AN INTEGRATED POWER, AREA, AND TIMING MODELING FRAMEWORK FOR THE DESIGN OF MULTITHREADED AND MULTI/MANYCORE ARCHITECTURES

A Dissertation

Submitted to the Graduate School

of the University of Notre Dame

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Sheng Li

Jay B. Brockman, Director

Graduate Program in Electrical Engineering

Notre Dame, Indiana

April 2010

AN INTEGRATED POWER, AREA, AND TIMING MODELING FRAMEWORK FOR THE DESIGN OF MULTITHREADED AND MULTI/MANYCORE ARCHITECTURES

Abstract

by

Sheng Li

Multithreaded and multi/manycore processors have already become an important new research direction. These processors have demonstrated great performance and efficiency advantages. This dissertation presents McPAT, an integrated power, area, and timing modeling framework that supports comprehensive design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm and beyond. McPAT includes models for the components of a complete chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, and integrated memory controllers. McPAT models timing, area, and dynamic, short-circuit, and leakage power for each of the device types forecast in the ITRS roadmap including bulk CMOS, SOI, and double-gate transistors. McPAT has a flexible XML interface to facilitate use with many performance simulators. Combined with a performance simulator, McPAT enables architects to consistently quantify the cost of new ideas and assess tradeoffs of different architectures using new metrics like energy-delay-area^2 product (EDA^{2}P) and energy-delay-area product (EDAP).

This dissertation also examines several new architecture ideas. We study the scaling trends of a multithreaded chip multiprocessor across technology generations from 90nm to 22nm. We also explore the interconnect options of future manycore processors by varying the degree of clustering over generations of process technologies. Clustering will bring interesting tradeoffs between area and performance because the interconnects needed to group cores into clusters incur area overhead, but many applications can make good use of them due to synergies of cache sharing. Combining power, area, and timing results of McPAT with performance simulation of PARSEC benchmarks at the 22nm technology node for both common in-order and out-of-order manycore designs shows that when die cost is not taken into account clustering 8 cores together gives the best energy-delay product, whereas when cost is taken into account configuring clusters with 4 cores gives the best EDA^{2}P and EDAP.

This dissertation also proposes a Lightweight Chip Multi-Threaded (LCMT) architecture targeting parallel irregular and dynamic applications. The LCMT is implemented a by extending techniques previously used in supercomputing framework to mainstream general purpose processors. The LCMT architecture is implemented atop a mainstream architecture with minimum extra hardware and leverage existing legacy software environments. We evaluate the proposed LCMT architecture using McPAT and a performance simulator. Comparisons between the proposed LCMT architecture with a Niagara-like baseline architecture show that LCMT achieves up to 1.74X better performance per Watt when running irregular and dynamic benchmarks, when compared to the baseline architecture.

To mom (Xiuqin), dad(Liang), my wife(Ying), and my son(Richard)

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

I would like to deeply thank my advisor, Dr. Jay Brockman for the enormous help and guidance during my years in graduate school. His open-minded research strategy, his patience with graduate students, and his constant and easy availability made him a wonderful advisor. I thank him for keeping spending his energy and effort to shape me into a mature engineer and researcher.

I would like to express my great gratitude to Dr. Norm Jouppi of HP Labs for being extremely supportive of my dissertation work and for being on my dissertation committee. I consider myself so fortunate that I have had the opportunity to work with Norm, and I have been tremendously benefitted from his great vision, his ability to see right through problems to solutions, and his expertise in almost every area.

I also would like to thank Dr. Peter Kogge and Dr. Greg Snider for reading my dissertation and serving on my committee. I also want thank Dr. kogge for inspiring me to conduct research on lightweight chip multithreading during my dissertation work.

I am also very grateful to Dr. Jung Ho Ahn and Dr. Naveen Muralimanohar of HP Labs. Jung Ho has been very supportive and gave me many invaluable feedbacks on McPAT and helped me to carry out the architectural simulations for our Micro paper. I want thank Dr. Naveen Muralimanohar for his feedbacks on McPAT.

Thank you also goes to Victor Zyuban at IBM for answering my questions on detailed circuit implementation.

The Notre Dame architecture research group has consistently been an engaging, friendly, and helpful environment. Shyamkumar Thoziyoor has been my long-term collaborator and a very good friend. He has been very supportive throughout my research work. Shannon Kuntz has also be a wonderful collaborator and a good friend. I benefit a lot from her in crystalize the research ideas. I want to thank them for their support and friendship. I would also like to express my thanks to the other wonderful graduate students in the group, Kamal, Amit, Arun, Srini, Dan, Peter, and Ke, for all the great discussions.

I want to thank HP, Cray, and DARPA for funding this research work.

I want to express my gratitude to my mom and dad for their unconditional love and support throughout my education.

I would like to deeply thank my wife for being together with me through all the heights and lows. I am cannot even imagine to finish this work if she were not there.

CHAPTER 1

INTRODUCTION

The computer architecture community is now facing new challenges. Multi-threaded and multi/manycore processors have already become an important new research direction. Processors have become complex systems on-chip with more and more resources including cores, large caches, networks, and memory controllers. As technology keeps scaling, processors with large number of cores such as the Intel 48-core single-chip-cloud computer (SCC) and the 100-core Tilera Tile-Gx processor are already available in either prototype or product. The rapid growth of core count will continue, as the semiconductor industry continues to provide more transistors per chip at a pace with Moore's Law.

Given the large number of powerful on-chip resources, there are many design decisions which need to be made such as: what is the optimal core count, the NoC topology, the optimum number of memory controllers, and optimum cache size; what clock frequency achieves the best performance for a reasonable power budget; How many threads should be used for each core; how the new device and interconnect technology can help to shape future architectures; How can the overhead of communication and synchronization among threads and cores be reduced. All these design choices are hard and can only be made through experiments from quantitative tools. The tools also need to explore some of the design space automatically and effectively since the design space is huge.

Moreover, for designs with low cost, area is another key issue. For a given die size, it is important to know how much area should be allocated for cores, caches, and interconnects. Finally, target clock frequency needs to be satisfied. All these challenges can only be solved by a unified infrastructure that can model a complete manycore processor for power, area, and timing simultaneously. When evaluating these design choices, three important and inseparable design constraints must be studied consistently and simultaneously. They are: *power*, *area*, and *timing*.

Power (including both power dissipation itself and power density), and the resulting heat issues, have become possibly the most critical design constraint of modern and future processors. This concern only grows as the semiconductor industry continues to provide more transistors per chip in pace with Moore's Law. Industry has already shifted gears to deploy architectures with multiple cores [54, 83], multiple threads [45, 51], and large last-level caches [54, 83] so that processors can be clocked at a lower frequency and burn less power, while still getting better overall performance. Controlling power and temperature in future multi-core and many-core processors will require even more novel architectural approaches.

Area is also a key, often neglected, constraint to keep the cost of designs under control as die costs are proportional to the fourth power of the die area [79] in theory and proportional to the second power of the die area because of good yield in current process [35]. Moreover, at very small feature sizes, little margin exists between design rules and manufacturing process variations, leading to an average 5% decrease in expected die yield with each successive technology node for mature IC designs [87]. Therefore, designers need to use silicon area more wisely as the technology keeps scaling.

Finally, cycle time is a key element of the performance equation. The first goal of any processor designs is to deliver required performance, which imposes constraints on target cycle time or clock frequency

It is increasingly difficult for architects to consider any of these elements in isolation, as most changes impact all three, as well as overall performance. For example, target clock frequency must be satisfied for any optimizations on area and power of the target processor as these optimizations on area and power can affect timing significantly. Also, when solving the power density and cooling issue, both power and area must be considered in a consistent manner simultaneously.

However, our ability to propose, design, and evaluate new architectures is currently limited by the quality of our tools. Four key factors drive the need for new tools to address changes in architecture and technology. These include the need to accurately model multicore and manycore architectures, the need to accurately model all sources of power dissipation, the need to evaluate power, area, and timing simultaneously, and the need to accurately scale circuit models into deep-submicron technologies.

1.1 Contributions

The design innovations to fully deploy the resources in multi/manycore processors and the urgent need for new tools to evaluate the innovations in computer architecture research are the two major drivers of this work. In this dissertation, we have had three major contributions.

- First, we have developed the *first* architecture level integrated power, area, and timing modeling framework which addresses these challenges, called McPAT (**M**ulti**c**ore **P**ower, **A**rea, and **T**iming) [59]. By modeling all fundamental components of a chip multiprocessor and by modeling power, area, and timing simultaneously, we show that McPAT supports comprehensive design space exploration for multicore and manycore processors ranging from 90nm to 22nm and beyond.

- Second, We have introduced new and comprehensive metrics that include both performance and cost, the operational cost (energy) and the capital cost (area) to evaluate new architectures. Energy-delay-area^2 product (EDA^{2}P) and energy-delay-area product (EDAP) are examples of these metrics. While a chip vendor may favor EDA^{2}P as *area*^{2} provides an approximation to die cost in practice [35], a system vendor could prefer EDAP as other fixed system costs such as memory and I/O reduce the overall system cost dependence on chip multiprocessor cost.

- Third, by combining McPAT with performance simulators, we conduct quantitative experiments and provide insights in several important aspects of multithreaded and multi/manycore processors. Specifically, we have studied the scaling trends and clustering trade-offs of future manycore processors. We have also proposed lightweight chip multithreading (LCMT), an architecture with direct hardware support for fine-grained multithreading, and show its advantages in power and area efficiency for throughput computing.

