool that is widely used for estimating the power of the cache subsystem [60, 61]. The trace of L2 access rate is extracted during execution using CPU performance counters. We isolate the core

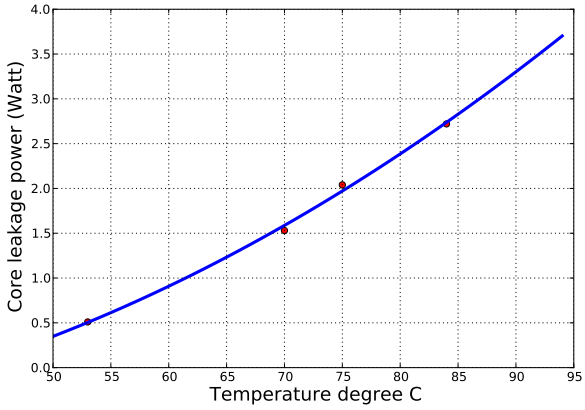


Figure 4.9: Leakage power model

power trace by subtracting the L2 power from core+L2 trace.

To generate the dynamic power model we ran five SPEC2000 benchmarks that span from memory intensive to cpu intensive applications. For each benchmark we collected 10 power and IPC samples that are sufficient since the benchmarks behavior is reasonably stable. To generate the model we use regression to find the best fit for the samples using *polyfit* function in python programming language. Figure 4.8 shows the dynamic power model for the cores where polynomial of order 2 is used. The results show that using IPC metric exhibit a good correlation with the dynamic power. The average error is less than 1 Watt which is sufficient for our thermal modeling needs.

The leakage model is a function of the components area and the current temperature. To generate the leakage model for Xeon E5440 we run experiments by blocking the air flow to increase the CPU temperature and collecting the power measurements concurrently. Figure 4.9 shows how the core leakage power changes with temperature and the model that we generated based on these measurements. We developed the leakage model using the regression method where the model can be represented by a simple 2nd order polynomial. The results show a good agreement between the modeling and the experimental outcomes.

4.3 Evaluation

4.3.1 Methodology

We evaluate our approach using 45nm Intel Quad Core dual socket Xeon E5440 server. CPU sockets share 32GB main memory. The system runs the latest Linux kernel (2.6.33.2) that manages all system resources. We extract the core power dissipation traces of different benchmarks using measurements on our machine. We run the benchmarks in one of the cores and keep the rest in idle mode to get an accurate core and L2 power traces per benchmark using the method that is described in Section 4.2.6. The core and L2 power traces are used to estimate the core and L2 temperatures respectively using HotSpot simulator [57] and the Xeon processor layout from [4]. We also extract the baseline power of the CPU using the method in Section 4.2.6. We include the baseline power of the CPU in the temperature simulations since its impacts the heat spreader and heat sink temperature.

We use simulation instead of running our algorithms in real system due to a number of issues. The built-in fan control algorithm runs all the CPU fans at a single speed that is related to cooling needs of the hottest socket, which lead to over provisioning. The algorithm is implemented in a separate controller that we don't have access to. Consequently, not all the benefits of our algorithms can be manifested with using the built-in fan control algorithm. HotSpot is also necessary to conduct experiments that require setting different fan speeds per socket because the hardware does not currently support such fine grained control. HotSpot enables us to profile and study the temperature breakdown in the CPU and thermal package as a function of fan speed

Quad Core Xeon E5440 package characteristics that have been used in thermal simulation are listed in Table 5.1. For the baseline *fan control algorithm* we assumed a closed loop controller similar to that used in modern systems. The fan algorithm adjusts the fan speed in proportion to the heat level of their respective CPUs. When the temperature is below a given threshold the fan is set to idle speed. We set the fan to trigger 3 degrees below the CPU threshold temperature

Table 4.2: SPEC Benchmarks characteristics

Benchmark	Dynamic core power (W)
perl	12.7
bzip2	11.6
eon	11.2
twolf	11.1
gzip	10.4
gcc	10.2
mcf	5.47

Table 4.3: Characteristics of CPU, thermal and cooling

CPU	XeonE5440
TDP	80W
CPU frequency	2.8GHz
Heat spreader thickness	1.5mm
Case to ambient thermal resistance, R_{ca} K/W	$R_{ca} = 0.141 + \frac{1.23}{V^{0.923}}$, V: air flow in CFM, [2]
CPU thermal threshold	85 ^{o}C
Max air flow rate per socket	53.4 CFM [5]
Fan power per socket	29.4 W [5]
Fan steps	32
Idle fan speed	10% of max speed

to allow for enough time to react to thermal emergencies. In cases when the temperature exceeds the CPU critical temperature we use throttling mechanism as back up. We assume there is a support for accurate temperature sensors reading in the system. Recent work [53, 64] has proposed efficient techniques to estimate accurate temperature in presence of noisy sensors. The local ambient temperature inside the server box is set to 42 ^{o}C

Benchmarks from the SPEC2000 suite have been used as workloads (see Table 4.2). A set of benchmarks are selected that exhibit various levels of CPU intensity to emulate real life applications. We run each benchmark in the work

set till its completion and repeat it until the end of simulation time. We also use synthetic benchmarks to test the transient behavior of our socket level scheduler.

We evaluate our core level and multi-tier algorithms. Our Core level Proactive Thermal Management, CPTM, reduces the cooling costs by minimizing the hot spots between cores, please refer to chapter 3. Multi-Tier Thermal Management, MTTM, performs core level and socket level thermal management to reduce the hot spots within and between sockets. We compare these algorithms against the following set of state of the art policies:

Dynamic Load Balancing, DLB: It is usually implemented in modern operating systems to enhance the utilization of the system resources. The DLB performs thread migration to minimize the difference in task queue lengths of the individual cores [21]. The operating system initiates dynamic load balancing every hundred milliseconds. The balancing threshold is set to one to keep the difference between queues equal to zero as possible. In this work we implement the DLB as our *default policy* for the purposes of comparison.

Core level Reactive Thermal Management, CRTM: This policy is similar to our CPTM except that the thermal scheduling decisions are reactive, they are based on the current temperature rather than predicted one.

Basic Socket level Thermal Management, BSTM: When there is imbalance in the fan speed, the policy migrates the hottest thread from the socket that is associated with the highest fan speed to the socket with the lowest fan speed. We set the scheduling period to be similar to that of our algorithm, MTTM, to eliminate fan instability that can result from poor selection of the scheduling period.

4.3.2 Results

To quantify and understand clearly the benefits of our algorithms in reducing the cooling energy, first we evaluate our core level algorithm. Then we study our multi-level technique. At the end we discuss the overhead associated with our algorithms. The results show that our technique gives average cooling energy savings of 80%.

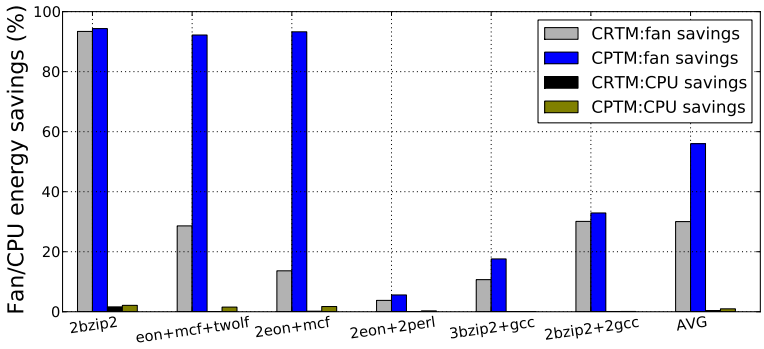


Figure 4.10: Cooling and CPU energy savings using core level polices

To evaluate the benefit of our core level approach for cooling energy savings, we run various combination of workloads in a single socket. We set the local ambient temperature to 42^oC. The results in Figure 4.10 show the energy savings of CRTM and CPTM compared to the default policy, DLB. The results show that using CPTM provides significant cooling savings over the other techniques; the average improvement over the DLB policy is 56%. As it can be seen from the results, the cooling savings are higher at lower utilization or with more heterogeneity in the workload. Execution at lower utilization gives more room for distributing the heat between cores, thus reducing thermal emergencies. For the case of 50% utilization, $2bzip2$, the cooling savings reach 95%. The other case of $2eon + mcf$ is an example of heterogeneous workload that is composed of cpu intensive, *eon*, and memory intensive *mcf* jobs and has a higher utilization factor of 75%. The reported results show that CPTM is able to deliver 93.3% cooling savings. It is able to provides nice cooling savings with heterogeneous workload even when the processor is executing four threads as shown in the case of $2bzip2 + 2gcc$. The limitation of CPTM is shown in the case when the socket is running four threads that are all CPU intensive, $2perl + 2eon$. The cooling savings in this case are small since there is a little room for better thermal balancing as all cores are heavily used. We will show shortly that such cases can be mitigated by using our socket level scheduling approach because it expands the scope of thermal management and provides more opportunities for better thermal distribution. On the other hand, the CPU socket energy savings due to lowering the leakage power are low;

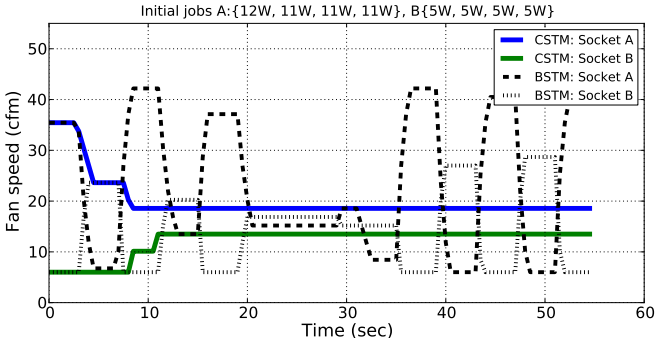


Figure 4.11: Workload spreading

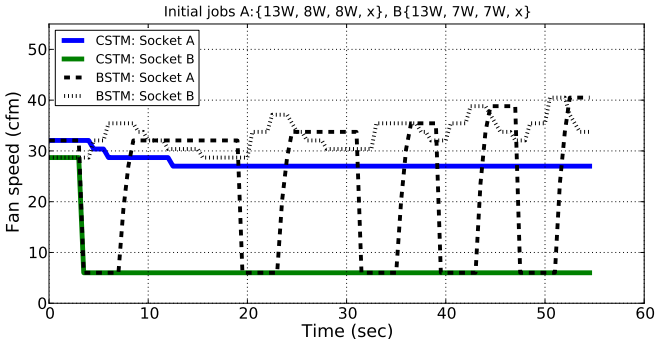


Figure 4.12: Workload consolidation

the average improvement using CPTM over the DLB policy is about 1.0% which is slightly higher than the case of CRTM. The reason for the low savings is related to the fan contribution of keeping the temperature around the target threshold for both DLB and core level policies, which minimizes the difference in leakage power between these polices.

Next we address the stability of our socket scheduling. In this study we compare our control-theoretic socket level thermal management, CSTM, against the basic socket level thermal management, BSTM. We study two situations, one where spreading strategy gives a good solution and the other where consolidation is more applicable. Figure 4.11 shows the spreading case with the following initial job assignment, socket A:$\{12W + 11W + 11W + 11W\}$ and socket B:$\{5W + 5W + 5W + 5W\}$ and local ambient temperature of 41^{o}C. The efficient solution for this case is to have the two sockets run a better balanced workload as A:$\{12W + 5W + 11W + 5W\}$ and B:$\{11W + 5W + 11W + 5W\}$. The CSTM controller converges to the target solution in two scheduling ticks only. In contrast, BSTM policy fails

Table 4.4: Workload combinations for multi-tier algorithm

Workload	Socket A	Socket B	Local ambient ^oC
W1	*3eon*	*eon + mcf + gcc*	42
W2	*2eon + mcf*	*eon + bzip2 + mcf*	42
W3	*2bzip2 + 2mcf*	*2bzip2 + 2mcf*	42
W4	*2perl + 2eon*	*2gcc + 2mcf*	42
W5	*2perl + 2bzip2*	*2gcc + 2mcf*	39
W6	*2perl + 2bzip2*	*2gcc + 2mcf*	36
W7	*2perl + bzip2*	*gcc+2mcf*	42
W8	*perl + 3gcc*	*perl + 2gcc*	42
W9	*perl + 3gcc*	*perl + 2gcc*	40
W10	*perl + 3gcc*	*perl + 2gcc*	38
W11	*perl + 3gzip*	*perl + gcc + gzip*	42
W12	*3eon*	*3eon*	42
W13	*2eon+mcf*	*2eon+mcf*	42
W14	*2bzip2*	*2bzip2*	42

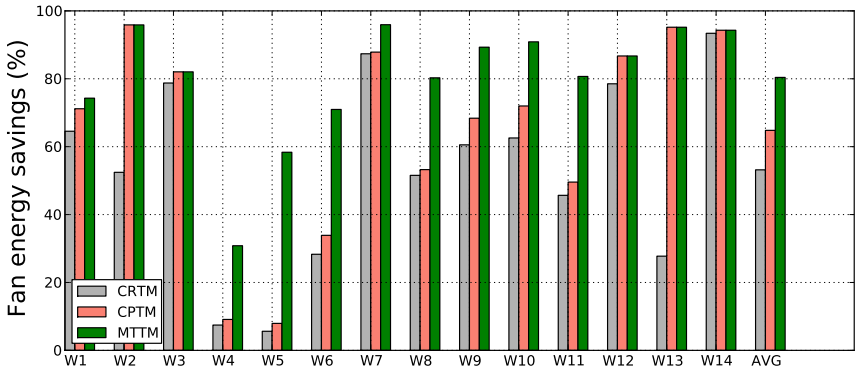


Figure 4.13: Cooling energy savings using multi-tier thermal management

to converge. The transient behavior of a consolidation case is shown in Figure 4.12. The initial workload assignment for sockets A and B is: $\{13W + 8W + 8W\}$ and $\{13W + 7W + 7W\}$ respectively. Our controller swaps the $13W$ thread from socket B with two threads of $8W$ from socket A and converges to the solution in one scheduling tick. In this case, the fan of socket A slightly reduces while the fan of socket B drops to idle speed. On the other hand, the BSTM fails to reach a stable solution. The results indicate that our controller is able to converge to the target solution while BSTM heuristic solution fails in this aspect.