1.2 Organization

The rest of this dissertation is organized as follows: First in Chapter 2 we give an overview of McPAT, including the methodology, the overview of its framework, and its capability. Next in Chapter 3, we describe the architecture level models in detail. In Chapter 4, we describe the detailed circuit level and technology level models. Then in Chapter 5, we present how we have validated McPAT against industrial commercial high performance processors. This is followed by Chapter 6, which describes our research on scaling trends and clustering trade-offs of future manycore processors. Chapter 6 also present the use of new metrics such as energy-delay-area^2 product (EDA^{2}P) and energy-delay-area product (EDAP). Chapter 7 presents research on the lightweight chip multithreaded (LCMT) architecture. Finally, Chapter 8 concludes with the major contributions of this dissertation and a discussion of future research directions.

CHAPTER 2

MCPAT: AN INTEGRATED POWER, AREA, AND TIMING MODELING FRAMEWORK FOR MULTI/MANYCORE ARCHITECTURES

It has always been true in the computer architecture community that tools both limit and drive research directions. Wattch [16], first presented in 2000, has been such a tool, enabling a tremendous surge in power-related architecture research. However, four key factors drive the need for new tools to address changes in architecture and technology. These include:

- the need to accurately model multicore and manycore architectures,
- the need to accurately model all sources of power dissipation,
- the need to evaluate power, area, and timing simultaneously,
- and the need to accurately scale circuit models into deep-submicron technologies.

We have developed a new power, area, and timing modeling framework called McPAT (**M**ulti**c**ore **P**ower, **A**rea, and **T**iming), which addresses these challenges. McPAT advances the state-of-the-art in processor modeling in several directions.

- McPAT is the *first integrated* power, area, and timing modeling framework, which models them all consistently under the same principles simultaneously. This methodologically solves a key issue in current tools: modeling power and/or area without considering timing constraints. It also enables architects to use new metrics combining performance with both power and area such as energy-delay-area product (EDAP), which are useful to quantify the cost of new architectural ideas.

- McPAT is the *first* architectural modeling framework that supports all important components needed to model modern multithreaded and multicore/-manycore processors. Contemporary multicores are complex systems of cores, caches, interconnects, memory controllers, multiple-domain clocking, etc. McPAT models the power of the important components of multicore processors, including all the components listed above. McPAT supports detailed and realistic models that are based on existing in-order and OOO (out-of-order) processors. McPAT can model both a reservation-station-model and a physical-register-file model based on real architectures, including the Intel P6 [41] and Netburst [37].

- McPAT is the *first* architectural modeling framework that optimizes individual components automatically: it optimizes unspecified low-level design parameters while enabling the user to focus on high-level architectural investigation. This approach enables the user, if they choose, to ignore many of the low-level details of the components being modeled.

- Rather than being hardwired to certain simulators, McPAT is the *first* architectural modeling framework that uses an XML-based interface to enable easy integration with various performance simulators. McPAT currently works with M5 [13] and our own proprietary simulator.

- McPAT is the *first* framework that models all three types of power dissipation—

dynamic, static, and short-circuit power—to give a complete view of the power envelope of multicore processors. This is critical in deep-submicron technologies since static power has become comparable to dynamic power [48, 88].

- McPAT handles recent technologies that can no longer be accurately modeled by the linear scaling assumptions used by previous tools. McPAT utilizes technology projections from ITRS [88]; as a result, it will naturally evolve with industry road maps.

- McPAT is the *first* architectural modeling framework that supports advanced power management techniques, such as the P- and C-state [69] power management of modern processors, and can interact with a performance simulator to study various power management alternatives.

2.1 Related Work

CACTI [101] was the first tool to address the need for rapid power, area, and timing estimates for computer architecture research, focusing on RAM-based structures. The most recent release of the tool supports SRAM and DRAM based caches as well as plain memory arrays. It uses device models based on the industry-standard ITRS roadmap [88], using MASTAR [88] to calculate device parameters at different technology nodes. CACTI uses the method of logical effort to size transistors. It contains optimization features that enable the tool to find a configuration with minimal power consumption, given constraints on area and timing.

The complexity-effective approach [76] was one of the first attempts to use analytic models to obtain rapid estimates for processor timing, focusing on control, issue, selection, and bypass logic. Using generic circuit models for pipeline stages, it estimates the RC delay for each stage and determines the critical path.

Wattch [16] is a widely-used processor power estimation tool. Wattch calculates dynamic power dissipation from switching events obtained from an architectural simulation and capacitance models of components of the microarchitecture. For array structures, Wattch uses capacitance models from CACTI, and for the pipeline it uses models from [76]. When modeling out-of-order processors, Wattch uses the synthetic RUU model that is tightly coupled to the SimpleScalar simulator [8]. Wattch has enabled the computer architecture research community to explore power-efficient design options, as technology has progressed; however, limitations of Wattch have become apparent. First, Wattch models power without considering timing and area. Second, Wattch only models dynamic power consumption; the HotLeakage package [112] partially addressed this deficiency by adding models for subthreshold leakage. Third, Wattch uses simple linear scaling models based on 0.8μm technology that are inaccurate to make predictions for current and future deep-submicron technology nodes.

Orion [46] is a tool for modeling power in networks-on-chip (NoC). Version 2.0 includes models for area, dynamic power, and gate leakage, but does not consider short-circuit power or timing. It uses repeated wire models for interconnect, as well as device parameters for future technology nodes obtained from the ITRS roadmap using MASTAR and other methods. Kumar et al. provide further details on NoC layouts that take chip floorplans into consideration [55].

2.2 Overview and Operation

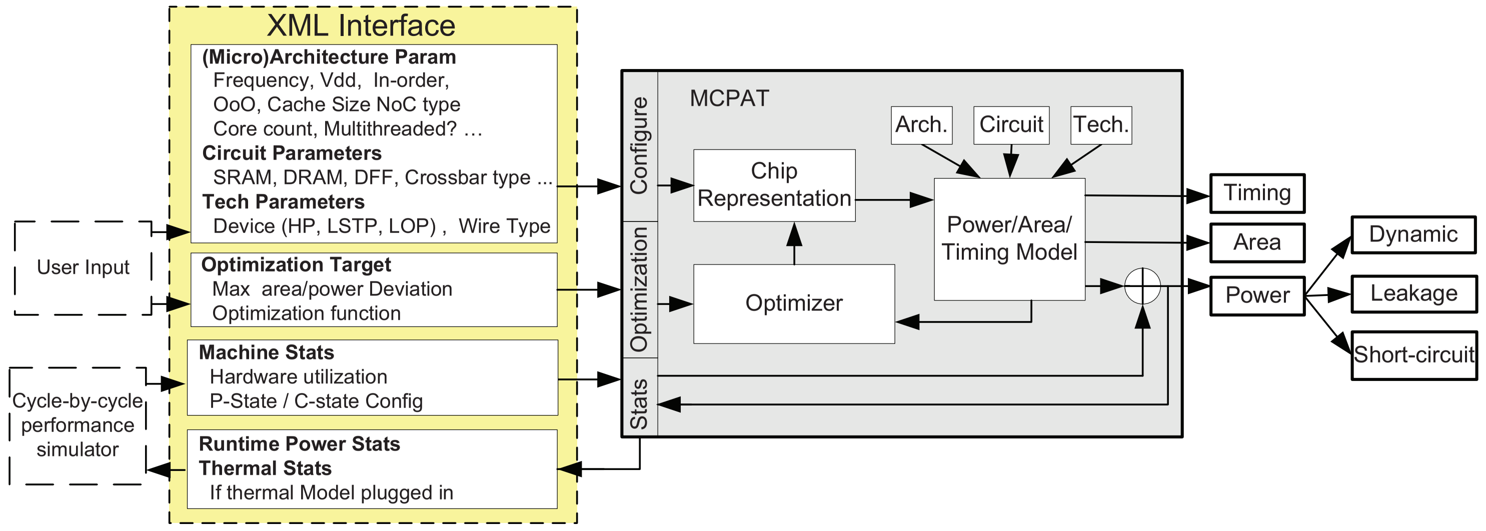


Figure 2.1. Block diagram of the McPAT framework.

McPAT is the first *integrated* power, area, and timing modeling framework for multithreaded and multicore/manycore processors. It is designed to work with a variety of processor performance simulators (and thermal simulators, etc.) over a large range of technology generations. McPAT allows a user to specify low-level configuration details. It also provides default values when the user decides to specify only high-level architectural parameters.

Figure 2.1 is a block diagram of the McPAT framework. Rather than being hardwired to a particular simulator, McPAT uses an XML-based interface with the performance simulator. McPAT uses an XML parser[42] developed by Berghen et.al to parse the large XML interface file. This interface allows both the specification of the static microarchitecture configuration parameters and the passing of dynamic activity statistics generated by the performance simulator. McPAT can also send runtime power dissipation results back to the performance simulator through the XML-based interface, so that the performance simulator can react to power or even temperature data. This approach makes McPAT very flexible and easily ported to other performance simulators. Since McPAT provides complete hierarchical models from the architecture to technology level, the XML interface also contains circuit implementation style and technology parameters that are specific to a particular target processor. Examples are array types, crossbar types, and CMOS technology generations with associated voltage and device types.

The key components of McPAT are (1) the hierarchical power, area, and timing models, (2) the optimizer for determining circuit level implementations, and (3) the internal chip representation that drives the analysis of power, area, and timing. Most of the parameters in the internal chip representation, such as cache capacity and core issue width, are directly set by the input parameters.

McPAT's hierarchical structure enables it to model structures at a low level including underlying device technology, and yet still allows an architect to focus on a high-level architectural configuration. The optimizer determines missing parameters in the internal chip representation. McPAT's optimizer focuses on two major regular structures: interconnects and arrays. For example, the user can specify the frequency and bisection bandwidth of on-chip interconnects or the capacity, associativity, the number of cache banks, while letting the tool determine the implementation details such as the choice of metal planes, the effective signal wiring pitch for the interconnect, or the length of wordlines and bitlines of the cache banks. These optimizations lessen the burden on the architect to figure out every detail, and significantly lowers the learning curve to use the tool. Finally, users always have the flexibility to turn off these features and set the circuit-level implementation parameters by themselves.

The optimizer generates the final chip representation, which is used to compute the area, timing, and peak power results. The peak power of individual units and the machine utilization statistics (activity factor) are used to calculate the runtime power dissipation results.

The detailed work flow of McPAT has two phases: the initialization phase and the computation phase. Specifically, in order to start the initialization phase a user first specifies static configurations, including parameters at all three levels, namely, architectural, circuit, and technology level. Architectural level parameters are similar to the parameters used in the configuration files of performance simulators, including the number of cores, the number of routers, shared last-level cache parameters, core issue width, OOO renaming schemes, OOO scheduling schemes, the number of hardware threads, and so on. Since McPAT needs to be paired up with a performance simulator for computing runtime power, a user can write a simple script to extract these architectural parameters from the performance simulator configuration files and generate the XML interface file for McPAT. Circuit-level parameters specify implementation details. For example, one can specify a certain array to use flip-flop based cells rather than SRAM based cells or to use a double-pumped crossbar for on-chip routers. Technology-level parameters include device type (high performance, low standby power, low operating power [88]) and interconnect. The static inputs also include the optimization options, such as the maximum area deviation, the maximum power deviation, and the optimization function. Once all the static configurations are set, the initialization of McPAT can be called by the performance simulator at the beginning of simulation.

During the initialization phase, McPAT will generate the internal chip representation using the configurations set by the user. The main focus of McPAT is accurate power and area modeling, and the target clock rate is actually used as a design constraint. Local greedy optimizations are used, except when there are obvious needs for considering the interplay of multiple components, which means McPAT finds the best solution for each component and assumes they will provide the best global configuration when individual components are put together. McPAT performs an intelligent and extensive search of the design space. The optimization space that McPAT explores can be huge, especially when there are many unspecified parameters. McPAT optimizes each component by varying circuit implementations. For example, McPAT varies the sub-array implementations of SRAM-based array components and metal layer/wiring pitch of interconnects. For each architectural component, McPAT first finds valid configurations meeting timing constraints. During this step, McPAT tries different configurations to increase the achievable target clock frequency, which leads to scarifying power and area. Then, if the resulting power and/or area are not within the allowed maximum deviation of the best value found so far, the configuration is discarded. Finally, McPAT applies a user-specified optimization function to find the one with the highest score among the configurations satisfying the power and area deviation. Users can easily add new objectives such as leakage power and assign different weights for each. During this initialization phase, the internal chip representation that meets the target clock rate is formed; the area of the target chip is reported; and dynamic energy per access and leakage power of each individual component are obtained. Finding valid configurations that meet the timing requirements is the most time-consuming step, since it will repeat many times until valid configurations are found or the possible configurations are exhausted. In order to reduce the initialization time, McPAT also provides a fast mode that can give a configuration with balanced power, area, and timing but does not guarantee to meet timing constraints. The fast mode enables expedited early-stage design space exploration. The execution time of McPAT in both normal and fast mode is shown in Table 2.1

Although local greedy optimizations are used for most components, there are exceptions when components need to be considered together. Currently, McPAT considers three global optimizations: 1) Bypass logic is assumed to be routed over functional units, register files, and reservation stations. 2) Global interconnects (links between routers) are assumed to be routed over last-level caches, if they are present, in horizontal, vertical, or even both directions. 3) Clock distribution networks are assumed to cover the whole chip for a global H-tree and whole domains such as cores for semi-global H-trees. Local clock distribution networks

TABLE 2.1

FEATURE AND EXECUTION TIME OF MCPAT IN NORMAL AND FAST MODE

Mode	Feature and Execution Time
`Normal`	Configuration meets timing constraints, or possible configurations are exhausted; in-order processor modeling time: $\sim$ 10 minutes; OOO processor modeling time: $\sim$ 20 minutes
`Fast`	Configuration with balanced timing, area, and power, but no guarantee to meet timing constraints; in-order processor modeling time: <5 minutes; OOO processor modeling time: 5 $\sim$ 10 minutes

are assumed to cover each entire domain. To accurately model the interplay of the scenarios, global optimization should be applied. For example, if the height of the interconnects is no larger than that of the L2 cache, McPAT assumes the interconnect can be routed over the last-level cache horizontally.

The computation phase of McPAT is called by the performance simulator during simulation to generate runtime power numbers. Before calling McPAT to compute runtime power numbers, the performance simulator needs to pass the statistics, namely, the activity factors of each individual components to McPAT via the XML interface file. As shown in Equation 2.1, the activity factor of a component is the product of access count of the component and the average Hamming distance of all accesses for the time interval. By changing the time interval, one can change the granularity of runtime power computation. If the performance simulator calls McPAT for runtime power computation every cycle, a cycle accurate power consumption profile will be generated, which will be useful to study realtime power spikes. If the performance simulator calls McPAT for runtime power computation after power simulation is complete, an averaged power profile will be generated.

$$ActivityFactor = (AccessCount * ((\sum_{i=1}^{n} HammingDistance)/n))/n \qquad (2.1)$$

In Equation 2.1, n is the cycle count for a given simulation period, $AccessCount$ is the number of accesses to a specific component during the period, and the $HammingDistance$ is the total number of flipped bits for two consecutive accesses. If the performance simulator cannot track the Hamming Distance, McPAT assumes that all bits are flipped per cycle. It is always best that the performance simulator can provide the activity factor for each individual component, however, McPAT also has the ability to reason about activity factors for components as long as the basic statistics information is provided. For example, if the performance simulator can track only the number of memory instructions rather than the detailed information of activity factors of the load or store queue, McPAT assumes that each memory instruction will involve one read, one write, and one or two search operations (depends on the hardware specification) on load and store queues. This assumption is made based on the default implementations of the load and store unit as discussed later in this dissertation.

McPAT runs separately from a performance simulator and only reads performance statistics from it — therefore, its impact on the simulation speed of the native performance simulator is minimal. Although the initialization phase of McPAT may take some time to complete because of the huge search space, it will not affect the simulator speed significantly since it needs to be done only once at the beginning of the simulation. During the computation phase, some simulator overhead may result from added performance counters.

2.3 Integrated and Hierarchical Modeling Framework

In order to model the power, area, and timing of a multicore processor, McPAT takes an *integrated* and *hierarchical* approach. It is integrated in that McPAT models power, area, and timing *simultaneously*. Because of this, McPAT is able to ensure that the results are mutually consistent from an electrical standpoint. It is hierarchical in that it decomposes the models into three levels: architectural, circuit, and technology level. This provides users with the flexibility to model a broad range of possible multicore configurations across multiple implementation technologies. Taken together, this integrated and hierarchical approach enables the user to paint a comprehensive picture of a design space, exploring tradeoffs between design and technology choices in terms of power, area, and timing.

2.3.1 Power Modeling

As shown in Equation (2.2), power dissipation of CMOS circuits has three main components: dynamic, short-circuit, and leakage power. All three contribute significantly to the total power dissipation of multicore processors fabricated using a deep-submicron technology.

$$P_{total} = \underbrace{\alpha C V_{dd} \Delta V f_{clk}}_{Dynamic} + \underbrace{V_{dd} I_{short\_circuit}}_{Short\_circuit} + \underbrace{V_{dd} I_{leakage}}_{Leakage} \tag{2.2}$$

The first term is the *dynamic power* that is spent in charging and discharging the capacitive loads when the circuit switches state, where α is the activity factor, C is the total load capacitance, V_{dd} is the supply voltage, ΔV is the voltage swing during switching, and f_{clk} is the clock frequency. C depends on the circuit design and layout of each IC component; we calculate it using analytic models for regular structures such as memory arrays and wires, along with empirical models for random logic structures such as ALUs. The activity factor α indicates the fraction of total circuit capacitance being charged during a clock cycle. We calculate α using access statistics from architectural simulation together with circuit properties.

The second term is the *short-circuit power* that is consumed when both the pull-up and pull-down devices in a CMOS circuit are partially on for a small, but finite amount of time. Short-circuit power is about 10% of the total dynamic power; however, it has been reported that the short-circuit power can be approximately 25% of the dynamic power in some cases [74, 110]. When circuit blocks switch, they consume both dynamic and short-circuit power. Inherent circuit properties determine the ratio of the short-circuit power to the dynamic power, which is a strong function of the V_{dd} to V_{th} ratio. Since the V_{dd} to V_{th} ratio shrinks for future low power designs, short-circuit power is expected to become more significant in future designs that require lower power and longer battery life. We compute the short-circuit energy per switch of a gate, using the Equations 2.3 to 2.5 derived in [74].

$$E_S = \frac{1}{\frac{1}{E_S(t_T<<\tau)} + \frac{1}{E_S(t_T>>\tau)}} \tag{2.3}$$

$$E_S(t_T << \tau) = \frac{3}{10} * \frac{(0.5 - \upsilon_T)^3}{\alpha^2 2^{3\upsilon_T}} C_{in} V_{dd}^2 \frac{f_o^2}{F_O \beta_r} \tag{2.4}$$

$$E_S(t_T >> \tau) = \frac{(0.5 - \upsilon_T)^{3/2}}{10 * 2^{3\upsilon_T + 2\alpha}} C_{in} V_{dd}^2 f_o \tag{2.5}$$

Here, $E_S(t_T << \tau)$ is the short circuit energy per switch of a gate with slow input, while $E_S(t_T << \tau)$ is the short circuit energy per switch of a gate with fast input, υ_T is the normalized threshold voltage (V_{th}/V_{dd}), α is the velocity saturation index which is close to one at deep sub-micron technology, C_{in} is the input capacitance of the gate, V_{dd} is the supply voltage, f_o is the transistor driveability ratio of succeeding gates, and F_O is fanout (C_{in}/C_{out}).