To evaluate our multi-tier thermal management, MTTM, we use a combination of workloads with various thermal stress levels and utilization values. The list

of the workload combinations that we use is given in Table 4.4. Figure 4.13 shows the cooling energy savings of core level reactive thermal management (CRTM), core level proactive thermal management (CPTM) and MTTM compared to the default dynamic load balancing (DLB) policy. The results show that using MTTM provides substantial savings over the other techniques. The average improvement over the default policy, CRTM and CPTM are 80.4%, 33.8% and 19.4% respectively. The DLB performed the worse as expected since it does not optimize the thermal distribution within the socket. CPTM performed better than DLB and CRTM since it provide better thermal distribution in the chip which leads to a lower cooling. The extra savings over CPTM come from the socket level scheduling. The savings in the example of workload W5 are due to workload spreading. In this case, socket A has high thermal stress while socket B remains under low to medium stress. Under such scenario core level management becomes ineffective since all threads in socket A have high power density, so there is not enough room to spread the heat. However, workload spreading at the socket level is highly effective because it can balance the temperature between the two sockets. The savings in this case are 58.3%. Workloads W8 and W11 get savings primarily from consolidation. More specifically, in case of W8 the scheduler swaps *perl* with two instances of *gcc*. CPTM plays a big role when socket level scheduling becomes ineffective. This is apparent in the case of workload W13 where the savings come from CPTM since the workload between the two sockets is initially balanced.

We also study the impact of changing local ambient temperature inside the server on relative savings over the default policy. In the first case we reduce the local ambient temperature from 39 ^{o}C (W5) to 36 ^{o}C (W6) where the actual workload stays the same. The results show that MTTM savings over the default policy is improved at 36 ^{o}C due to the extra savings from the CPTM policy. Similar behavior can be seen in the cases {W8, W9 and W10} where the ambient changes from 42 ^{o}C to 38 ^{o}C. Our multi-tier thermal management effectively mitigates thermal problems when executing workloads with highly diverse thermal characteristics.

Figure 4.14 shows the maximum performance loss per core due to thermal emergencies when the fan cooling is insufficient or when the fan response is slow.

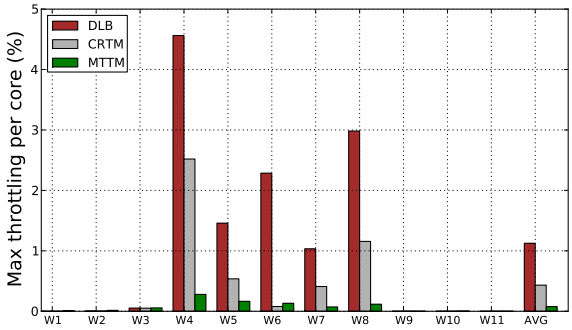


Figure 4.14: Reducing thermal emergencies

The results show that employing MTTM keeps the percentage of thermal emergencies below 1%, while for the other policies DLB, CRTM, and CPTM can reach 4.5%, 2.5% and 2.5% respectively. This advantage of MTTM come from managing the temperature at multiple-levels.

Overhead

The performance overhead of our multi-tier thermal management can broken down into the overhead of the core and the socket levels. The migration overhead between cores is in the range of few microseconds, e.g. 5us [40], which is much smaller compared to the time between OS scheduling ticks. The primary cause of this overhead is the warm up of the private cache of the cores. The computational overhead is negligible because new temperature prediction takes only few CPU cycles. The socket level migration overhead includes both thread migration and scheduler computational costs. When a thread migrates from a socket to another it needs to warm up the cache subsystem of the destination CPU socket. The memory subsystem is shared among the CPU sockets in our server which implements a single coherent memory system, where memory coherency is enforced by the hardware. Warming up the cache subsystem is the primary source of the migration overhead. This warm up time includes updating both L2 cache of the CPU and the private cache of the destination core as well. The overall warm up time increases by 10-20X compared to core level migration warm up. This is because the L2 needs to bring the data of the newly migrated thread from the memory this costs around 100-200 cycles (e.g. 150 cycles [40]). The overall migration time

is on the order of 50-100us which is included in our results. Despite this extra overhead, it is still orders of magnitude smaller than the period between the two scheduling ticks, which is on the order of seconds. The socket scheduler executes ordinary computations which require no more than several milliseconds. This analysis shows that our multi-tier thermal management is a light weight technique that makes it applicable to real systems.

4.4 Conclusion

With the ever increasing computational demand, the high end servers undergo tremendous thermal stress that has detrimental effect on the system characteristics. Many of the proposed dynamic thermal management techniques do not model system cooling dynamics and are limited to single socket CPU systems. Consequently, these techniques compromise system performance and provide substandard system power optimization.

In this work we propose a new multi-tier workload scheduling technique that minimizes the cooling energy costs of fan subsystem in a multi-socket CPU platform. At the core level, our band limited predictor is employed for proactive thermal management that minimizes the cooling costs and reduces the frequency of thermal emergencies. The top level scheduler leverages freedom in workload assignment at socket level and helps achieve better thermal distribution via spreading and consolidation. It employs our provably stable control theoretic framework for scheduling the workload with negligible performance overhead. The reported results show that our approach delivers 80% average cooling energy savings compared to the state of the art policies.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the International Symposium on Low Power Electronics and Design, 2009. Ayoub, R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in IEEE Transac-

tions in Computer Aided Design of Integrated Circuits and Systems, 2011. Ayoub, R.; Indukuri, K. R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Integrated Energy, Temperature and Cooling Management for CPU and Memory Subsystems in Servers

Traditionally the CPU is known to be a primary source of system power consumption. Over the years the designers have developed techniques to improve the energy efficiency of the CPU subsystem which accounts for approximately 50% of the total energy budget. Less attention is given to energy optimization in rest of the system components which leads to poor energy proportionality of the entire system [15]. Memory subsystem is the other major power hungry component in the server systems as it consumes up to 35% of total system energy and has poor energy proportionality [14, 15]. Capacity and bandwidth of the memory subsystem are typically designed to handle the worst case scenarios. Applications vary significantly in terms of their access rates to memory. Only a fraction of the pages are active while the rest are dormant. One solution is to activate a subset of the memory modules that can serve the applications needs to save energy [28]. Such clustering increases the power density of the active memory modules which can cause thermal problems in the memory. As CPU and memory consume the ma-

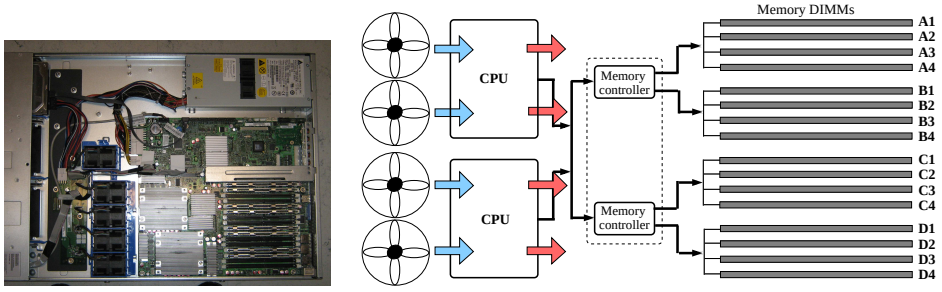
jority of the overall server power, they represent the key components for thermal and cooling management.

Modern servers incorporate a fan subsystem to reduce the temperature within servers. Due to cost and area constraints, a common set of fans is normally used to cool both the CPUs and memory [3]. For such scenarios, the inlet temperature of the downstream components that are at the end of air flow path becomes a function of the upstream component power density in addition to the primary inlet temperature of a server. In general, energy management techniques must consider temperature constraints and cooling energy costs for making intelligent decisions.

In this chapter we present CETC, a combined energy, thermal and cooling management algorithm for servers. Providing an integrated solution is necessary due to the thermal dependencies between the CPU and memory when both share same cooling resources. CETC maximizes the energy efficiency of the machine by controlling the number of active memory modules to just what is needed to provide sufficient storage capacity and bandwidth while minimizing the cooling costs. CETC also schedules the workload between the CPU sockets to create a more balanced thermal distribution between them not only to minimize the thermal coupling effect but also to mitigate their thermal hot spots as well. We developed a control theoretic framework that controls memory page assignment, socket level scheduling and fan speed to guarantee convergence to the desired objectives. Finally, we show that using CETC results in 70% average energy reduction of memory and cooling subsystem.

5.1 Combined thermal and cooling model for CPU and memory subsystems

In modern systems both CPU and memory components require cooling. Due to cost and area constraints, both components are normally cooled using a shared fan subsystem. Figure 5.1(a) depicts the photo of our 45nm Intel Quad Core dual socket Xeon E5440 server where the CPU is placed close to the fan



(a) Server machine (b) System organization and cooling

Figure 5.1: Intel dual socket Xeon server

while the memory is downstream [3]. Having a shared cooling resource creates thermal dependencies between the upstream and downstream components, so thermal modeling becomes more challenging. The memory subsystem in our server is shared among the CPU sockets where memory coherency is enforced by hardware. This server supports two off-chip memory controllers where each is connected to memory by two memory channels, each channel is connected to a 4 DIMMs slots as shown in Figure 5.1(b), each 4GB DDR2. We estimate the power of individual DIMMs by adding extenders that come with current sensors in the supply lines of each DIMM and aggregated the data using data acquisition system. The specs of the CPU, memory and cooling subsystems are given in Table 5.1. Benchmarks from the SPEC2000 suite have been used as workloads.

We next describe and verify our models. In this section we focus on developing an integrated thermal and cooling model for both CPU and memory which accounts for thermal dependency between them. We then investigate the opportunities of energy savings in memory and the associated thermal challenges.

5.1.1 System thermal model

Cooling systems use fans to generate air flow to reduce the temperature of hot components. In our dual socket system, the fan generates air flow that passes through the heat sink of the CPU and eventually reaches the memory DIMMs. This means that the memory and CPU are thermally connected as the CPU heats up the common air flow before it reaches the memory subsystem. The inlet temperature

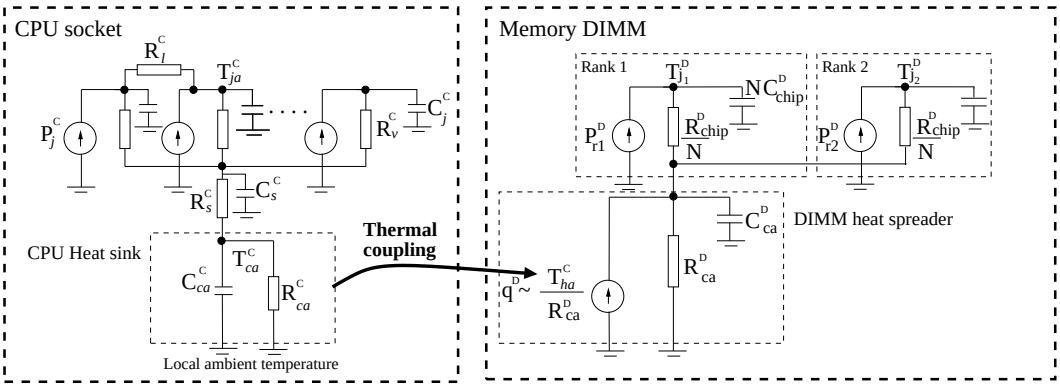


Figure 5.2: Combined thermal model

of the memory is a function of primary inlet air temperature and the heat sink temperatures of the CPUs. In a multisocket system, each CPU socket can heat air going to the memory subsystem to a different temperature depending upon power consumption of the socket. As a result, the temperature of the memory is a strong function of the hottest CPU.

Figure 5.2 show the unified thermal/cooling model of both CPU and memory. We combine CPU and memory thermal model using dependent thermal coupling heat sources to model the extra heat that is generated by the hottest upstream CPU. Definitions of CPU and memory thermal models are discussed in sections 4.2.1 & 5.1.2 respectively. The dependent coupling heat source of the memory, q^D, is proportional to the heat sink temperature of the CPU, T^C_{ha}, and inversely proportional to the case to ambient thermal resistance of the memory DIMMs, R^D_{ca}, as follows:

$$q^D \propto \frac{T^C_{ha}}{R^D_{ca}} \tag{5.1}$$

When the temperature of the CPU's heat sink rises, we expect the temperature of the memory to increase gradually due to the thermal capacitance effect of its heat spreader. Increasing the fan speed reduces the CPU temperature directly by lowering it's convective resistance. Unlike the case of CPU cooling, increasing the fan speed reduces the memory temperature in two ways. First, it reduces the temperature of the DIMM by reducing it's convective thermal resistance. It also

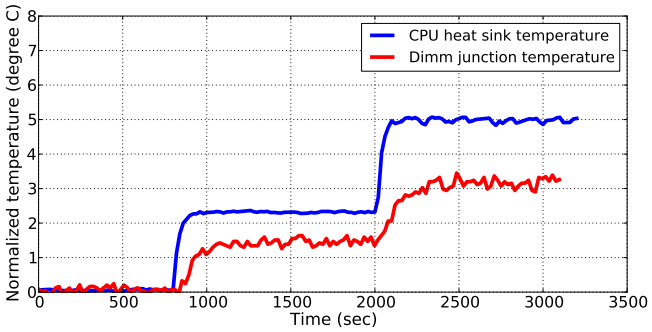


Figure 5.3: Thermal coupling between CPU and memory

lowers the CPU temperature which reduces the effect of the heat coupling, which further reduces the memory temperature.

Experimental verification of thermal coupling: We measure the temperature of the CPU heat sink which is located on the upstream of air flow using a thermal sensor. At the same time we measure the maximum junction temperature of the memory using Intelligent Platform Management Interface, IPMI, to observe the correlation between heat sink temperature and that of memory. We also measure the power of the DIMMs to account for temperature changes due to power variations. For this experiments we use 4 DIMMs, each connected to a single channel, where every two DIMMs are located after one CPU socket. We run a single threaded memory intensive workload from SPEC2006 suite, *swim*, on a single socket followed by progressively adding a CPU intensive workload, *perl*, to the same socket. The rationale behind running *swim* is to keep the memory power at the same level during the experiment. We run swim alone for 1100 seconds and execute a cpu intensive workload right after this period and we add additional CPU intensive workload at time 20000 seconds. During this experiment, we restart any finished jobs. Figure 5.3 shows the temperatures of memory and the CPU heat sink that are referenced to their values at the end of the warm up period that last for 300 seconds. We also plot the power consumption per DIMM. The results clearly show that a rise in the heat sink temperature causes a rise in the memory temperature due to the extra heat that is transfered to memory. We have observed

that the DIMMs that are located after the active CPU experience extra heat which creates a thermal imbalance among them.