The third term is the *static* power consumed because of *leakage* current through the transistors, which in reality function as "imperfect" switches. There are two distinct leakage mechanisms, and the magnitude of each leakage current is proportional to the width of the transistor and depends on the logical state of the device. The first type of leakage, *subthreshold leakage*, occurs when a transistor that is supposedly in the off state actually allows a small current to pass between its source and drain. We let I_{sub} denote the subthreshold through a unit-sized (off) transistor. The second type, *gate leakage*, is the current that leaks through the gate terminal. We determine the unit leakage current using MASTAR [88]. In McPAT, the unit $I_{g_{on}}$ includes both the gate leakage current flows through the channel to the gate and the gate leakage current flows through the source/drain and gate overlap.

In order to model the leakage current for a circuit block with many transistors, we need to consider which logical state each transistor is in, then sum up the leakage current components for each. If a circuit block is in some logical state s, we can express the *effective width* of all (off) transistors exhibiting subthreshold leakage in that state as $W_{sub}(s)$; similarly, we can express the effective widths for gate leakage for on and off transistors as $W_{g_{on}}(s)$ and $W_{g_{off}}(s)$. If $\Pr(s)$ is the probability that a circuit is in state s, we can express the total leakage current for a given block as the average leakage current over all possible states S, as shown in Equation (2.6):

$$I_{leakage} = \sum_{s=1}^{S} \Pr(s)[W_{sub}(s)I_{sub} + W_{g_{on}}(s)I_{g_{on}} + W_{g_{off}}(s)I_{g_{off}}] \qquad (2.6)$$

In McPAT, we calculate the leakage current for different states of each individual basic circuit block. McPAT assumes that a circuit has equal possibilities to be in any states by default and computes total leakage. This probability distribution can also be overridden by the performance simulator when it has the ability to track the states of individual circuit blocks. Subthreshold leakage and gate leakage have a different leakage path even for the same circuit block in the same state. Gate leakage current when the device is on ($I_{g_{on}}$) differs greatly from the gate leakage current when the device is off ($I_{g_{off}}$). Since the off state gate leakage current is at least an order of magnitude less than the on state gate leakage current, McPAT ignores the off state gate leakage in the models. Figure 2.2 shows the subthreshold leakage and gate leakage paths in an SRAM cell at idle state with "0" being stored. Figure 2.3 shows subthreshold leakage and gate leakage paths

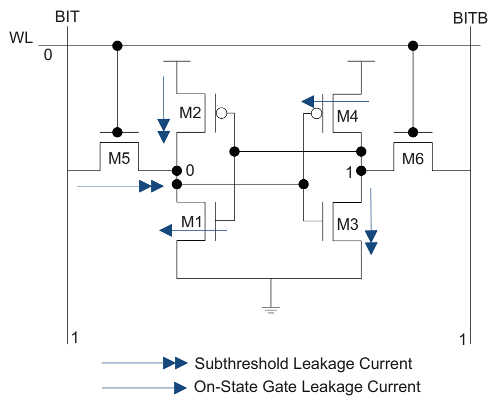


Figure 2.2. Leakage current paths in SRAM cell at idle state. Bitlines are precharged to VDD.

in a NAND2 gate for all its possible states. When inputs are "11", subthreshold leakage reaches a peak since there are two transistors leaking in parallel. When inputs are "00", subthreshold leakage reaches a minimum because two leaking devices are stacked. Research in [67] shows that the stacking effects can reduce the subthreshold leakage significantly because of negative V_{gs}, a lowered signal rail V_{ds}, reduced drain induced barrier lowering (DIBL), and body effect. The stacking factor of a static CMOS gate with fan-in being equal to two as shown in Figure 2.3 can be obtained using Equation 2.7, where $\alpha = \frac{\lambda_d}{1+2*\lambda_d}$, λ_d is DIBL, S is subthreshold swing, and V_{dd} is the supply voltage. In the circuit level models, McPAT assumes the maximum fan-in and fan-out does not exceed four to achieve good performance. The stacking factors of CMOS gates with fan-in of three or four are calculated using methods in [5].

$$StackingFactor = 10^{\frac{\lambda_d * V_{dd}}{s}*(\alpha-1)} \tag{2.7}$$

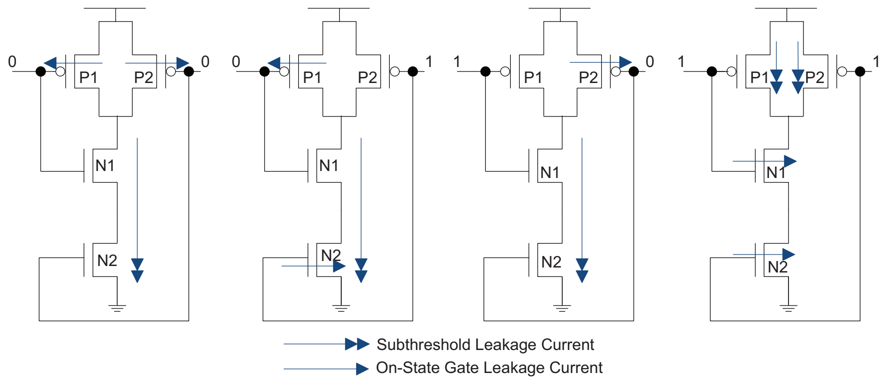


Figure 2.3. Leakage current paths in a NAND2 gate for all possible states.

2.3.2 Timing Modeling

Like the power model, McPAT's timing model breaks the system down into components and stages. While the power model requires only the capacitance to compute dynamic power, the timing model uses both resistance and capacitance to compute RC delays, using a methodology similar to CACTI [101] and the work by Palacharla et al. [76], with significant changes in implementation described later. McPAT determines the achievable clock frequency of a processor from the timing results of its components along the critical path. Every component has timing design constraints with two criteria: throughput and latency. Throughput determines the cycle time of the component, while latency determines the access time. For components that can be pipelined, including caches and global interconnects, their achievable throughput sets the upper limit of the achievable clock rate of the processor. For the components that cannot be pipelined such as the wakeup logic and result broadcast buses, their cycle time and access time are actually the same. Therefore, their latency and throughput set the upper limit of the achievable clock rate. When the latency or throughput of components along the critical path cannot satisfy the target clock rate, McPAT will output warning messages and the user will need either change the configuration parameters or lower the target clock frequency.

2.3.3 Area Modeling

McPAT uses the analytical methodology described in CACTI to model area of basic logic gates and regular structures, including memory arrays (e.g., RAM, CAM (content addressable memory), and DFF (D flip-flop)), interconnects (e.g., router, link, and bus), and regular logic (e.g. decoder and dependency-checking unit). An algorithmic approach does not work well for complex structures that have custom layouts, such as ALUs. For these, currently McPAT takes an empirical modeling approach [33, 82] which uses curve fitting to build a parameterizable numerical model for area from published information on existing processor designs, and then scales the area for different target technologies. McPAT computes the final die area of the target design by adding up the area of individual components. When adding area up, McPAT considers 10% placement and routing overhead. As mentioned in Section 2.2, there are three major places where interactions between components are considered. For these places, McPAT assumes the upper layer interconnect can be routed over lower layer components with minimum overhead.

2.3.4 Hierarchical Modeling Framework

McPAT's integrated power, area, and timing models are organized in a three-level hierarchy, as illustrated in Figure 2.4. McPAT's modeling framework provides comprehensive models for multicore/manycore processors from the architecture to the technology level. On the architectural level, a multicore processor is

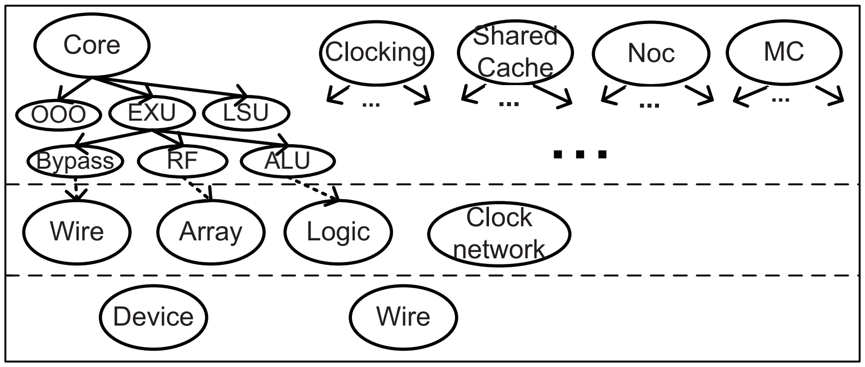


Figure 2.4. Modeling methodology of McPAT.

decomposed into major architectural components such as cores, NoCs (network-on-chips), caches, memory controllers, and clocking. On the circuit level, the architectural building blocks are mapped into four basic circuit structures: hierarchical wires, arrays, complex logic, and clocking networks. On the technology level, data from the ITRS roadmap [88] is used to calculate the physical parameters of devices and wires, such as unit resistance, capacitance, and current densities. As part of McPAT, we developed an enhanced CACTI that contains new CAM and fully-associative cache models, new technology models for 22nm double gate (DG) devices, new global interconnect models, new clocking models, new pipeline models, and new short-circuit and leakage models. McPAT is tightly coupled with the enhanced CACTI and calls CACTI to obtain partial results at different modeling levels.

2.4 Modeling Power-saving Techniques

McPAT models two major power saving techniques: clock gating to reduce dynamic power and power-saving states to reduce static power, which results in another distinguishing feature of McPAT—its ability to model advanced power management techniques, such as the P- and C-state [69] power management of modern processors. This allows the simulator to react to simulated power or thermal sensors (assuming a temperature model attached to the backend) by changing voltage and frequency settings, or by invoking one of multiple power-saving states on idle circuit blocks. Architects can use the framework to model power management alternatives.

2.4.1 P-state Modeling

A P-state is an operational state defined by a combination of clock frequency and supply voltage. A core or processor can perform useful work in any P-state. By reducing its clock frequency and/or supply voltage, the core or processor can achieve a much lower power profile while still being active although the processing speed will be lower. The modeling of P-states is based on the McPAT's ability to model clock gating and its flexible XML-interface. Since McPAT models dynamic energy per access per port for most components, it inherently models clock gating. Moreover, McPAT models the actual clock gating buffer circuitry at the distribution network heads. Combining these two features, McPAT can model clock gating schemes accurately.

After calling McPAT to finish initialization, a performance simulator can pass statistical information and invoke McPAT anytime during the simulation, and McPAT will calculate the corresponding power dissipation for the particular period and send it back to the performance simulator when required. When a performance simulator calls McPAT to compute the runtime power for a given period, the performance simulator can also change Vdd and clock frequency settings. In this way, the performance simulator can apply P-state management techniques.

2.4.2 C-state Modeling

A C-state is an idle state that is characterized by the amount of power consumed during the state and the latency and power consumption to enter and exit the state. As in modern processors [69], C0 is the state in which power-saving techniques are applied, where C1 to CN are different C-states with different static power saving levels. In C-states (except C0), the amount of leakage power can be reduced greatly. However, circuits in some deep power-saving modes cannot retain circuit states. For example, the contents of a cache will be lost if the cache is in a deep power-saving mode. McPAT's ability to model C-states is supported by its ability to model power-saving states with multiple sleep modes as described in [3]. The user can specify power-saving modes for various components. A performance simulator can apply C-state management techniques to instruct the core or processor to enter a particular C-state during the simulation.

2.4.2.1 Power-saving State Modeling

In order to model C-states, McPAT models a circuit used to support power-saving states inside each component and the wake-up overhead in terms of both timing and power. The circuit to support power-saving states is called a *sleep* transistor. The term sleep transistor describes either a PMOS or NMOS high Vth transistor that connects an external power supply to an internal circuit power supply which is commonly called a virtual power supply. The sleep transistor is controlled by a power management unit to shift the voltage level of the virtual power supply so that the circuit can be put into different power-saving states

S1S0	VG	Mode
00	0	Snore
01	V1	Dream
10	V2	Sleep
11	Vdd	Active

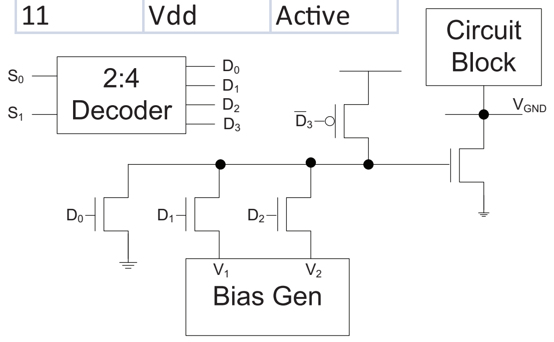


Figure 2.5. Circuit for multi-mode power-saving state control based on [3].

when idle. A PMOS sleep transistor is used to switch the Vdd supply and hence is named as a header switch. An NMOS sleep transistor controls the Vss supply and hence is called a footer switch. A PMOS-based header switch can control a virtual Vdd supply, while an NMOS based footer switch can control a virtual ground. Circuit designers use footers, headers, or both to achieve optimal tradeoffs between performance and overhead. PMOS transistors are less leaky than NMOS transistors of a same size. However, the advantage of a footer switch is its high drive and hence smaller area.

McPAT models the circuit used to support multiple power-saving states according to [3], where NMOS-based footer switches are used. In our modeling framework, there are four operating modes: active, sleep, dream, and snore. As shown in Figure 2.5, the circuitry uses footer devices and adjusts their gate biases to achieve different states. Having different sleep modes allows tradeoffs between the power saving and the wake-up overhead with respect to wakeup power and

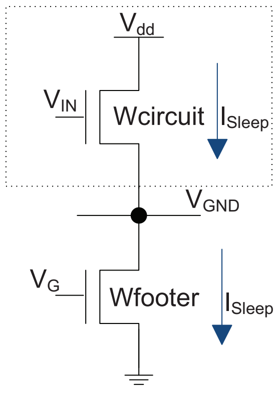


Figure 2.6. Footer sleep transistor model based on [3].

delay. For example, dream mode can save 50% more static power than sleep mode, but at the expense of twice the wake delay and three times the wake-up energy. As shown in Figure 2.5, the active mode is the state when the footer transistor is fully turned on. In this case, because of the large width of the footer transistor, the virtual ground V_{GND} has no significant difference from the real ground. The snore state is when the footer transistor is completely off with $V_G = 0$.

Figure 2.6 shows the model of a footer device associated with a circuit block. In order to match the effective width of the circuit block, the width of the footer transistor usually must be large. Therefore, the footer transistor dominates the wake-up penalty in terms of timing and power. The wake-up time and power when exiting a power-saving state are modeled using Equation 2.8 and Equation 2.9, where T_{wakeup} is the wake-up time, E_{wakeup} is the wake-up energy, V_{GND} is the virtual ground voltage level, C_{footer} is the capacitance of the footer transistor, $C_{circuit}$ is the total capacitance of the circuit on the charge/discharge path when entering/exiting power-saving states, and $I_{ON,F}$ is the saturation current of the footer device when it is on. The same delay and power consumption also occur when the circuit block enters a power-saving state. T_{wakeup} cannot be in parallel with any other operations since the circuit has to wait until its virtual ground has been completely discharged.

$$T_{wakeup} = \frac{(C_{circuit}+C_{footer})*V_{GND}}{I_{ON,F}} \tag{2.8}$$

$$E_{wakeup} = \frac{1}{2}(C_{circuit} + C_{footer})V_{GND}^2 \tag{2.9}$$

V_{GND} is calculated using Equation 2.10, where Vth_C and Vth_F represent the threshold voltages of the logic circuit and the footer device respectively, λ_d is DIBL, S is subthreshold swing, V_G is the gate voltage of the footer device, and V_{DD} is the supply voltage. Equation 2.10 shows that the steady state of V_{GND} linearly depends on footer gate voltage V_G, with a negative slope. The upper limit of V_{GND} is obtained by setting $V_G = V_{DD}$ in Equation 2.10, which is about 80% of V_{DD}.

$$V_{GND} = \frac{-V_G+S*log_{10}(\frac{W_{circuit}}{W_{footer}})+Vth_F-Vth_C+\lambda_d*V_{DD}}{2*\lambda_d} \tag{2.10}$$

$$\frac{I_{sleep}}{I_{active}} = 10^{-\frac{\lambda_d*V_{GND}}{S}} \tag{2.11}$$

Among all the power-saving states, only the *sleep* mode is state retentive. The work in [3] chooses sleep and dream states to be the points that are evenly spaced in the wake-up penalty vs. leakage current graph. However, this choice cannot guarantee that the sleep state can retain the state of the circuit blocks since the circuit block's rail-to-rail voltage has to be larger than a certain value to preserve the data in standby. Therefore, we define the sleep state to be when

$V_{GND} = 10\% * V_{DD}$, which causes the circuit block's rail-to-rail voltage to drop by 10%. Then, the *dream* mode is defined as a middle spot between *sleep* and *snore*. With all the V_{GND} calculated, the bias V1 and V2 can be obtained using Equation 2.10.

When the circuit block is in any of the power-saving states (sleep, dream, and snore), the footer transistor is turned off ($V_G < Vth_F$), operating in a weak inversion region. The leakage currents that follow through the circuit and the footer transistor are the same as shown in Figure 2.6. According to [3], the sleep leakage current to active leakage current can be expressed by Equation 2.11, where I_{sleep} is the leakage current in power-saving states, and I_{active} is the *leakage current* in active states. By changing V_{GND}, we can change the leakage current, and thus change power-saving states. When a circuit block is in a power-saving state, its static power can be computed using Equations 2.12 through 2.14, where P_{static} is the total static power, $P_{Cstatic}$ is the static power of the circuit block, and $P_{Fstatic}$ is the static power of the footer transistor. Our modeling results show that the sleep state can cut the static power of SRAM arrays in half while still retaining data in standby. This fits well with the published data of the 50% reduction of leakage power in the Intel 16MB L3 cache [19] because of using sleep transistors.

$$P_{static} = P_{Cstatic} + P_{Fstatic} \tag{2.12}$$

$$P_{Cstatic} = (V_{DD} - V_{GND}) * I_{sleep} \tag{2.13}$$

$$P_{Fstatic} = V_{GND} * I_{sleep} \tag{2.14}$$

2.4.2.2 Power-saving Models of Logic and Memory Structures

The circuit block shown in Figure 2.6 can be a single gate, a circuit block (e.g. an adder or a cache block), or a complete unit. When the sleep transistor is implemented in every single gate (cell), it is a fine-grain power gating style and called MTCMOS[18]. On the other hand, when the sleep transistors are shared in a circuit block, it is a coarse-grain power gating style. The advantage of the fine-grain sleep transistor implementations is that the virtual power nets are short and hidden in the cell. However, the fine-grain sleep transistor implementation adds a sleep transistor to every MTCMOS cell, which results in a significant area increase. The fine-grain sleep transistor implementation also suffers from high IR-drop variation in the cell and hence performance variation. The main advantage of the coarse-grain power gating is that sleep transistors share charge/discharge current. Consequently, it introduces less IR-drop variations than the fine-grain implementations. Also, the area overhead is significantly smaller due to sharing among the sleep transistors. Most power-gating designs prefer the coarse-grain sleep transistor implementation than the fine-grain implementation which incurs large area penalty and higher process-voltage-temperature (PVT) sensitivity. Therefore, we use a coarse-grain sleep transistor implementation in McPAT's models.