5.1.2 Memory thermal and cooling model

Typical DRAM subsystem is organized as an array of multiple DIMMs (see Figure 5.1(b)), where each DIMM is composed of ranks, usually 2, and each rank contains multiple memory chips (e.g. 8 chips per rank). Each DRAM chip in a rank stores data that belongs to a predetermined segment of bits in the data word (e.g. the first 8 bits of a 64 bit data word goes to the first chip). As a result, the rank power is distributed almost equally across the DRAM chips inside the rank. To enhance the heat transfer between the DRAM chips and local ambient (ambient temperature is measured inside the server enclosure) the DIMMs have a heat spreader.

Figure 5.2 shows the thermal model of a DIMM that uses a heat spreader. We use superposition theory to simplify the RC network of the DRAM chips of each rank to a single RC node. In this Figure, the heat source P^D_{chip} equals to the power dissipated in each DRAM chip, R^D_{chip} is the vertical thermal resistance of each chip, C^D_{chip} is the thermal capacitance of each chip and T^D_j is the junction temperature of a DRAM chip. The number of DRAM chips in each rank is assumed to be N.

The heat transfer between the heat spreader and local ambient is modeled as an RC node as shown in Figure 5.2. This is because the heat spreader can be assumed to be a single node due its small horizontal thermal resistance relative to the convective resistance [6, 36, 37]. The thermal capacitance between heat spreader and ambient is represented by C^D_{ca}. The component R^D_s represents thermal resistance of the heat spreader. The heat transfer between the heat spreader and the interior ambient of the machine is modeled by a convective resistor, R^D_{conv}.

The junction transient temperature is a function of the temperature across the DRAM chips and the temperature from the heat spreader to ambient. However, the *time constant* (the time required for the temperature to rise from reference ambient temperature to 63.2% of the maximum value) of heat spreader is in the range of tens of seconds [37] while the time constant for the DRAM chip die is in

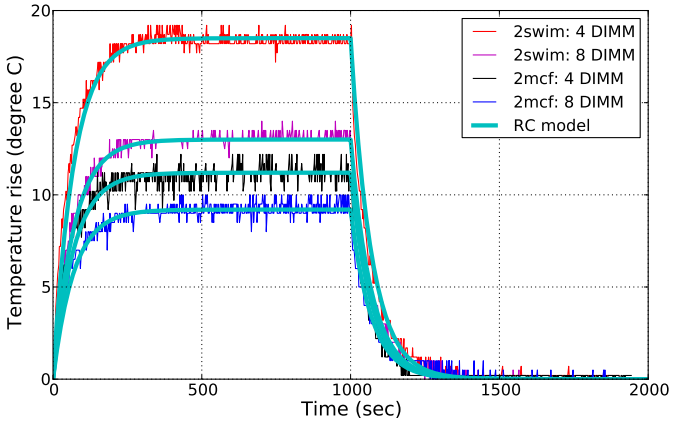


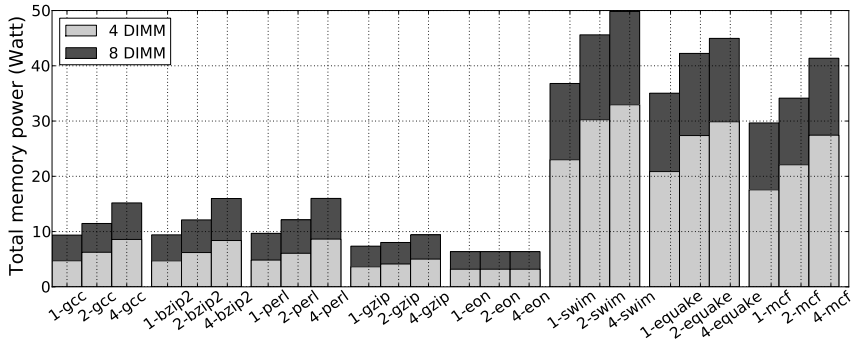
Figure 5.4: DIMM's transient temperature

the order of tens of milliseconds [57]. This means that the transient behavior of the junction temperature is dominated by the heat spreader temperature dynamics over the long run as the DRAM chip temperature reaches steady-state quickly. We exploit the big differences in time constants to reduce the thermal model complexity. We can simplify the model further by assuming that the power across ranks is similar since the memory controller uses interleaving to uniformly distribute the activities across the DIMMs as well as their ranks, n_r. The junction temperature of a given rank can be modeled as:

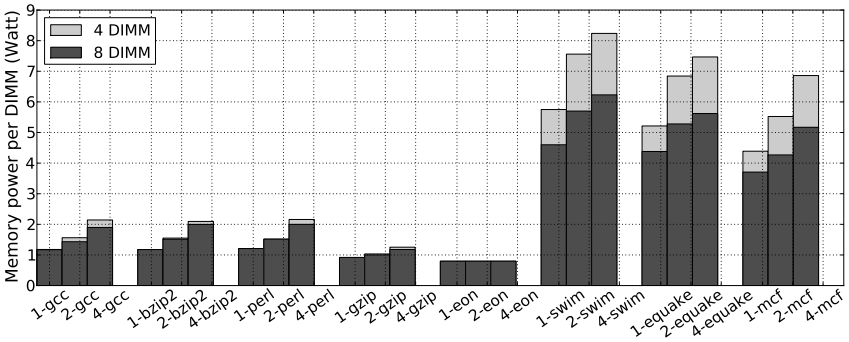
$$\frac{dT_j^D(t)}{dt} = -\frac{T_j^D(t)}{\tau_{ca}^D} + \frac{\gamma}{C_{ca}^D}(P^D + \frac{q^D}{\gamma}) \tag{5.2}$$

where $\gamma = (1 + \frac{R_j^D}{R_{ca}^D})$, $R_j^D = \frac{R_{chip}^D}{Nn_r}$, P^D represent the total operational power dissipated in the DIMM, τ_{ca}^D is the time constant of the heat spreader component which equals to $C_{ca}^D R_{ca}^D$, where R_{ca}^D is the case to ambient thermal resistance which is the sum of R_s^D and R_{conv}^D.

Experimental verification of DIMM thermal model We run memory intensive benchmarks, *swim* and *mcf* from SPEC2K, and collected the DIMMs junction temperature traces using IPMI. Figure 5.4 shows the temperature traces of 4 DIMMs and 8 DIMMs configurations to show that our model is not limited to a



(a) Total power of memory DIMMs



(b) Per DIMM power

Figure 5.5: Memory power. The number in front of the benchmark indicates the number of instances we run

specific setup. The 4 DIMMs and 8 DIMMs are placed in the memory slots following the order, (A_1, B_1, C_1, D_1) and $(A_1, A_2, B_1, B_2, C_1, C_2, D_1, D_2)$ respectively as shown in Figure 5.1(b). The fan speed is set to maximum using boost mode option to keep the time constant of the heat spreader of the memory modules constant. The figure shows clearly that the DIMM temperature dynamics is dominated by the heat spreader temperature since it changes slowly, with a time constant of 70 seconds. We plotted the transient temperature of the RC model and compared it with the actual temperature readings to show that the case to ambient temperature changes can be modeled using a single RC node as shown in Figure 5.4. The results show a strong match between the actual and ideal model with average error of 0.27^{o}C relative to the real measurements.

5.1.3 Sources of energy savings in memory

The DRAM energy consumed to do the actual job of reading or writing data is only a fraction of the total DRAM energy. This is because DIMMs consume energy, baseline energy, to keep state of the stored information and to provide sufficient responsiveness to the incoming requests (e.g. refresh, PLL, etc ...) [28]. We can save energy by consolidating the active memory pages of the workload into a smaller set of DIMMs and place the rest in a self-refresh mode, which consumes around 1% of active mode [28]. However, this needs to be done by taking the thermal issues into account.

Figure 5.5(a) shows the total memory power for two memory configurations. In this experiment, we distribute workload across the dual sockets in a load balanced way. In the first configuration, we use 4 DIMMs where each is connected to a separate memory channel to maximize bandwidth. For the second case, we use 8 DIMMs with 2 DIMMs per channel. The results show that savings of up to 16W can be achieved with consolidation, with higher savings are when the system has more DIMMs. The increase in total power in the case of 8 DIMMs indicate that a fraction of the memory power is used to keep the DIMMs functional, which we call *baseline power*. The savings are higher for memory intensive workloads (*swim, equake, mcf*) compared to CPU intensive ones (*eon, gzip, perl, bzip2, gcc*) as expected. Figure 5.5(b) shows the impact of consolidation on power per DIMM. The power density of DIMMs increases by up to 33% which can lead to potential temperature problems and higher cooling costs. It should be noted that the memory controller employs interleaving policy to maximize the use of all channels bandwidth by evenly distributing memory accesses and memory pages across the channels.

Next we address the impact of consolidation on performance. We performed experiments on our machine using different DIMMs organizations as follows: (a) single DIMM, (b) two DIMMs where each DIMM is connected to a memory channel that belongs to a separate memory controller, (c) four DIMMs where each DIMM is connected to a separate memory channel, (d) eight DIMMs where every 2 DIMMs are connected to a separate memory channel. Figure 5.6 shows the effect of DIMM

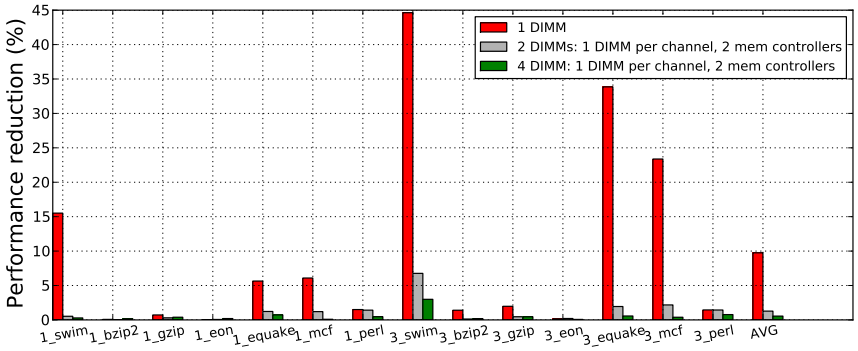
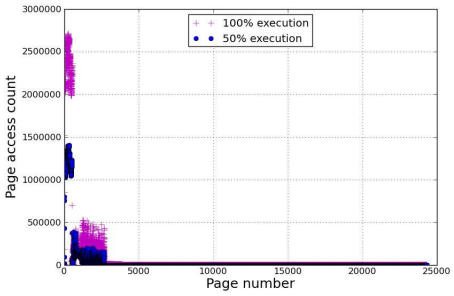
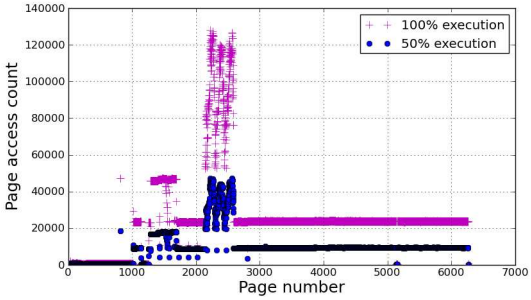


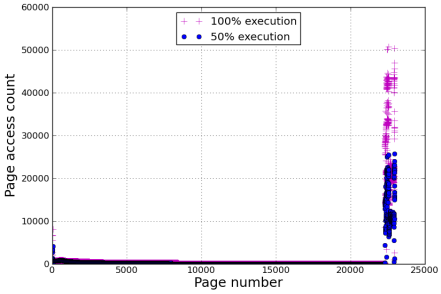
Figure 5.6: Performance reduction with consolidation



(a) mcf (memory bound)



(b) equake (memory bound)



(c) bzip2 (CPU bound)

Figure 5.7: Memory page access pattern

consolidation on performance compared to a system with 8 active DIMMs. For the case of using 1 and 2 DIMMs the performance degradation is noticeable since only a fraction of the bandwidth is utilized. However, when we fully utilize the memory bandwidth (as in the case of 4 DIMMs) the resultant performance is close to the baseline case. This slight improvement in the case of 8 DIMMs compared to 4 DIMMs is related to the reduction in banks conflicts as we are adding more banks. Nevertheless, this improvement is small since the number of pages (usually order of thousands or more) is much larger than the number of banks (order of tens). Hence, little improvement in temporal locality can be attained. This indicates that the performance overhead due to consolidation is in the acceptable range assuming that we are using the entire memory bandwidth and the memory pages can fit in the consolidated DIMMs.

To ensure the page migration policy is beneficial, we need to have fast and energy efficient migrations. Figures 5.7(a), 5.7(b), 5.7(c) show the page access patterns of memory and CPU intensive applications. We use M5 a micro-architectural simulator [44], to generate these statistics. We simulate for a representative period of 15 billion instructions of each benchmark. Each graph shows the access distribution for half and full of the simulated interval to examine the changes in the access pattern. These statistics are obtained based on the number of accesses to the physical addresses of the memory pages . From these results, it can be seen that the number of active pages (*hot pages*) is a small fraction of the total number of pages of the application. The hot pages are distributed uniformly across the active memory modules due to the interleaving policy of the memory controller. The average time between accessing different hot pages is usually in the order of microseconds which is thousands times smaller than the temperature transient time of memory. This indicates that we can capture sufficient number of hot pages to migrate during execution quickly enough to resolve thermal emergencies.