As shown in Equations 2.8 to 2.14, sizing of the footer transistor is critical to achieve good trade-offs between static power saving and wake-up penalty. According to [3], we choose the width of footer transistor (W_{footer}) as 15% of the accumulated width of transistors in the charge/discharge path in the circuit block when exiting the power-saving states. The charge/discharge path depends on the inputs of the circuit block, and we assume the pull-down network is open when the circuit clock wakes up. Therefore, $Wfooter = 15\% * \sum C_{NMOS}$. As $Wfooter$ is

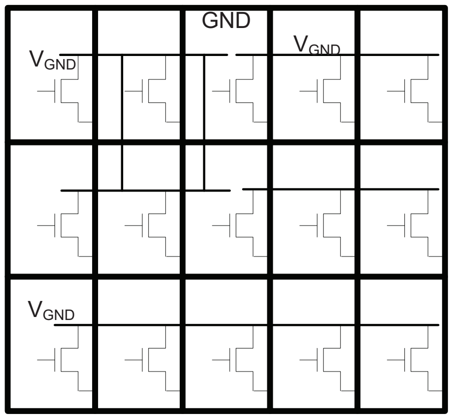


Figure 2.7. Distributed sleep transistor.

usually large, it can be implemented as a group of parallel-connected transistors. The wake-up time and energy remain the same. Although the area also remains approximatly the same, the area efficiency can be better since narrower and longer channels can be used for virtual ground and the aspect ratio of the virtual ground channel can be adjusted according to the width of the circuit block. As shown in Figure 2.7, the number and distribution of sleep transistors depends on the area of the function unit/circuit block. McPAT tries to fill the width of a circuit first then increase the height to accommodate sleep transistors.

In a memory array, the footer transistor can be implemented on a per row basis [65]. But this implementation tends to have a larger IR drop or more area overhead. Our models follow a section-based implementation as in [111], where the footer transistors are implemented inside each subarray as shown in Figure 2.8. Similar to the original memory models in CACTI, the pre-decoder block also has the 2-4 decoder to choose the sleep modes, and each memory subarray contains sleep transistors which result in area overhead, wake-up delay, and wake-up power.

As in real industrial high performance processors, McPAT assumes that circuit

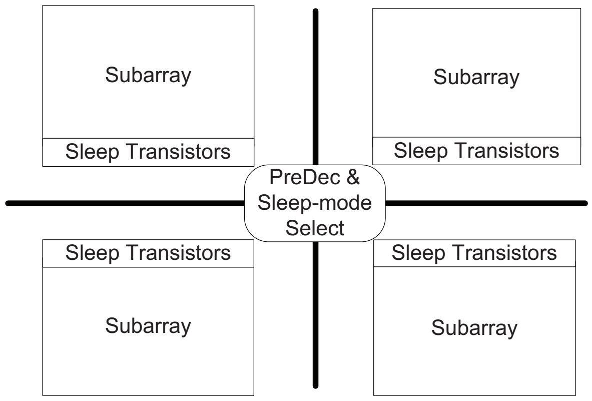


Figure 2.8. A mat with four subarrays with associated power-saving logic.

blocks can retain states only when they are in *sleep* mode. If a user or performance simulator chooses to set hardware units into dream or snore state, McPAT assumes that all the states/contents in the circuit blocks that are in *dream* or *snore* will be lost (McPAT does not model the extra hardware to save the circuit state since the state can be huge, e.g. contents of last-level cache). The McPAT user needs to make sure that the performance simulator is configured properly to reflect this wake-up overhead, for example the whole cache contents are flushed and the cold-start effects on the cache need to be appropriately reflected in the statistics that are sent from performance simulator to McPAT.

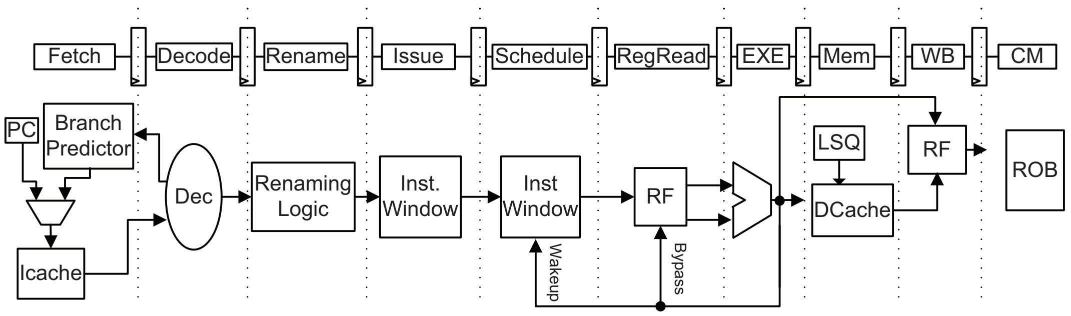
CHAPTER 3

ARCHITECTURAL LEVEL MODELING

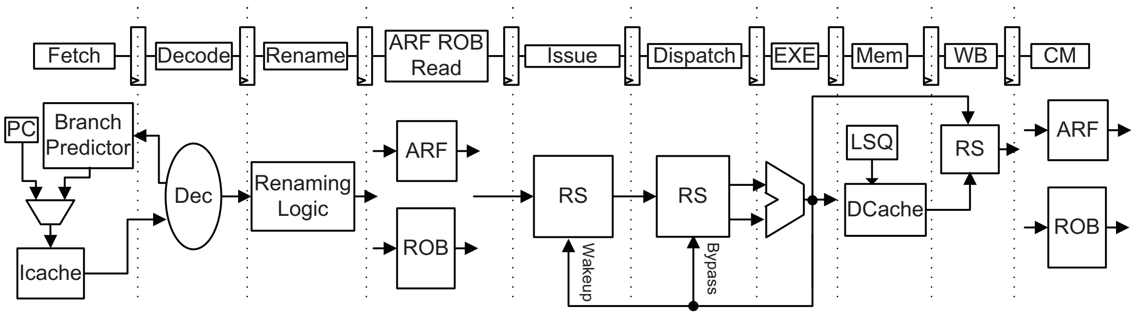
This chapter introduces the architecture level models in McPAT. The highest level is the architecture level that represents the basic building blocks of a multicore processor system. McPAT is designed to be flexible enough so that it can work with various performance simulators, and it is also designed to be accurate enough so that the results are widely applicable. One big challenge of achieving these two goals simultaneously is that different performance simulators differ greatly on the level of detail of the statistics they can provide and therefore McPAT might not have a way to fill in all missing statistics necessary to compute runtime power. For example, SimpleScalar [8] only provides the number of memory instructions, while M5 [13] can give the number of accesses to a load queue and a store queue. In order to solve this problem, we made detailed default assumptions on the architecture level based on real designs of high performance processors.

3.1 Core

Figure 3.1 shows McPAT's default out-of-order core models. The reservation-station-based OOO model is extracted from the Intel P6 [41] architecture, and the physical-register-file based OOO model is extracted from the Intel Netburst [37]



(a) McPAT model for physical register based OOO core.



(b) McPAT model for reservation station based OOO core.

Figure 3.1. Out-of-order core model in McPAT

architecture and the Alpha 21264 processor. By default, both OOO models are assumed to have ten pipeline stages in McPAT's modeling framework. In-order processors are modeled in a similar way but with only five stages including fetching, decoding, execution, memory access, and write back. A McPAT user can override the number of pipeline stages of both in-order and OOO processors. A core can be divided into several main units: an instruction fetch unit (IFU), an execution unit (EXU), a load and store unit (LSU), a memory management unit (MMU), a renaming unit, and an out-of-order (OOO) issue/dispatch unit for OOO processors. Each of them can be further divided into hardware structures. McPAT models each of these units in detail at architecture level.

3.1.1 Instruction Fetch Unit (IFU)

McPAT models the IFU as three major parts: the instruction cache together with its cache controller, the instruction buffer, and the branch predictor. The instruction cache is mapped to memory arrays and modeled using McPAT's internal CACTI module. The cache controller is modeled as three major structures: the miss buffer (miss status handling registers (MSHRs)) [53], the fill buffer, and the prefetch buffer. Almost all current high-performance processors support non-blocking caches. A plain cache can only handle one outstanding request at a time. If a request is made to the cache and there is a miss, the cache pipeline must wait for the memory to supply the value that was needed, and until then it is "blocked". By adding a special miss buffer, the cache can store the missed memory requests into the miss buffer and continue working on later requests while waiting for memory to supply previous misses. Although usually the miss buffer is designed for load misses since load misses are much worse than store misses, McPAT assumes that both load and store misses will setup an entry in the miss buffer based on the miss address file (MAF) design of Alpha 21264 and the non-blocking cache of Intel P6 [41]. The miss buffer is modeled as a fully associative cache which has a non-tag portion and a CAM-based tag portion. The non-tag portion holds the data, the internal cache address, and the destination register, while the tag portion contains the address. McPAT assumes that each cacheable miss uses one CAM operation.