The energy savings from consolidation are expected to scales as DRAM technologies advance. The savings mainly depend on the parameters that affect the baseline power: operating frequency, supply voltage and memory density. The baseline power is expected to increase linearly with frequency and device density

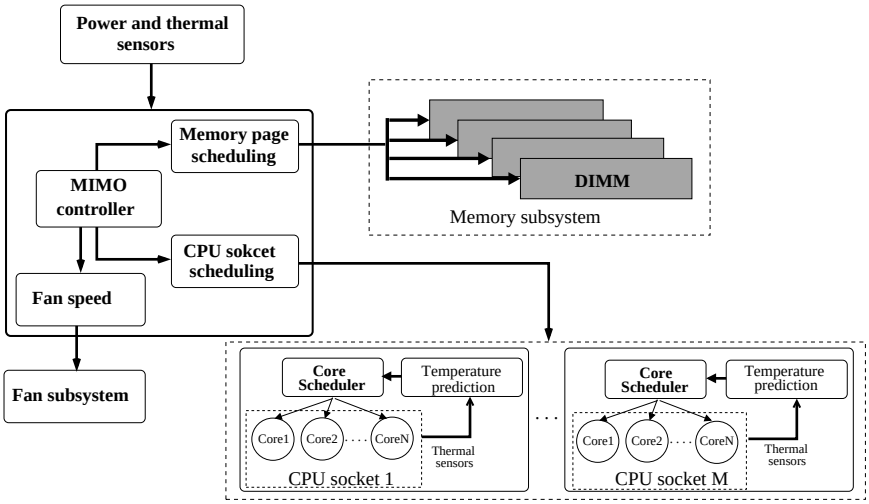


Figure 5.8: Overview of our Integrated Memory-CPU Management

& quadratically with the operating voltage. The current trend in increasing the frequency and memory density can offset the reduction in operating voltage that keeps the consolidation concept effective as DRAM technologies advance. The upcoming DDR4 memory is expected to support DVFS to optimize power by lowering the frequency when there are infrequent accesses. As the modern memory subsystem is interleaved, the chances are dramatically reduced for memory to benefit from DVFS as it reduces the intervals between two consecutive memory accesses. In contrast, clustering the hot pages to a subset of the DIMMs allows for more savings opportunities in the DIMMs the hold dormant pages. The exploitation of this DVFS technology is part of our future work.

5.2 Combined Energy, Thermal and Cooling Management

In this section we introduce our joint energy, thermal and cooling management for Memory-CPU subsystems, CETC. Figure 5.8 illustrates our framework which consists of formal multi-input multi-output (MIMO) controller, actuators (memory page scheduler, CPU socket scheduler, fan speed actuator) and sensors

(temperature and power). Actuators translate the MIMO controller's decisions into proper actions on memory, CPU sockets and fan. The controller take feedback signals using thermal and power sensors from the CPU and memory subsystems. We designed our scheme in a unified fashion since the temperature of memory is dependent upon the temperature of CPU. Two independent thermal management units for CPU and memory lead to inefficiencies. Formal control is used to guarantee stability. The MIMO controller is implemented in the operating system layer. This approach can also be extended to virtualized environments. In this case, the MIMO controller needs to be implemented inside the virtual machine manager, hypervisor, which manages the allocation of the physical resources in the system. We design the MIMO controller using state-space control mechanism because it is robust and scalable [26].

5.2.1 State-space control

We formulate a unified state-space model for memory and CPU subsystems. The vector of junction temperature in memory, $[T^D_{j_1}(t), T^D_{j_2}(t), \ldots, T^D_{j_{n_D}}(t)]^T$, is defined as $\mathbf{T}^D_j$. Here n_D is the number of DIMMs. The vector of heat sources in DIMMs, $[P^D_1(t) + \frac{q^D_1(t)}{\gamma_1}, P^D_2(t) + \frac{q^D_2(t)}{\gamma_2}, \ldots, P^D_{n_D}(t) + \frac{q^D_{n_D}(t)}{\gamma_{n_D}}]^T$, is defined as $\mathbf{U}^D$. Here $P^D_i(t)$ and $q^D_i(t)$ correspond to the sources of DIMM power and heat coupling respectively. The value of γ_i is defined in equation (5.2) as $1 + \frac{R^D_{j_i}}{R^D_{ca_i}}$. In case of no thermal coupling we can set $q^D_i(t)$ to zero, for $1 \leq i \leq n_D$.

Using equation (5.2) we can express the thermal model for memory subsystem as:

$$\frac{d\mathbf{T}^D_j(t)}{dt} = \mathbf{Y}^D\mathbf{T}^D_j(t) + \mathbf{Z^D}\mathbf{U}^D(t) \tag{5.3}$$

where temperature coefficient $\mathbf{Y}^D$ and power coefficient $\mathbf{Z}^D$ are defined as:

$$\mathbf{Y}^D = \begin{bmatrix} \frac{-1}{\tau^D_{ca_1}} & & 0 \\ & \ddots & \\ 0 & & \frac{-1}{\tau^D_{ca_{n_D}}} \end{bmatrix} \qquad \mathbf{Z}^D = \begin{bmatrix} \frac{\gamma_1}{C_{ca_1}} & & 0 \\ & \ddots & \\ 0 & & \frac{\gamma_{n_D}}{C_{ca_{n_D}}} \end{bmatrix}$$

Similarly, we can develop the thermal model for a set of CPU sockets. The socket to ambient temperature vector, $[T^C_{ca_1}(t), T^C_{ca_2}(t), \ldots, T^C_{ca_{n_c}}(t)]^T$, is defined as

$\mathbf{T}^C_{ca}$. Here n_C is the number of CPU sockets. The vector for instantaneous power dissipated in CPU sockets, $[P^C_1(t), P^C_2(t), \ldots, P^C_{n_C}(t)]^T$, is defined as $\mathbf{P}^C$. Using (4.5), the CPU thermal model becomes:

$$\frac{d\mathbf{T}^C_{ca}(t)}{dt} = \mathbf{Y}^C\mathbf{T}^C_{ca}(t) + \mathbf{Z}^C\mathbf{P}^C(t) \tag{5.4}$$

where temperature coefficient $\mathbf{Y}^C$ and power coefficient $\mathbf{Z}^C$ are diagonal matrices. $Y^C_{ii} = \frac{-1}{\tau^C_{ca_i}}$ and $Y^C_{ii} = \frac{1}{C^C_{ca_i}}$, for $1 \leq i \leq n_C$. The continuous systems given in (5.3) and (5.4) can be discretized using the transformations given in [26] as follows:

$$\mathbf{T}^D_j(k+1) = \mathbf{\Phi}^D\mathbf{T}^D_j(k) + \mathbf{\Gamma}^D\mathbf{U}^D(k) \tag{5.5}$$

$$\mathbf{T}^C_{ca}(k+1) = \mathbf{\Phi}^{\mathbf{C}}\mathbf{T}^C_{ca}(k) + \mathbf{\Gamma}^C\mathbf{P}^C(k) \tag{5.6}$$

where the temperature and power coefficients for these discretized systems are defined as follows:

$$\mathbf{\Phi}^D = e^{\mathbf{Y}^D\Delta t} \tag{5.7}$$

$$\mathbf{\Phi}^C = e^{\mathbf{Y}^C\Delta t} \tag{5.8}$$

$$\mathbf{\Gamma}^D = \mathbf{Z}^D\int_0^{\Delta t} e^{\mathbf{Y}^D u}\,du \tag{5.9}$$

$$\mathbf{\Gamma}^C = \mathbf{Z}^C\int_0^{\Delta t} e^{\mathbf{Y}^C u}\,du \tag{5.10}$$

The coefficient in this discretized system are diagonal because they are function of diagonal coefficients in the continuous system. Substituting $\mathbf{Y}^D$ and $\mathbf{Z}^D$ in (5.7) and (5.9), we get:

$$\mathbf{\Phi}^D = \begin{bmatrix} e^{-\frac{\Delta t}{\tau^D_{ca_1}}} & & 0 \\ & \ddots & \\ 0 & & e^{-\frac{\Delta t}{\tau^D_{ca_{n_D}}}} \end{bmatrix}$$

$$\mathbf{\Gamma}^D = \begin{bmatrix} \frac{\gamma_1}{C_{ca_1}}(\int_0^{\Delta t} e^{-\frac{u}{\tau^D_{ca_1}}}\,du) & & 0 \\ & \ddots & \\ 0 & & \frac{\gamma_{n_D}}{C_{ca_{n_D}}}(\int_0^{\Delta t} e^{-\frac{u}{\tau^D_{ca_{n_D}}}}\,du) \end{bmatrix}$$

The integrals of $\mathbf{\Gamma^D}$ have a simple analytical solutions as, $\mathbf{\Gamma}^D_{ii} = (R^D_{ca_i} + R^D_{j_i})(1 - e^{-\frac{\Delta t}{\tau^D_{ca_i}}})$ for $1 \leq i \leq n_C$. $\mathbf{\Gamma^C}$ and $\mathbf{\Phi^C}$ can be computed similar way.

After considering thermal dependency between CPU and memory, we formulate the unified state-space model using (5.5) and (5.6), which yields:

$$\begin{bmatrix} \mathbf{T}^C_{ca}(k+1) \\ \mathbf{T}^D_j(k+1) \end{bmatrix} = \begin{bmatrix} \mathbf{\Phi}^C & \mathbf{0} \\ \mathbf{\Phi}^{CD} & \mathbf{\Phi}^D \end{bmatrix} \begin{bmatrix} \mathbf{T}^C_{ca}(k) \\ \mathbf{T}^D_j(k) \end{bmatrix} + \begin{bmatrix} \mathbf{\Gamma}^C & \mathbf{0} \\ \mathbf{0} & \mathbf{\Gamma}^D \end{bmatrix} \begin{bmatrix} \mathbf{P}^C(k) \\ \mathbf{P}^D(k) \end{bmatrix} \tag{5.11}$$

where $\mathbf{P}^D(k)$ corresponds to the vector of DIMM's power consumption. The terms, $\mathbf{\Phi}^{CD}$, represents the thermal coupling coefficient from CPU sockets to memory. $\mathbf{\Phi}^{CD}_{ij} = \frac{\lambda_{ij}\mathbf{\Gamma}^D_{ii}}{R^D_{ca_i}+R^D_{j_i}} \frac{R^C_{conv_j}}{R^C_{ca_j}}$, for $1 \leq i \leq n_C$ and $1 \leq j \leq n_D$, where λ_{ij} represents the thermal coupling factor between the DIMM i and the CPU socket j.

MIMO controller We use a MIMO controller as the primary management unit which is implemented in the operating system layer. With a final target of minimizing the cost of cooling and operational energy, the MIMO controller takes all the important decisions, e.g. (a) memory module activation or deactivation, (b) workload assignment or scheduling between or within sockets, and (c) fan speed adjustment, while ensuring convergence to the target temperatures of both CPU and memory subsystems with minimal performance overhead. To make these decisions intelligently, the controller takes input from thermal & power sensors in CPU & memory and readings of fan speed. These sensors are commonly present in the state of the art servers. As output it provides a vector of desired power distributions in CPU & memory, along with a temperature vector that is used to determines fan speed. There are three actuators: CPU, memory and fan. Actuators ensure that fan speed, power of CPU and memory are set according to the controller's output.

The controller is designed based on the linear feedback control law as this is a liner system [26], using $\mathbf{G}$ and $\mathbf{G_0}$, gain matrices with dimensions $n = n_C + n_D$. Gain matrices are computed to ensure stability as discussed in the subsequent paragraphs. The control law is applied to the combined state-space model (5.11) which yields,

$$\mathbf{P}(k) = -\mathbf{G}\mathbf{T}(k) + \mathbf{G_0}\mathbf{T}_r(k) \tag{5.12}$$

$\mathbf{P}(k)$ is the output of the controller and represents the power consumption vector of CPU sockets and memory modules that needs to be maintained by the actuators in the period between (k and $k+1$). Fan speed is set by the fan actuator based on the desired temperature distribution specified by the controller. The controller's temperature vector can be calculated by substituting (5.12) in (5.11), as:

$$\mathbf{T}(k+1) = (\mathbf{\Phi} - \mathbf{\Gamma}\mathbf{G})\mathbf{T}(k) + \mathbf{\Gamma}\mathbf{G_0}\mathbf{T}_r(k) \tag{5.13}$$

Vector $\mathbf{T}(k) = [\mathbf{T}^C(k), \mathbf{T}^D(k)]^T$, represents the thermal states of the system consisting of CPU, $\mathbf{T}^C(k)$, and memory temperatures, $\mathbf{T}^D(k)$. The target temperature vector $\mathbf{T}_r(k) = [\mathbf{T}^C_r(k), \mathbf{T}^D_r]^T$ consists of target CPU, $\mathbf{T}^C_r(k)$, and memory temperatures, $\mathbf{T}^D_r$. The elements of $\mathbf{T}^D_r$ are the thermal emergency thresholds of the memory modules. The components of $\mathbf{T}^C_{r_i}(k)$ are calculated as follows:

$$\mathbf{T}^C_{r_i}(k) = \mathbf{T}^C_{ca_i}(k) + \Delta T_{ca_i}(k) - \delta T_{th_i}(k) \tag{5.14}$$

where $\Delta T_{ca_i}(k) = \Delta R_{ca_i}(k)\frac{T^C_{ca_i}(k)}{R^C_{ca_i}(k)}$. $\Delta R_{ca_i}(k)$ is the change in the CPU case to ambient resistance which relates to the difference between the current fan speed and the target speed. The difference between the junction temperature to the threshold is represented by $\delta T_{th_i}(k)$.

In general, the controller guarantees convergence to the desired target values, $\mathbf{T}_r(k)$, if the eigenvalues of the controlled feedback system are within the unit circle [26]. One way to determine the feedback gain matrix, $\mathbf{G}$, is to use the desired eigenvalues as an input and calculate $\mathbf{G}$ accordingly. To obtain the optimal gain matrix we can use the linear quadratic regulator (LQR) optimization [26]. This method calculates the gain matrix in a way that minimizes the following cost function:

$$J = \sum_{k=0}^{\infty}[\mathbf{T}^T(k)\mathbf{Q}\mathbf{T}(k) + \mathbf{u}^T(k)\mathbf{R}\mathbf{u}(k)] \tag{5.15}$$

where $\mathbf{u} = -\mathbf{G}\mathbf{T}(k)$. $\mathbf{Q}$ and $\mathbf{R}$ are symmetric weight matrices of size $n \times n$ and they are specified by the designer. They are selected based on the importance of the

states and the energy of the control outputs respectively. The LQR computations are done off-line. It takes around one second to compute a gain matrix for different fan speeds. The gain matrix is stored in an array and accessed at run time. The input gain matrix $\mathbf{G_0}$, it calculated using the standard reference input method that is described in [26]. The controller interval is on the order of several seconds since the thermal time constant of the DIMMs and the CPUS is on the order of tens of seconds.

5.2.2 Actuators

At each controlling tick, k, MIMO controller determines the next set of operating power values for the CPUs, memory modules and fan speed for the next interval (k and $k+1$). The output of the controller is communicated to the actuators to execute the controller requests. Each actuator operates independently as the MIMO controller already considers the thermal dependencies between the components.

Controller convergence in the face of errors: Actuator's goal is to minimize the differences between the calculated power values provided by the controller and the current measured ones. When the action of the actuator has some small errors within a given threshold then no correction is needed. The cost of this is a slight decrease in energy savings. The controller can ensure convergence even when some errors exceed the threshold occasionally. To validate our assumptions we assume a state space model that is similar to the CPU and memory. We use a CPU model as an illustrative example:

$$T^C_{ca}(k+1) = \Phi T^C_{ca}(k) + \Gamma P^C(k) \tag{5.16}$$

where Φ and Γ are coefficients and P^C is the input power of a given CPU. To control this system we use state space control that we have in (5.12). Lets assume a reference point to be 0 to simplify the analysis. The feedback control in this case can be computed as: $P^C(k) = -GT^C_{ca}(k) + e(k)$. G is the gain and $e(k)$ is the actuator average error. The accumulated errors in $T^C_{ca}(k)$ at $k = n$

equals to $\sum_{i=0}^{n} \Gamma e(i) \upsilon^{n-i}$. Where υ is the eigenvalue of the controller. Since the eigenvalue of the controller is less than 1, the accumulated errors converge to 0 which ensures convergence. The details of the actuators implementation are below.

CPU actuator: Controller's input to CPU actuator is the vector of the desired power values for each of the CPUs in the interval between k and $k+1$. This algorithm calculates the vector ΔP^C by subtracting the controller requested CPU power from the CPUs average power measured in the interval between $k-1$ and k. The details of the CPU scheduler are given in *Algorithm* 2. The algorithm starts by estimating the power consumed by the individual threads in each CPU by modeling their core and last level cache power similar to what is proposed in section 4.2.6. The core dynamic power is modeled using a regression model that is based on instruction per cycle (IPC) metric while leakage power is calculated based on the temperature. The last level cache is calculated by multiplying the access rate per second with the power consumed to access a cache block. Subsequently, our algorithm traverses the workload and spreads the threads from the hot CPU starting with cooler threads to have finer grain control of the total power on each socket and to reduce the chance of errors. It moves from a hot CPU_i a number of threads that have total power that that is less than or equal to ΔP^C. A cool CPU_j gets threads with total power is less than or equal to ΔP_j^C. Before each migration we evaluate the cooling energy savings to prevent ineffective scheduling decisions similar to method given in section 4.2.5. The cooling energy savings estimator calculates the resultant temperature after the migration using our thermal model. Subsequently, we translate the difference in maximum temperature into change in fan speed using the equation (5.20) which we use in the fan actuator. Individual scheduling events are allowed only when the predicted cooling savings are higher than a given threshold, S_{min}.

Memory actuator: at the beginning of each interval, controller provides power vector, $\mathbf{P}^D$ of power dissipation per each DIMM module. As a result, a DIMM may need to either increase or decrease its power dissipation. In both cases migration of pages is used if possible. However, when there are no active DIMMs

Algorithm 2 Socket level scheduling

```
1: Calculate ΔP^C and the set of P^C_thr for each CPU. Set Q as an empty queue
2: for i in set of hot CPUs do
3:     for j in unmarked threads in CPU i do
4:         dest ⇐ index of coolest CPU (min(ΔP^C))
5:         if (P^C_thr_j ≤ ΔP^C_i and P^C_thr_j ≤ −ΔP^C_dest and min(ΔP^C) < 0) then
6:             if CPU dest has idle core then
7:                 Calculate cooling savings of migrating thread j to CPU dest
8:             else
9:                 Calculate cooling savings of swapping thread j with the coolest thread from CPU dest
10:            end if
11:            enqueue this migration event Q and threads allocation status
12:            Update ΔP^C and threads assignment
13:        end if
14:    end for
15: end for
16: Execute all migration events in Q
```

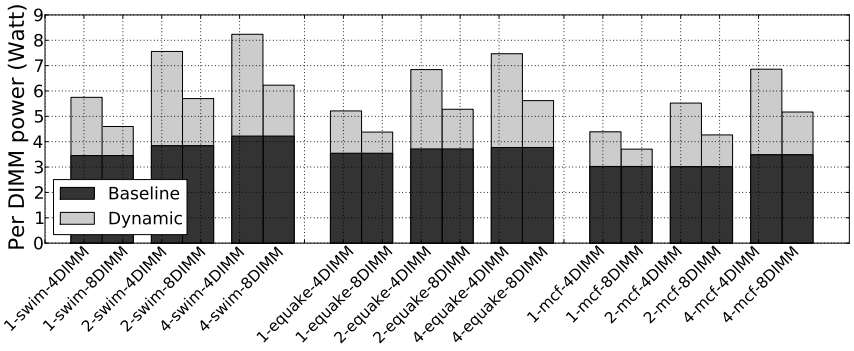


Figure 5.9: Power breakdown of memory DIMMs

that can accept additional pages, then we need to choose between activating a new DIMM or letting the fan spin up to cool the DIMMs. We decide between these two options based on the temperature reduction each choice can deliver under the same energy budget. In the following subsections we study these scenarios:

1. Increasing fan speed versus activating a new DIMM: Figure 5.9 shows that doubling the number of DIMMs does not reduce the power per DIMM to half. This is due to the baseline power P_{base} component that is in the range of 3.5 W for memory bound applications measured on 4GB DDR2 DIMMs (please refer to section 5.1.3). This means that increasing the fan speed may be a better option if it results in a lower temperature at power consumption of P_{base}. Increasing the fan

speed reduces the memory temperature by lowering the effect of thermal coupling and self heating of the DIMMs as it reduces the convective resistance of both the CPU and memory. The temperature reduction for increase in fan power by P_{base} can be computed as:

$$\Delta T_{fan} = \Delta F(\lambda P^C_{act} \frac{dR^C_{conv}(F)}{dF} + P^D_{act} \frac{dR^D_{conv}(F)}{dF}) \tag{5.17}$$

where ΔF is the increase in fan speed of the hottest memory zone. P^C_{act} and P^D_{act} represent the actual CPU and hottest DIMM power that is located in the CPU zone respectively. When the actuator decides to activate a new DIMM, it migrates pages from other DIMMs to the newly activated DIMM progressively from the application starting with the one that has the highest memory access to reduce power density. Using our thermal model, the temperature reduction, i.e. $|\Delta T_{mem}|$, for adding one more DIMM can be computed as:

$$\Delta T_{mem} = \frac{(T^D_{j_{max}} - P_{base} R^D_{ca} - \lambda T^C_{ha})}{n_D} \tag{5.18}$$

where n_D is the number of DIMMs after expansion and λ is the thermal coupling factor between CPU and memory. The $T^D_{j_{max}}$ is the maximum temperature of the DIMMs that are located in the zone that is associated with a CPU. T^D_j is the junction temperature of the DRAM chip. The controller choses to activate a new DIMM only if the temperature reduction, $|\Delta T_{mem}|$, is higher than $|\Delta T_{fan}|$ when a new DIMM is activated, as follows:

$$\begin{aligned} &\textbf{if}(|\Delta T_{fan}| < |\Delta T_{mem}|) \;\Rightarrow\; \textit{activate a new DIMM} \\ &\textbf{else} \;\Rightarrow\; \quad \textit{increase fan speed} \end{aligned} \tag{5.19}$$

2. Controlling DIMMs power by page migration: Power vector $\mathbf{P}^D$ specifies power distribution of each DIMM over the next period. If a memory module, DIMM_i, needs its power to be reduced in the next interval when $\mathbf{P}^D_i(k) < \mathbf{P}^D_{meas_i}(k-1)$ ($\mathbf{P}^D_{meas_i}(k-1)$ measured power in the previous interval) this is then equivalent to $\mathbf{T^D}_i(k+1) < \mathbf{T^D}_i(k)$. This power reduction in DIMM_i is achieved by migrating portion of its pages to other DIMMs. This migration is performed by using the

memory actuator. In contrast, the power dissipation of a memory module, say DIMM$_i$, may also need to be increased when $\mathbf{P^D}_i(k) > \mathbf{P}^D_{meas_i}(k-1)$ which is equivalent to $\mathbf{T^D}_i(k+1) > \mathbf{T^D}_i(k)$. Hence it requires migrating more pages to DIMM$_i$ from other memory modules to match DIMM$_i$'s power with the desired value by the controller. Whenever a page is migrated from DIMM$_i$ to DIMM$_j$, the virtual to physical address mapping for the page is updated in the page table of the operating system.

When all active memory modules can accept more pages, the controller starts migrating from the most recently activated module so that it can put it in low power mode when no active pages are left and save energy. When there are some hot DIMMs, the controller tries to maximize performance by migrating pages to a memory module only if it has lower memory access rate/power consumption than the average value. In this way, all the accesses are uniformly distributed among the active memory modules. To minimize hot spots, the controller balances the number of active DIMMs per CPU zone (CPU and its associated downstream DIMMs). For example, the activation order of DIMMs shown in Figure 5.1(b) should be $A_i, B_i, C_i, D_i, A_{i+1}, B_{i+1}$ etc. The page migration continues until the access rate of the newly activated DIMM becomes equal to that of already active DIMMs. The overhead of migrating pages is acceptable since the difference between the temperature time constant and page migration time is over 6 orders of magnitude.

Fan actuator: Fan speed is updated periodically based on the requested temperatures from the controller, $\mathbf{T}(k+1)$. Controlling the fan is used as a complementary measure to the CPU and memory optimizations to minimize the remaining difference between the current measured temperature and the controller's requested temperature vector.

Let's assume $\mathbf{\Delta T}^D > \mathbf{0}$ and $\mathbf{\Delta T}^C > \mathbf{0}$ are vectors of the temperature difference between the current and target value of the controller for memory modules and CPUs that belong to the same cooling zone respectively. The actuator estimates the new fan speed F_{new} based on the current fan speed, F_{cur}, and highest

requested change in fan speed due to memory, ΔF^D, and due to CPU, ΔF^C, as follows:

$$\Delta F^D = max(\frac{\mathbf{\Delta} T_i^D}{\lambda P_j^C \frac{dR_{conv_j}^C(F)}{dF} + P_i^D \frac{dR_{conv_i}^D(F)}{dF}}) \tag{5.20}$$

$$\Delta F^C = \frac{\mathbf{\Delta} T_j^C}{P_j^C \frac{dR_{conv_j}^C(F)}{dF}} \tag{5.21}$$

$$F_{new} = F_{cur} + max(\Delta F^D, \Delta F^C) \tag{5.22}$$

where i represents the ith DIMM that are in the zone of processor j. The interval for the fan actuator can be set to be equal to the controller interval or smaller. Setting the fan actuator interval to a smaller value may help to provide more detailed control over temperature. The desired temperature at the subintervals is computed based on the slop of $\mathbf{T}(k)$ and $\mathbf{T}(k+1)$.

5.3 Evaluation

5.3.1 Methodology

In this study we run the workload in our Intel server and collected real power traces from the memory modules. These power traces are used to estimate temperature of the memory and CPU using our extended version HotSpot simulator [57]. We measure the CPU core, L2 and baseline power using the method described in 4.2.6. The core and L2 power traces are used to estimate the core and L2 temperatures respectively using HotSpot simulator and Xeon processor layout. We include the baseline power of the CPU in the temperature simulations since it impacts the heat spreader and the heat sink temperature. We also account for leakage power using the model given in section 4.2.6.

The memory organization is assumed to be similar to our Intel server (see Figure 5.1(b)). The memory controller put the inactive DIMMs in a self-refresh mode and the rest in the active mode. Power consumption during self-refresh

and transition penalty are given in Table 5.1. We use M5 a micro-architectural simulator [44], to generate the page access statistics of the applications. M5's memory management unit performs the virtual to physical address mapping. This mapping is usually stored in the page table within the operating system. Whenever a page is migrated to a different DIMM the virtual to physical address mapping for the page is updated.

The characteristics of the fan and thermal package of both CPU and memory DIMMs that we use in thermal simulation are listed in Table 5.1. For the default *fan control algorithm* we assume a closed loop PI (Proportional and Integral) controller that is usually used in modern systems. The fan algorithm adjusts the fan speed in proportion to the heat level of their respective CPUs and the accumulated temperature errors. When the temperature is below a given threshold the fan is set to idle. We set the fan to trigger 5 degrees below the thermal threshold of the chip (please refer to Table 5.1) to allow for enough time to react to thermal emergencies since it is a mechanical device. We extracted the value of R_{ca}^{D} (case to ambient thermal resistance of a DIMM), and time constant using our transient model and measured temperature & power data and chip thermal resistance [6]. We use the built-in thermal sensors in DIMMs to collect temperature via IPMI. To model the effect of airflow, we measured the value of R_{ca}^{D} at different fan speeds

We use simulation instead of running our algorithms in real system due to a number of issues. The built-in fan control algorithm runs all the CPU fans at a single speed that is related to cooling needs of the hottest socket, which lead to over provisioning. The algorithm is implemented in a separate controller that we don't have access to. Consequently, not all the benefits of our algorithms can be manifested with using the built-in fan control algorithm.

We consider the local ambient temperature inside the server to be between 35-45^{o}C. The server inlet temperatures may be much higher than the room ambient [52]. The new trend in data centers is to increase the ambient temperature to lower the costs of the air conditioning system [8], which results in even higher inlet temperatures. Since the local ambient within the server is hotter than the inlet temperature, it is not uncommon for the local ambient temperature to reach up

Table 5.1: Characteristics of CPU, memory, thermal packages and cooling

CPU	CPU	Xeon E5440
	TDP	80W
	CPU frequency	2.8GHz
	Heat spreader thickness	1.5mm
	Case to ambient thermal resistance in K/W	$R^C_{ca} = 0.141 + \frac{1.23}{V^{0.923}}$, V: air flow in CFM, [2]
	Heat sink time constant at max air flow	25 seconds
	Temperature Threshold	90^oC [47]
DIMM	DIMM size	4GB
	Max DRAM power/DIMM	10W
	Case to ambient thermal resistance in K/W	$R^D_{ca} = 0.75 + \frac{45}{V^{0.9}}$ V: air flow in CFM
	Per chip thermal resistance in K/W	$R^D_{chip} = 4.0$ [6]
	Heat spreader time constant at max air flow	70 seconds
	Temperature Threshold	85^oC [36, 37]
	Self refresh power per DIMM	0.15W
	Transition between active and self-refresh modes	11us [28]
	Thermal coupling factor with CPU	0.65
Fan	Fan power per socket	29.4 W [5]
	Max air flow rate per socket	53.4 CFM [5]
	Fan steps	32
	Fan sampling interval	1 second
	Idle fan speed	10% of max speed

to 45^oC [57, 58].