The fill buffer is also modeled as a fully associative structure using the internal CACTI module based on the Sun Niagara [97]. The data portion contains cacheline-sized entries to store data from the lower level memory (such as L2 or DRAM) before it fills the current cache. Addresses are stored in the CAM portion for maintaining the age ordering in order to satisfy coherency conditions. Although it contains a cacheline per entry, the block size of the fill buffer is not the instruction cache line size but only the size of the machine word. This is because the data bypass allows the instructions pending on cache misses to enter the pipeline as soon as the critical machine-word arrives at the fill buffer. After all transfer segments arrive, the fill buffer assembles them into a cache line and then write the cache line into the cache. As most simulators do not model miss buffers and fill buffers, McPAT uses the inherent internal relationship between the buffers and cache access patterns provided by the performance simulator to model the operations on miss and fill buffers. Every cache miss will access the miss buffer. Every data block written into the cache will access the fill buffer. When data comes back to the fill buffer, this means that a miss has been serviced, and the miss buffer is accessed again to read out the instruction that can be re-issued.

Prefetch logic is also an important part of cache controller hardware and is modeled in McPAT. Prefetching instructions and/or data before they are requested by the processor can hide the memory latency greatly. Since directly prefetching data into the caches may cause cache pollution as the prefetched data may replace the cache entry resulting in later cache misses, McPAT assumes that an external buffer is used to hold the prefetched data as in Sun Niagara design [97]. Unless the performance simulator provides the prefetch information, McPAT assumes that the processor always fetches two blocks on a miss: the requested block and the next consecutive block as in the Sun Niagara [97]. The requested block is placed in the instruction cache when it returns, and the prefetched block is placed in the instruction prefetch buffer. All the modeled buffers have the same number of ports as the plain instruction cache which has one port by default. McPAT does not model the state machine inside the cache controller in the current version.

Branch predictor hardware is modeled as three main parts: the branch predictor, the return address stack (RAS), and the branch target buffer (BTB). The default predictor in McPAT is modeled as the tournament predictor as in Alpha 21264 [47]. The predictor has three RAM structures: the global predictor, the local predictor, and the chooser. Both the global predictor and the chooser are single-banked RAM arrays, while the local predictor is a double-banked array, with one bank being used for first-level prediction and another bank being used for second-level prediction. The branch target buffer is modeled as a normal cache. The RAS that is accessed when function calls and returns happen is also modeled as a RAM array. The number of ports of all arrays in the branch prediction structures are assumed to be the same as that of the instruction cache.

An instruction buffer is a part of the IFU that temporarily stores the instruction stream before sending it into the decoding stage, and it is mapped to a RAM array at the circuit level.

3.1.2 Renaming Unit

The renaming units are very important for OOO processors. They consume significant power and have high power density [32]. Figure 3.2 shows the renaming logic model in McPAT. As shown in Figure 3.2, there are three major parts in a renaming logic: the register alias table (RAT), the dependence check logic, and the free list buffer. The free list buffer is modeled as a multiported FIFO mapped to an SRAM array at the circuit level. The RAT is modeled as a RAM or CAM array depending on the renaming scheme and is mapped to a RAM or CAM structure at the circuit level. Dependency check logic is modeled analytically as random

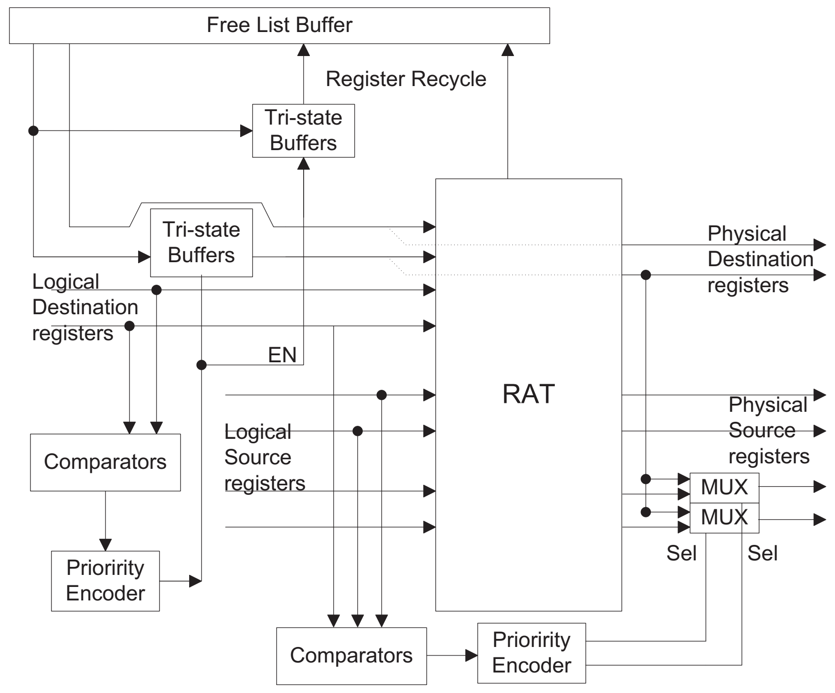


Figure 3.2. Architectural models of renaming logic in McPAT.

logic. McPAT also considers the instruction decoder as part of this unit since its outputs are steered into the renaming logic and later pipeline stages.

The renaming operations proceed as follows. At every cycle, a batch of instructions are fetched and decoded. These instructions form the current renaming pool. The rename logic is responsible for removing the false data dependencies including write-after-write (WAW) and write-after-read (WAR) and maintaining true data dependencies of read-after-write (RAW) among instructions in the current renaming pool and in-flight. During the renaming process, the architectural destination register of each instruction in the current rename pool is renamed with a physical register, taken from a pool of available registers (free list), so that no two instructions with the same destination register are in the processor pipeline at the same time. When physical registers are assigned to the architectural destination registers, the designators of all the architectural destination registers of instructions in the current renaming pool need to be compared against each other to avoid mapping same physical register to different architectural registers. The comparison is done by the dependency check logic as shown in left side of Figure 3.2. As a result, the WAW data dependencies are removed. Since each logical destination register is assigned a new physical register from the free list, this physical destination register is different from the source registers of instructions in-flight and instructions in the current renaming pool. Therefore, WAR data dependencies are also removed, and only RAW dependencies are left. The RAT is maintained to translate architectural source registers of an instruction into physical register designators. In every clock cycle, the name mapping of the instruction source registers is read from the RAT (by using the architectural register designator as the index in RAM-based RAT or by associative search using the designator as the key word in CAM-base RAT), while the new mapping of the destination registers is written in the RAT. At the same time, the overwritten designators must be read and recycled to the free list buffer when the last instruction that uses the designators commits.

In order to maintain RAW dependencies, another section of the dependency check logic is used to detect cases where the registers being mapped are destination registers of an earlier instruction in the same renaming pool. The dependency check logic compares source architectural register designators of instructions in the same rename pool against each other. A match in the dependency check logic indicates that an RAW is within the current renaming pool. The output MUXes of the dependency check logic select the mapping used for destination register in the current renaming pool, instead of the physical register found in the RAT. At the same time, the new designator of the source register overrides the old physical register mapping in RAT. In case of multiple matches in the dependence logic, a priority encoding logic selects the latest match in fetch order as the valid mapping of the source register. This source physical register designator is sent to later pipeline stages. If dependency check logic finds no matches and there is a valid mapping found in the RAT, the mapping found in RAT is passed to later pipeline stages and the RAT entry stays valid.

3.1.2.1 Register Alias Table (RAT)

McPAT supports both a RAM-based RAT as in MIPS R10K[109], Intel P6 architecture [41], and Intel NetBurst architecture [37] as well as a CAM-based RAT as in Alpha processors[47]. The RAM-based RAT is modeled as an SRAM array with the number of entries being equal to the number of architectural registers. The entries are indexed by the designator of the architectural registers. The physical register designators for the corresponding renamed architectural registers are stored in the entries. For the CAM scheme, the RAT is modeled and mapped to CAM array models at the circuit level with the number of entries being equal to the number of physical registers (or reorder buffer entries). Each entry has two data fields: one data field contains the designator of the architectural register that is mapped to this physical register, and the other data field contains a valid bit to indicate that the current mapping is valid. During the renaming operation, the designator of the architectural register is applied on the search line of the CAM. An associative search is performed to find a match between the data on search lines and the data stored in the cross-coupled inverters inside the CAM cells. If there is a match and the valid bit is set, the match line will be activated. An external encoder will encode the match line into the designator of the corresponding physical register and send the designator of the physical register to components that need the information. The CAM scheme is believed to be less scalable than the RAM scheme since the number of entries of a CAM-based RAT grows proportional to issue width [76]. However, the CAM scheme is advantageous in terms of checkpointing, and this advantage can be seen from McPAT results.

For physical register-based cores, the physical register file holds both speculative and non-speculative data. However, the non-speculative subset of the data must be explicitly denoted when the the processor state must be saved in case of interrupts/context switches. This requirement of denoting the subset of the non-speculative data leads to the dual-RAT technique [37]. McPAT models the dual-RAT scheme in its physical-register-based core models. A front-end RAT (FRAT) is used for register renaming as usual, while a retire RAT (RRAT) is used to store the current mapping for architectural registers. Since the FRAT contains the latest renaming information, there is no change for operation of the renaming unit with single RAT. An RRAT is only accessed during the instruction commit stage and does not affect the FRAT. The FRAT is modeled using both RAM and CAM schemes in McPAT. The RRAT is always modeled as a RAM structure with its number of write ports equal to the commit width of the modeled core.

McPAT also models checkpointing hardware in renaming logic based on the designs of global checkpointed RATs in [31, 84]. The renaming unit needs to be recovered to correct non-speculative states when branch mispredictions happen. There are three major techniques used in current high performance OOO processors for renaming unit checkpointing and branch misprediction recovery: (1) using a reorder buffer (ROB) as in the MIPS R10K to save non-speculative mapping, (2) using global checkpoints (GCs) to take a complete snapshot of the RAT upon renaming a branch instruction as in the Alpha 21264, and (3) not to checkpoint the RAT to achieve a simpler designs at the expense of sacrificing performance some as in the Intel P6 and Netburst architectures.