For the memory and socket level thermal managers we set the scheduling interval to 4 seconds, since the thermal time constant of both CPU and memory is on the order of 10s of seconds. Our technique is not restricted to this interval and other intervals around this value can be used. The heat spreader is simplified to a single node because it behaves as an isothermal layer due to its high thermal conductivity. The interval of fan control is set to 1 second so as to allow the fan to for a detailed control over temperature to ensure reliability. We set the cooling savings threshold to a conservative value of 10%. For core level thermal management policy, we use our proactive thermal management which performs scheduling at each OS tick.

A set of benchmarks are selected that exhibits various levels of CPU intensity to emulate real life applications (see Table 5.4). We run each benchmark in the work set till its completion and repeat it until the end of simulation time. The evaluation time for our algorithms is set to 10 minutes after the warm-up period.

In our experiments we use a set of workloads that have a representative mix of CPU and memory bound applications. The list of the workload combinations that we use is given in Table 5.3.

CETC does an integrated energy, thermal and cooling management for CPU and memory. It uses a MIMO controller to manage CPU socket scheduling, memory module management and fan speed. It also implements core level management to reduce the temperature within the sockets. In CETC we set the minimum number of active DIMMs to 4 during clustering mode to exploit the full bandwidth as shown in Figure 5.1(b). We evaluate CETC against the following set of policies:

Dynamic Load Balancing, DLB: It is usually implemented in modern operating systems to enhance the utilization of the system resources. DLB policy performs thread migration to minimize the difference in task queue lengths of the individual cores [21]. Operating system initiates dynamic load balancing every hundred milliseconds. The balancing threshold is set to one to make the difference in task queue length equal to zero as possible. In this work we implement the DLB as our *default policy* for the purposes of comparison. DLB uses the default PI fan controller. It keeps all memory modules active. We choose this policy to show that balancing the utilization is not sufficient to mitigate the hot spots in the CPU subsystem. When the temperature of CPU or memory exceeds the threshold, their activity is throttled progressively depending upon the source of the problem [37]. This throttling is included in all the policies including CETC.

Dynamic Thermal Management of Core and Memory with PI fan control, DTM-CM+PI: This policy implements both core and memory state of the art thermal management techniques. Core level proactive thermal management optimizes dynamically the threads assignment within the individual sockets (see chapter 3). Memory thermal problems are managed separately by halting the accesses when the temperature exceeds thermal emergency level until it drops to the safe zone [37]. All memory modules stay active in this policy. The fan in this policy is controlled separately by the default PI fan controller. This policy is chosen to evaluate the savings using state of the art thermal management techniques.

No Fan-Memory Optimization, NFMO: This policy is like CETC except

Table 5.2: SPEC Benchmarks characteristics

Benchmark	IPC	Power per DIMM (Watt)	Characteristics	Benchmark	IPC	Power per DIMM (Watt)	Characteristics
swim	0.55	4.65	Memory bound	bzip2	1.36	1.18	CPU bound
equake	0.51	4.38	Memory bound	gcc	1.23	1.17	CPU bound
mcf	0.18	3.71	Memory bound	eon	1.33	0.79	CPU bound
perl	2.18	1.21	CPU bound	gzip	1.16	0.92	CPU bound

Table 5.3: Workload combinations for multi-tier algorithm

Workload	Socket A	Socket B	Workload	Socket A	Socket B
W1	*equake + 2gzip*	*3bzip2*	W9	*mcf + 2gcc*	*perl + bzip2 + eon*
W2	*2bzip2 + 2eon*	*mcf + gcc + 2gzip*	W10	*3gzip*	*2perl + eon*
W3	*equake + 2bzip2*	*2gzip + gcc*	W11	*2equake + gcc*	*2perl + equake*
W4	*perl + eon + bzip2*	*2equake + gcc*	W12	*mcf + 2gcc*	*2gcc + 2perl*
W5	*2gcc + perl*	*swim + 2gcc*	W13	*gcc + 2perl*	*equake + 2gcc*
W6	*mcf*	*mcf*	W14	*2mcf + gcc*	*2perl + mcf*
W7	*gcc + 2gzip*	*2bzip2 + perl*	W15	*2swim + gcc*	*2perl + swim*
W8	*perl + bzip2 + 2eon*	*2mcf + 2gcc*			

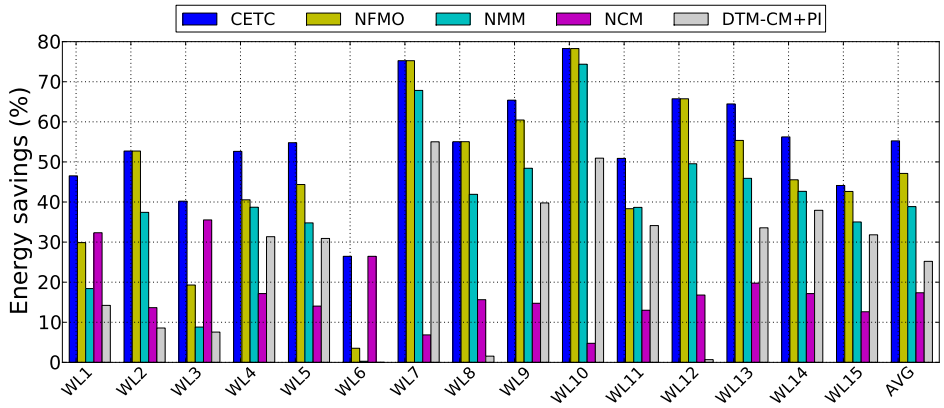
that we disable the actuator that performs the trade off between activating a new memory module and increasing the fan speed. We choose this policy to evaluate the impact of deactivating this optimization on the overall savings. The minimum number of active DIMMs is set just as in CETC.

No Memory Management, NMM: This policy is like CETC except all memory modules remain active the entire time. The purpose of this policy is to evaluate the savings that we can achieved with managing the thermal hot spots and optimizing the cooling costs in the CPU subsystem only.

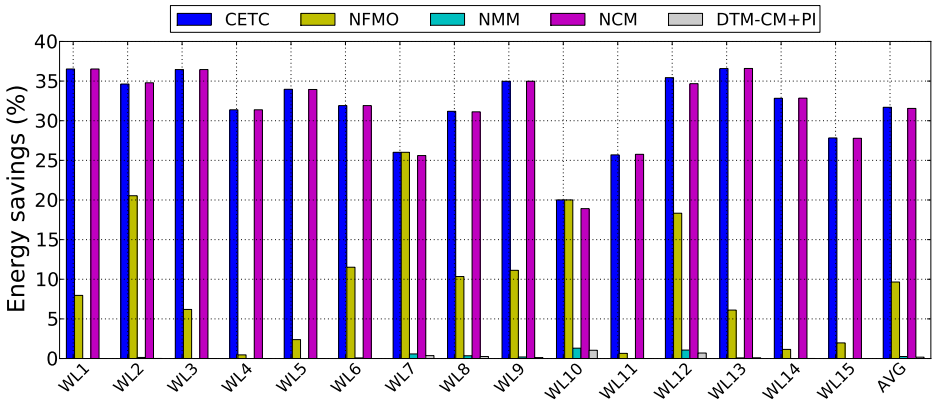
No CPU Migration, NCM: This policy disables the CPU management (socket and core scheduling) from CETC. We study NCM to show the impact of deactivating CPU policy on the overall cooling energy savings. The minimum number of active DIMMs is set as in CETC.

5.3.2 Results

We next compare our CETC algorithm to other policies. We evaluate energy savings, fan balancing, page migration rate, stability and overhead of applying our



(a) Local ambient temperature is 45^{o}C



(b) Local ambient temperature is 35^{o}C

Figure 5.10: Total energy savings (memory+cooling) relative to default dynamic load balancing in a system with 8DIMMs

policy. We study CETC with representative set of local ambient temperatures and number of DIMMs to have a thorough evaluation. The results indicate that CETC policy is able to achieve on average 70.1% combined cooling and memory energy savings at performance loss of less than 0.2%.

Energy savings The first set of results focuses on the total energy savings of CETC policy compared to the other policies. Our energy calculations include total energy of memory subsystem and cooling. Figure (5.10(a) and 5.10(b)) show the total energy savings in a system that has 8 DIMMs with 45^{o}C and 35^{o}C local server ambient temperature respectively. CETC achieved savings reaching an average of

55.3% and 31.7% relative to DLB for 45^{o}C and 35^{o}C respectively. Figure 5.10(a) clearly shows that CETC outperforms all other policies. For example, the case of workload W6 includes two mcf (memory bound) one on each socket, hence they produce comparable heat in each socket and their thermal coupling effect on the memory is comparable. The CETC savings in this case come from clustering the memory accesses to a smaller number of DIMMs since the thermal distribution across the sockets is balanced and there are no thermal problems in the CPUs. The policy NCM performs equally well to CETC since it optimizes for memory and there are no savings opportunities from the CPU side. The policies NMM and DTM-CM+PI deliver almost no savings since they do not optimize for memory energy. On the other hand, the savings opportunities in the cases of W7 and W10 are mainly related to imbalance in the thermal distribution between and within the two sockets which raise cooling energy costs. CETC performs quite well since it has the capability to balance the temperature between and within the CPU sockets. The policies NFMO and NMM perform equally well to CETC since they balance socket and core temperatures. On the other hand, DTM-CM+PI delivers a fraction of the CETS savings since it performs only local thermal optimizations. The policy NCM performs poorly in these cases since it targets lowering only memory energy. The other important scenario is when there are savings opportunities from CPU temperature imbalance and memory energy. Example of this can be seen in the cases of W4 and W5. In these cases, CETC is superior to all other policies since it is the only one that can capture these classes of savings at the same time.

Figure 5.10(b) shows the combined memory and cooling energy savings with local ambient temperature of 35^{o}C. In general, the savings are lower than the case of 45^o ambient temperature because most come from memory clustering as the system requires less cooling due to a lower local ambient temperature. The evidence of this can be seen from the savings of NMM policy which is close to zero while NCM perform closer to CETC. The CETC policy is able to perform well since it can capture this class of savings opportunities by clustering the memory access to a smaller set of DIMMs. These results also illustrate the benefits of the optimization that trade off between activating a new DIMM and speeding up the fan. The

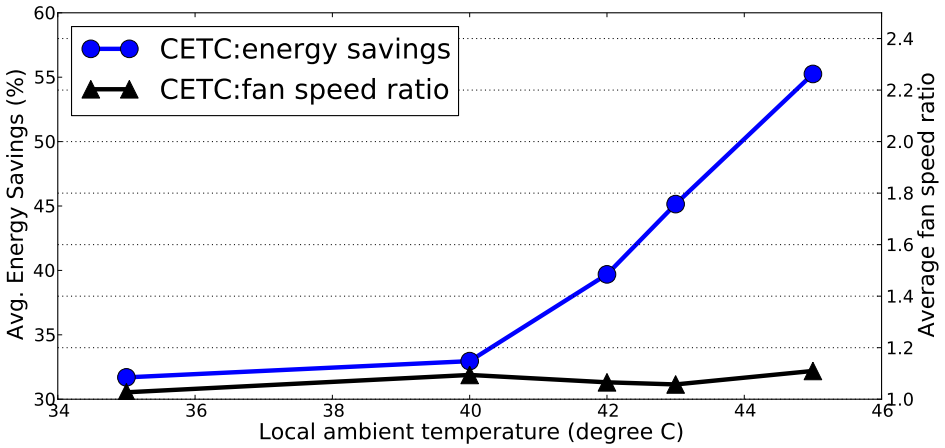
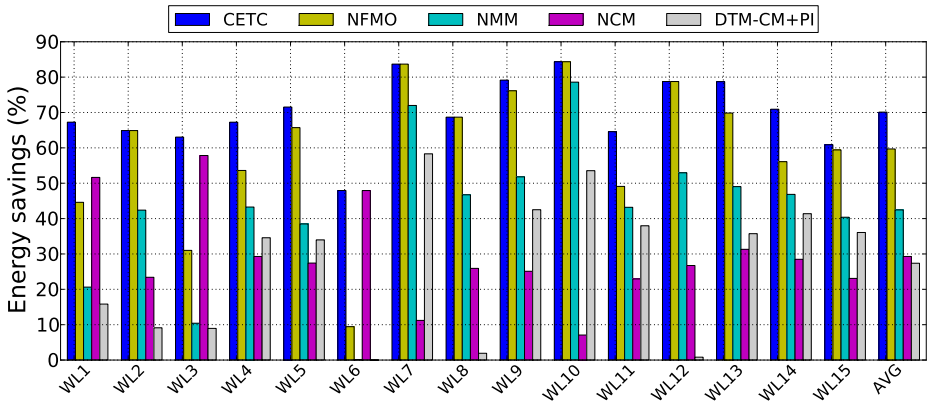


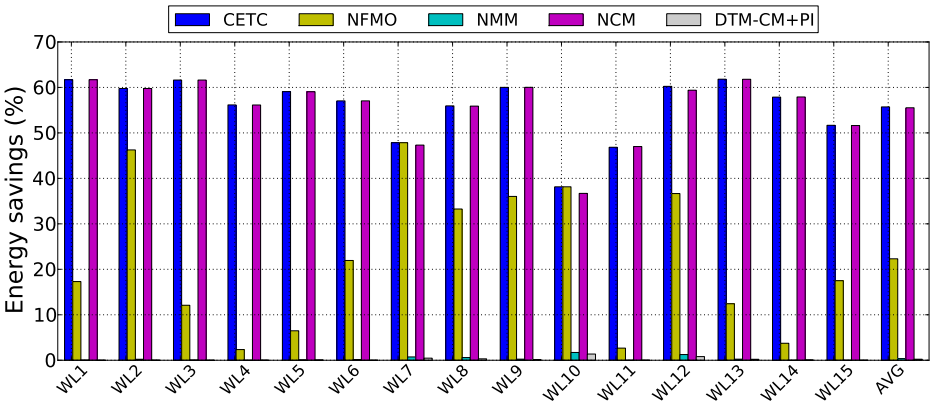
Figure 5.11: Temperature sensitivity for energy and fan speed with 8 DIMMs

evidence of this benefit can be indicated from the poor savings of NFMO since it does not perform this optimization. Since most of the savings come from clustering the memory modules then we expect that the savings would be lower when the workload exercises memory lightly. The cases of W7 and W10 illustrate this. In Figure 5.11 we give results that show the average combined memory and cooling energy savings of CETC as a function of local ambient temperature at a finer scale. The savings between 35^{o}C and 40^{o}C come primarily from clustering the memory accesses to a subset of the memory modules. When the local ambient temperature increases, the savings are higher due to balancing fan speed as both memory and CPU experience more thermal issues. This indicates that our technique provides better savings at higher local ambient temperature.

In Figures 5.12(a) and 5.12(b) we study the benefits of increasing the number of DIMMs to 16. The results show that CETC achieved higher savings reaching an average of 70.1% and 55.7% relative to DLB for 45^{o}C and 35^{o}C respectively. The savings of CETC increase when using 16 DIMMs as compared to the case of 8 DIMMs since more DIMMs can transition to low power modes. The results also show that the relative increase in savings is higher in the case of 35^{o}C compared to 45^{o}C for CETC. This is because the savings in the case of 35^{o}C are dominated by memory subsystem while the memory contributes to only a fraction of savings in the case of 45^{o}C. In summary, the energy savings results clearly show that



(a) Local ambient temperature is 45^{o}C



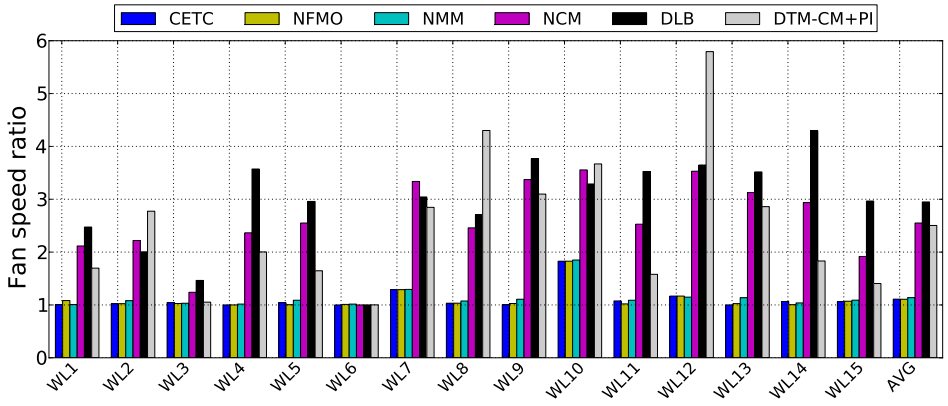
(b) Local ambient temperature is 35^{o}C

Figure 5.12: Total energy savings (memory+cooling) relative to default dynamic load balancing in a system with 16 DIMMs

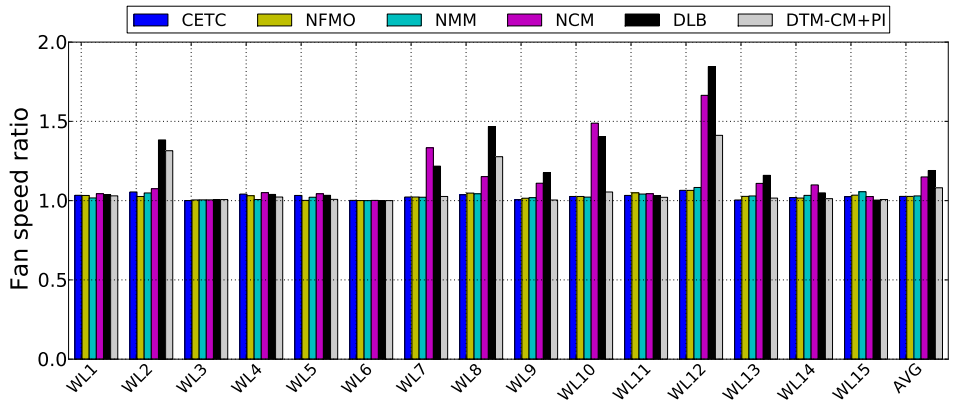
performing thermal management in a holistic fashion leads to large energy savings.

Fan balancing Next we evaluate the effectiveness of the socket level scheduling in balancing the fan speed to minimize cooling energy. The optimal ratio of fan speed in two sockets (we call it fan speed ratio) is 1. Figure 5.13(a) shows fan speed ratio in a system with 8 DIMMs and 45^{o}C local ambient temperature. The results show that CETC is able to reduce the standard deviation from the optimal fan speed ratio by 76% compared to the DLB policy. The reason that DLB has a high standard deviation from the optimal fan speed is because it does not account for heterogeneity in the workload in terms of the induced temperature in the CPU. The policy DTM-CM+PI performs poorly as well since it does not balance the temperature between CPU sockets. The fan imbalance with this policy can be even higher than DLB (e.g W8 and W12). This scenario occurs when core level scheduling is effective in one socket while it is not as effective in the other one (e.g. running CPU intensive workload). This increases the imbalance as the speed of one fan can be lowered while the other stays almost the same. The other polices that implements socket level thermal balancing perform close to CETC. The Figure 5.13(b) shows the fan speed ratio results in a system with 8 DIMMS and 35^{o}C ambient temperature. The reduction in standard deviation to the optimal target in this case is 93%. However, the actual benefits in the case of 45^{o}C ambient temperature are higher because the absolute difference between the speed of two fans is much higher in default policy. Figure 5.11 shows sensitivity analysis of average fan speed ratio with small grain changes in local ambient temperature for CETC. The results clearly show that CETC is able to keep the ratio very close to 1.0 in the entire temperature range between 35^{o}C and 45^{o}C.

In Figures 5.14(a), 5.14(b), 5.14(c) we study the benefits of CETC on balancing fan speeds for illustrative set of workloads W13, W11 and W15 respectively. In these experiments we run the workload for 200 seconds with DLB then we activate CETC for 600 seconds. We select illustrative workloads which represent a mix of cpu and memory intensive jobs to evaluate CETC effect on CPU, memory and the thermal coupling. The results show that using DLB can lead to a big imbalance in fan speeds (first 200 seconds of execution). It is clear from the results

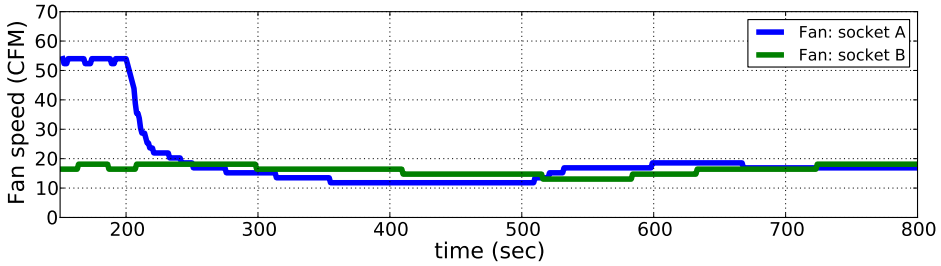


(a) Local ambient temperature 45^{o}C

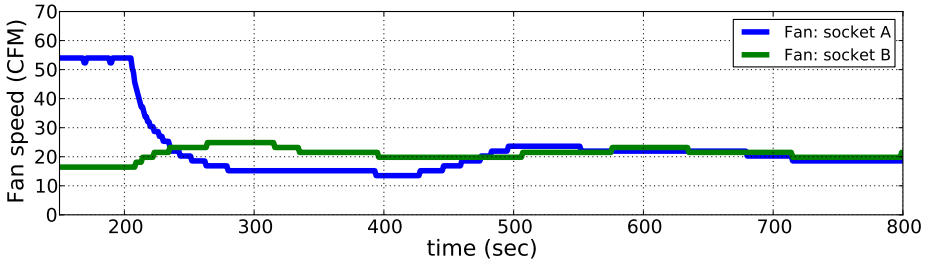


(b) Local ambient temperature is 35^{o}C

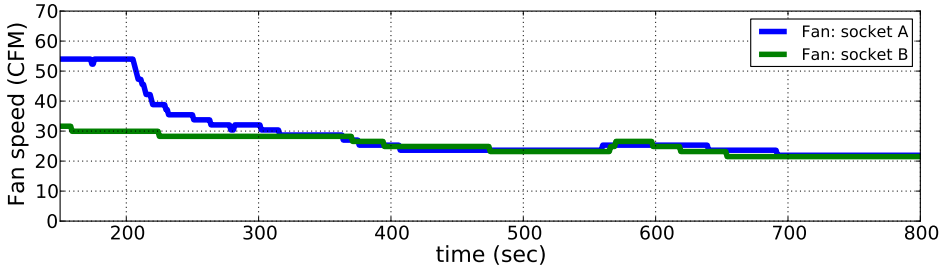
Figure 5.13: Fan speed ratio in a system with 8 DIMMs



(a) Workload, W13



(b) Workload, W11

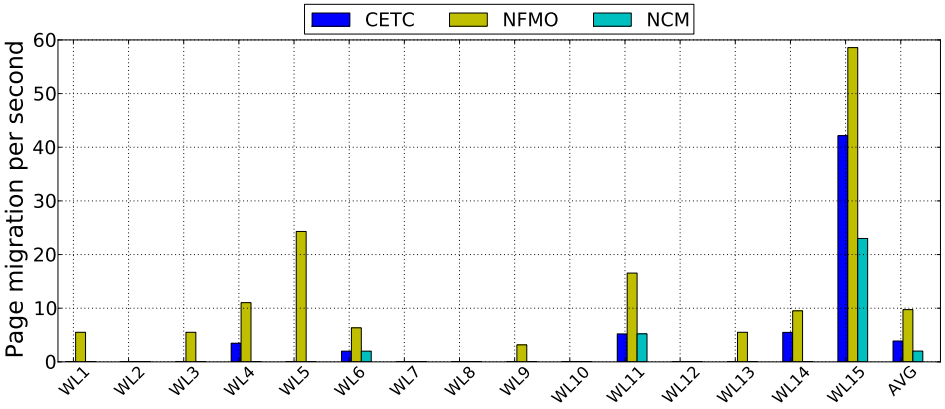


(c) Workload, W15

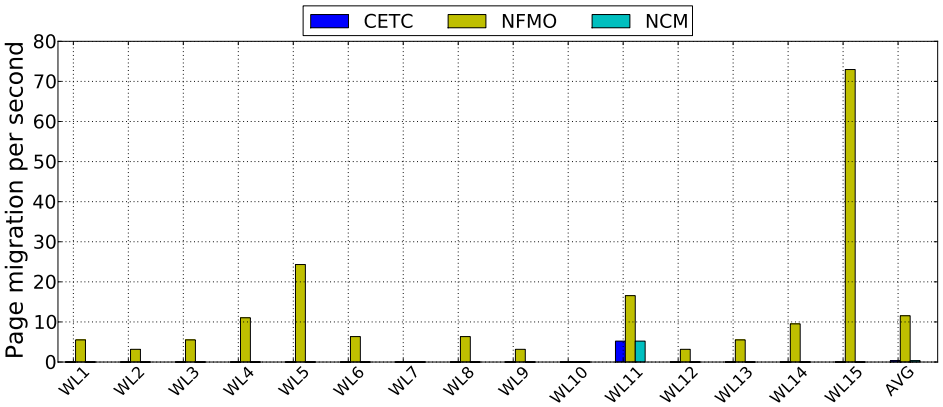
Figure 5.14: Fan speed response with CETC, 8DIMM and 45^{o}C

that CETC is able to converge and balance the fan speed in all cases.

Page migration We computed page migration per second to show the robustness of our solution. Figure 5.15(a) shows the rate of page migration in a system with 8 DIMMs and 45^{o}C ambient temperature. The results show that CETC average rate of page migrations below 5 pages per second. This overhead is negligible in practice since migrating a page takes a few microseconds which makes our solution highly attractive. Furthermore, having a few migrations is an indication that our solution is stable. The NFMO has the highest migration rate since it may



(a) Local ambient temperature 45^{o}C



(b) Local ambient temperature is 35^{o}C

Figure 5.15: Page migration in a system with 8 DIMMs

Table 5.4: Performance overhead (%)

Local ambient temperature	number of DIMMs	CETC	NFMO	NMM	NCM	DLB	DTM-CM+PI
45^{o}C	8	0.175	0.184	0.093	0.102	0.001	0.169
35^{o}C	8	0.115	0.120	0.069	0.100	0.000	0.120
45^{o}C	16	0.175	0.187	0.109	0.102	0.001	0.080
35^{o}C	16	0.115	0.122	0.033	0.100	0.000	0.023

activate more DIMMs as it does not consider the trade off between fan speed and number of active DIMMs. The case of W15 has the highest migration rate since the workload mix has 3 instances of swim, swim has a wide set of hot pages. The page migration rate for the system with 8 DIMMs and 35^{o}C ambient temperature is shown in Figure5.15(b). In this case, the migration rate for CETC drops below one page per second which causes almost no overhead.

Overhead Table 5.4 shows the average performance overhead for the entire set of workloads due to clustering, page and CPU migrations and throttling for 45^{o}C ambient. The overhead is lower in 35^{o}C ambient. In these results we account for the migration overhead of the pages between memory models as well as threads between cores and sockets. We also account for the overhead of throttling when the temperature exceeds the temperature threshold. The results show that the overhead is below 0.2% for CETC, which is negligible. This overhead mainly comes from workload scheduling. The overhead is small due to the big difference between migrations cost and temperature change. The overhead of CETC is lower than NFMO since the later cause more page migrations. The policy NCM has lower overhead then CETC since there are no CPU migrations and the maximum fan speed in this policy is higher which provides better cooling for memory. The overhead in DLB is lowest since it rely on the fan subsystem only to prevents the thermal problems which leads to high cooling energy costs.

5.4 Conclusion

Memory and cooling subsystems consume a significant amount of energy in high end servers. Prior research handled temperature and computing energy problems separately and also did not consider cooling, resulting in suboptimal solutions. In this thesis, we develop CETC, a unified solution that integrates the energy, thermal and cooling management to maximize energy savings. As part of CETC, we have proposed a unified thermal and cooling model for CPU and memory subsystems that is applicable for online management. Our solution is designed using formal MIMO control to ensure stability. CETC reduces the operational energy of the memory by clustering memory accesses to a subset of memory modules while considering thermal and cooling metrics. It also removes hot spots between and within the sockets and reduces the effects of thermal coupling to minimize cooling costs. CETC also controls fan speed to ensure stability in control. The reported experimental results show that our approach delivers an average cooling energy savings of 70% compared to the state of the art techniques with negligible performance overhead of less than 0.2%.

Chapter 5, in part, is a reprint of the material that is under preparation to be submitted to IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems. Ayoub, R.; Nath R. ;Indukuri, K. R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion and Future Work

Temperature and cooling challenges are consistently increasing due to technology scaling, rising in integration level and performance. High temperature affects reliability, performance, leakage energy, thermal packaging costs and cooling energy costs. Cooling energy costs contribute to a big portion of the energy costs in modern servers and data centers. As a result, proper consideration of these two important metrics is required to ensure quality and usability of the computing products.

In this thesis we demonstrate that managing the temperature, computational and cooling energy needs to be in performed in an integrated fashion to maximize efficiency due to the dependencies between these components. In general, the characteristics of the workloads are heterogeneous in nature and require intelligent resource management techniques to assign the workload to the resources in a thermally and energy aware manner. In conjunction with proper placement of the workload on the resources we utilize low power states in the system to put the unnecessary components in a low power mode to save energy. The management layers spans from microarchitectural level up to server level with multiple sockets and memory. The following sections summarize the contributions of this thesis and outline the future research directions.

6.1 Thesis summary

Microarchitectural level management (CPU): We demonstrate that the microarchitectural level provides unique opportunities for optimizations. In this work we target application specific processors where the tasks are known in advance. Our analysis show that the register file as a thermal hot spot and energy hungry component. Temperature and energy optimizations are obtained by eliminating the redundant writes to the register file at virtually no performance overhead. We use a combined software-hardware approach to capture these writes during run time at low cost. These redundant writes are identified at the compilation time and this information is encoded in the registers name space using post compiler register renaming algorithms. We developed a cost efficient hardware support to capture these redundant writes. The hardware that we propose is programmable, hence preserving the generality of computing. The experimental results show that this technique is able to achieve 22.3% energy savings in register file and 4^{o}C reduction in temperature on average.

Core level management (CPU): In this thesis we introduce a novel core level proactive thermal management technique that intelligently allocates the jobs across the cores in a chip multiprocessor environment. This layer of management can operates as a complementary to microarchitectural level to maximize savings. We adopt a proactive paradigm to be able to act before reaching thermal emergencies to prevent them when possible. In order to predict temperature accurately at low cost, we introduce a novel temperature predictor that is based on the band limited property of the temperature frequency spectrum. The temperature prediction depends on the previous temperature samples. The prediction coefficients can be identified accurately at the design stage which makes our predictor workload independent and of negligible overhead Our results show that applying our algorithm considerably reduces the average system temperature, hottest core temperature, and improves performance by 6 ^{o}C, 8 ^{o}C and 72% respectively.

Multi-tier CPU management: In general, at any given speed setting the fan can dissipate only a limited amount of heat from a CPU. Increasing the speed requires exponential increase in fan power. This indicates that the temperature solutions that act only within a socket are not sufficient to minimize cooling energy since some sockets may generate much more heat than others, which requires a better heat balance between them. We propose a new multi-tier algorithm that schedules the workload at the core as well as the socket levels to minimize cooling energy and the occurrence of thermal emergencies. To ensure stable solutions we developed a control theoretic framework. For the core level we use our proactive thermal management technique to reduce the hot spots across the cores and improve cooling savings within a given socket. Reported results show that our multi-tier scheme is able to increase the cooling energy savings to 80% on average with performance hit less than 1%.

Combined memory and CPU management: In this thesis we introduce a holistic energy, temperature and cooling management technique that significantly lowers the energy consumption and cooling costs of CPU sockets and memory. We develop a comprehensive thermal and cooling model which is used for online decisions. This technique reduces the operational energy of the memory by clustering pages to a subset of memory modules while accounting for thermal and cooling aspects. At the same time it tries to remove hot spots between and within the sockets, and reduces the effects of thermal coupling with memory to save cooling costs. We designed our technique using formal control method to ensure stability. The experimental results show that this approach delivers a total cooling and memory energy savings on average of 70% compared to the state of the art techniques at performance overhead of less than 1%.

6.2 Future research directions

6.2.1 Temperature and cooling management in data center

This thesis focuses on optimizing thermal management and cooling techniques in a single machine environment that result in appreciable savings. However, in a data center the temperature profile and energy consumptions across these machines is likely to vary despite using load balancing, an issue that makes our techniques behave as a local optimizations in data centers. One reason for thermal and energy variations between machines is related to the heterogeneity in the workloads since different tasks can exercise the resources differently. The other reason is related to temperature distribution in data center which can be far from uniform due to poor cooling distribution (e.g. air circulation).

A natural way to manage multi-machine environment is to add higher layers of management to control the workload allocation between the machines. In our recent work [13] we evaluate thermal and cooling aware workload management technique between machines that focuses on the CPU subsystem. Our workload manager exploits virtual machine technology with live migration to schedule the workload between the physical machines at low overhead. However, we use a simplistic thermal model for the data center and we do not account for energy and thermal optimizations for the memory subsystem. It is important to develop a robust thermal model at the data center level that is simple enough so it can be used for run time decisions. Our prior work does not include memory optimizations which is more challenging than the case of a single machine. This is because migrating the workload between machines affects power and temperature profiles of memory in a complex way. It is also important to investigate distributed control in a data center environment since centralized solutions may not scale adequately.

6.2.2 Liquid cooling

Technology scaling and increase in integration are escalating the power density and temperature in chips. Moreover, recent research demonstrates that 3D stacking is a promising integration approach to mitigate power and timing

constraints of the interconnects [16]. However, its drawbacks are the high power density and temperature. The key problems with air cooling is that removing the heat from the CPU and memory becomes very energy consuming when the temperature reaches very high levels.

Liquid cooling is emerging as alternative technology for removing the excess heat since it can lower the convective resistance to a much smaller values compared to air cooling at the same energy cost. The other advantage of liquid cooling is the ability to direct the flow of the liquid very precisely to the hot spots which is hard to do with air cooling. Techniques that are developed for air cooling in this thesis can be extended to study liquid cooling due to the similarities in their model. However, liquid cooling also has drawbacks in terms of reliability and cost which need to be improved.

Bibliography

[1] *www.sun.com/servers/x64/x4270/.*

[2] *Quad-Core Intel Xeon Processor 5300 Series: Thermal/Mechanical Design Guidelines.*

[3] *www.intel.com/products/server/motherboards/s5400sf/.*

[4] *http://www.digitalbattle.com/2007/11/12/intel-launches-45nm-cpus/.*

[5] *http://www.sunon.com.tw/products/pdf/DCFAN/PMD4056.pdf.*

[6] *www.micron.com/products/dram/.*

[7] Failure mechanisms and models for semiconductor devices, jedec publication jep122c. *http://www.jedec.org.*

[8] Reducing data center cost with an air economizer. *Intel*, 2008.

[9] A. Ajami, K. Banerjee, and M. Pedram. Modeling and analysis of nonuniform substrate temperature effects on global interconnects. *IEEE Trans. on CAD*, pages 849–861, 2005.

[10] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[11] J. Ayala, M. Lopez-Vallejo, A. Veidenbaum, and C. Lopez. Energy aware register file implementation through instruction precode. *ASAP*, pages 86 – 96, 2003.

[12] R. Ayoub, P. Petrov, and A. Orailoglu. Application specific instruction memory transformations for power efficient, fault resilient embedded processors. *SOCC*, pages 195 – 198, 2004.

[13] R. Ayoub, S. Sharifi, and T. S. Rosing. Gentlecool: Cooling aware proactive workload scheduling in multi-machine systems. *DATE*, pages 295–298, 2010.

[14] L. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[15] L. Barroso and U. Holzle. The datacenter as a computer an introduction to the design of warehouse-scale machines. 2009.

[16] B. Black, M. Annavaram, N. Brekelbaum, J. Devale, L. Jiang, G. H. Loh, D. Mccauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3d) microarchitecture, 2006.

[17] P. Bose. Power-efcient microarchitectural choices at the early design stage. *Workshop on Power-Aware Computer Systems*, 2003.

[18] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *ISCA*, pages 83–94, 2000.

[19] P. Chaparro, J. Gonzalez, G. Magklis, Q. Cai, and A. Gonzalez. Understanding the thermal implications of multicore architectures. *IEEE TPDS*, 18(8):1055–1065, 2007.

[20] H. Chiueh, L. Luh, J. Draper, and J. Choma. A novel fully integrated fan controller for advanced computer systems. *SSMSD*, pages 191–194, 2000.

[21] J. Choi, C. Cher, H. Franke, H. H. A., Weger, and P. Bose. Thermal-aware task scheduling at the system software level. *ISLPED*, pages 213–218, 2007.

[22] A. Coskun, T. Rosing, and K. Gross. Proactive temperature management in mpsocs. *ISLPED*, 2008.

[23] L. Cruz, A. Gonzalez, M. Valero, and P. Topham. Multiple-banked register file architecture. *ISCA*, pages 316 – 325, 2000.

[24] G. Dhiman, K. Mihic, and T. Rosing. A system for online power prediction in virtualized environments using gaussian mixture models. *DAC*, pages 807–812, 2010.

[25] X. Fan, C. Ellis, and A. Lebeck. Memory controller policies for dram power management. *ISLPED*, pages 129 – 134, 2001.

[26] G. Franklin, J. Powel, and M. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, 1990.

[27] D. R. Gonzales. Micro-risc architecture for the wireless market. *International Symposium on Microarchitecture*, pages 30 – 37, 1999.

[28] H. Hai, S. Kang, L. Charles, and K. Tom. Improving energy efficiency by making dram less randomly accessed. *ISLPED*, pages 393–398, 2005.

[29] S. Haihua, L. Frank, D. Anirudh, A. Emrah, and N. Sani. Full chip leakage estimation considering power supply and temperature variations. pages 78–83, 2003.

[30] A. Hashimoto and J. Stevens. Wire routing by optimization channel assignment within large apertures. *Design Automation Workshop*, pages 155–163, 1971.

[31] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and freon: temperature emulation and management for server systems. *ASPLOS*, pages 106–116, 2006.

[32] J. Hennessy and D. Patterson. *Computer Architecture*. Morgan Kaufmann, 2003.

[33] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. *ISLPED*, pages 217–222, 2003.

[34] N. Kim and T. Mudge. The microarchitecture of a low power register file. *ISLPED*, pages 384–389, 2003.

[35] A. Kumar, L. Shang, L. Peh, and N. Jha. Hybdtm: A coordinated hardware-software approach for dynamic thermal management. *DAC*, pages 548–553, 2006.

[36] C.-H. Lin, C.-L. Yang, and K.-J. King. Ppt: Joint performance/power/thermal management of dram memory for multi-core systems. *ISLPED*, pages 93–98, 2009.

[37] J. Lin, H. Zheng, Z. Zhu, H. David, and Z. Zhang. Thermal modeling and management of dram memory systems. *ISCA*, pages 312–322, 2007.

[38] R. Lyon and A. Bergles. Noise and cooling in electronics packages. *IEEE Trans on Components and Packaging Technologies*, 29(3):535–542, 2006.

[39] F. Marvasti. *Nonuniform Sampling Theory and Practice*. Kluwer Academic/Plenum Publishers, New York, 2001.

[40] G. Mohamed, M. Powell, and T. Vijaykumar. Heat-and-run: leveraging smt and cmp to manage power density through the operating system. *SIGOPS Oper. Syst. Rev.*, 38(5):260–270, 2004.

[41] D. Mugler. Computationally efficient linear prediction from past samples of a band-limited signal and its derivative. *IEEE Transactions on Information theory*, 36:589–596, 1990.

[42] D. Mugler. Computational aspects of an optimal linear prediction formula for band-limited signals. *Computational and applied mathematics*, pages 351–356, 1992.

[43] D. Mugler and Y. Wu. An integrator for time-dependent systems with oscillatory behavior. *Computer Methods in Applied Mechanics and Engineering*, 171:25–41, 1999.

[44] B. Nathan, D. Ronald, H. Lisa, L. Kevin, S. Ali, and R. Steven. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[45] A. Papoulis. *Signal analysis*. McBraw-Hill, London, 1981.

[46] I. Park, D. Powell, and N. Vijaykumar. Reducing register ports for higher speed and lower energy. *MICRO*, 2002.

[47] M. Patterson. The effect of data center temperature on energy efficiency. *Proc. ITHERM*, pages 1167–1174, 2008.

[48] M. Pedram and S. Nazarian. Thermal modeling, analysis, and management in vlsi circuits: principles and methods. pages 1487–1501, 2006.

[49] P. Petrov and A. Orailoglu. Performance and power effectiveness in embedded processors - customizable partitioned caches. *IEEE TCAD*, 20(11):1309–1318, 2001.

[50] P. Petrov and A. Orailoglu. Tag compression for low power in dynamically customizable embedded processors. *IEEE TCAD*, 23(7):1031–1047, 2004.

[51] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon. Low-power data forwarding for VLIW embedded architectures. *IEEE Transactions on* VLSI *Systems*, 10(5):614–622, 2002.

[52] R. Schmidt, E. Cruz, and K. Iyengar. Challenges of data center thermal management. *IBM Journal of Research and Development*, 49(4/5):709–723, 2005.

[53] S. Sharifi and T. Rosing. Accurate direct and indirect on-chip temperature sensing for efficient dynamic thermal management. *IEEE TCAD*, 29(10):1586–1599, 2010.

[54] D. Shin, J. Kim, J. Choi, S. Chung, and E. Chung. Energy-optimal dynamic thermal management for green computing. *ICCAD*, 2009.

[55] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. *ISCA*, pages 2–13, 2003.

[56] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. *ISCA*, pages 2–13, 2003.

[57] K. Skadron, M. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *TACO*, pages 94–125, 2004.

[58] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware computer systems: Opportunities and challenges. *IEEE Micro*, 23:52–61, 2003.

[59] Q. Tang, S. Gupta, and G. Varsamopoulos. Thermal-aware task scheduling for data centers through minimizing heat recirculation. *Proc. ICCC*, pages 129–138, 2007.

[60] D. Tarjan, S. Thoziyoor, and N. Jouppi. Cacti 4.0. *Technical report, HP Laboratories*, pages 1–15, 2006.

[61] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. *ISCA*, pages 51–62, 2008.

[62] Z. Wang, C. Bash, N. Tolia, M. Marwah, X. Zhu, and P. Ranganathan. Optimal fan speed control for thermal management of servers. *IPAC*, pages 1–10, 2009.

[63] I. Yeo, C. Liu, and E. Kim. Predictive dynamic thermal management for multicore systems. *DAC*, pages 734–739, 2008.

[64] Y. Zhang and A. Srivastava. Accurate temperature estimation using noisy thermal sensors. *DAC*, pages 472–477, 2009.