In the ROB checkpointing scheme, the recovery time is proportional to the number of instructions on the wrong path. This approach also increases the size and power of the reorder buffer. McPAT currently does not support this approach but supports the other two checkpointing schemes. The major difference between the use of checkpointing or not is that processors without RAT checkpointing must stall the renaming unit and wait for the OOO core to drain out non-speculative instructions by letting these instructions commit to the architectural register file or RRAT. A processor with RAT checkpointing can start to restore the renaming map immediately after the misprediction is detected. A processor without renaming checkpointing has to pay the penalty of execution-to-retirement delay, which is a variable factor that depends on the number of instructions between the executing branch and the retirement point and can be huge when the machine is full of instructions. McPAT supports both approaches to enable the comparison between alternative approaches.

Since most performance simulators do not model the internal operations of renaming units in detail, McPAT models the internal operation by itself and only requires the access counts to the whole renaming unit. McPAT assumes the RAT needs $2 * W$ read ports [94] and W write ports, where W is the issue width.

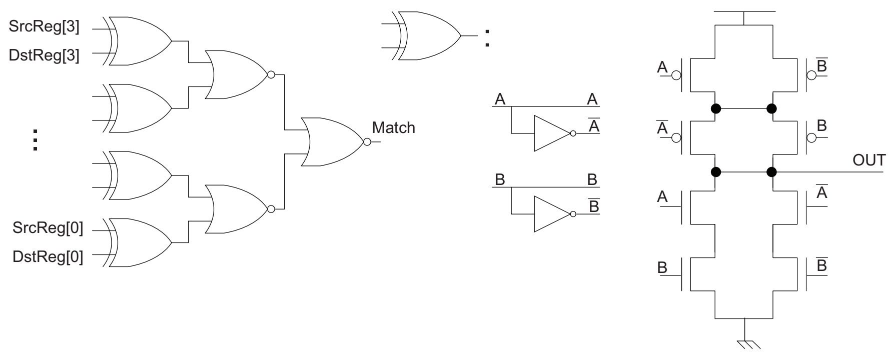


Figure 3.3. Comparator Sets of dependency check logic (DCL) modeled in McPAT

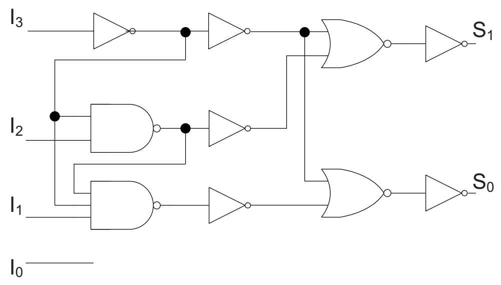


Figure 3.4. Priority Encoder of DCL modeled in McPAT

3.1.2.2 Dependency Check Logic (DCL)

As shown in Figure 3.2, there are two sections of dependency check logic (DCL). The first section checks the WAW dependency among destination registers of instructions in current renaming pool. This dependency check logic has three major parts: the comparators, the priority encoders, and the tri-state buffers. When a dependency is detected among destination registers, the output of the priority encoders act as the enable signals, disabling the write path to the RAT and enabling the recycle path to the free list buffer. The comparators are modeled based on [14]. We model a a scalable priority encoder as shown in Figure 3.4. Since McPAT supports processors with issue width up to eight it is important to have the DCL logic scalable so that it will not limit the clock rate. In the priority encoder the signal with lowest priority, I_0, does not connect to the main logic. When I_0 is set, the output of the priority encoder will still be 00, but the match signal from comparator sets will indicate that data dependencies are detected on instruction associated with I_0. The circuit of comparator sets is shown in Figure 3.3. The total number of comparator sets of this section is shown in Equation 3.1, where S is the total number of sets needed for destination register renaming and W is the decode width.

$$S = W * (W - 1) \tag{3.1}$$

The second section of DCL is shown at the bottom of Figure 3.2. This section of dependency check logic is modeled as multiple sets of comparators, priority encoders, and multiplexors according to [14]. It detects RAW, chooses the correct mapping of the source registers, and sends them to later pipeline stages as mentioned earlier. The number of the comparator sets used for second stage of the dependency check logic is twice the number used in comparing the destination registers. Therefore, the total number of comparator sets is shown in Equation 3.2, where S is the total number of sets needed for destination register renaming and W is the decode width.

$$S = 3 * W * (W - 1) \tag{3.2}$$

3.1.2.3 Power, Area, and Timing of Renaming Logic

When modeling timing, the internal CACTI module is used to model the array structures, namely, the RAT and the free list buffer. Since McPAT models power and area of target processors for given timing constraints, it focuses on the critical path for timing.

The total delay of the renaming logic is shown in Equation 3.3, where $T_{FreeList}$ is the access time of the free list buffer, $T_{DCLsec1}$ is the delay of the first section of dependency check logic that is used to detect WAW, $T_{DCLsec2}$ is the delay of the second section of dependency check logic that is used to detect RAW, $T_{RATread}/T_{RATwrite}$ is the read/write latency of the RAT (in a dual RAT scheme, FRAT determines the timing of the critical path), and $T_{DCLSec2MUX}$ is the latency of the MUXes at the final stage of the dependency logic. The total delay must be less than the target clock period in order to satisfy the timing constraint.

For each instruction that has register source operands, McPAT assumes it accesses dependency check logic and reads the RAT once for each of its source operand registers. For each instruction that has register destination operands, McPAT assumes it accesses dependency check logic and writes the RAT once for each of its destination operand registers. In a dual RAT scheme, RRAT is updated when an instruction commits. Energy consumed by each access is computed as shown in Equation 3.4, where E_{DCL} is the energy per access of the dependency check logic, E_{RAT} is the energy per access for the RAT which depends on the operation type (in a dual RAT scheme, E_{RAT} is the total energy of FRAT and RRAT for the same instruction), and $E_{FreeList}$ is the energy per access of the free list buffer.

The area of the renaming unit is shown in Equation 3.5, where A_{DCL} is the total area of the two sections of the dependency check logic, A_{RAT} is the area of the RAT (in a dual RAT scheme, A_{RAT} is the total area of the FRAT and RRAT), and $E_{FreeList}$ is the area of the free list buffer. We consider 10% overhead for macro level placement and routing when putting all individual components together to form the renaming logic.

$$T = max(T_{DCLsec1}, T_{FreeList}) + max(T_{RATread}, T_{RATwrite}), T_{DCLsec2} + T_{DCLSec2MUX} \tag{3.3}$$

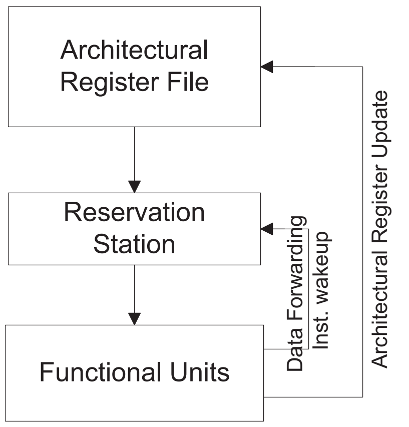
$$Energy_{RenamingLogic} = E_{DCL} + E_{RAT} + E_{FreeList} \tag{3.4}$$

$$Area_{RenamingLogic} = (A_{DCL} + A_{RAT} + A_{FreeList}) * 1.1 \tag{3.5}$$

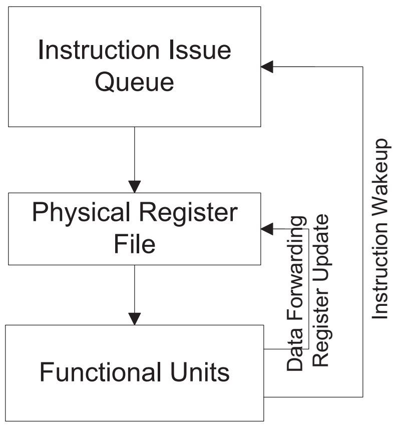
3.1.3 Scheduling Unit and Execution Unit

McPAT supports detailed and realistic models of the scheduling unit that are based on existing high-performance OOO processors. McPAT models both reservation-station-based (data-capture scheduler) architectures such as the Intel P6 [41] and physical-register-file-based (non-data-capture scheduler) architectures such as the Intel Netburst [37] and the DEC Alpha [47]. For an in-order processor, the scheduling unit degenerates to a simple instruction queue. In McPAT, the scheduling unit has three major components: the scheduling window, the reorder buffer, and the broadcast buses.

Figure 3.5 shows the conceptual view of both the reservation-station-based (data-capture scheduler) cores and physical-register-file-based (non-data-capture scheduler) cores. The two core models have the same in-order frontends that include instruction fetch stage, decoding stage, and register renaming stage. After



(a) McPAT models of Reservation Station based OOO Scheduler



(b) McPAT models of Physical Register File based OOO Scheduler

Figure 3.5. Conceptual view of architectural models in McPAT of reservation-station-based and physical-register-file-based OOO scheduler [91]

passing the frontend, instruction streams enter the OOO scheduler where the key difference of the two models resides. Renamed instructions are issued into the scheduling window, waiting for their source operand and functional units to be available before they can be dispatched to the execution stage. In the reservation-station-based model, the reorder buffer holds the speculative register data and the architectural register file holds the non-speculative data. Available source registers are read out before instructions are issued into the scheduling window. The designators of the unavailable source registers are also copied into the scheduling window and are used for matching the results from functional units and waking up appropriate instructions.

3.1.3.1 Reservation Station

The reservation station is modeled as a fully associative structure with a CAM array and an SRAM array. The CAM array holds the designators of the source register operands and is multiported with $2 * W * Nsource$ search ports and W write ports, where W is the issue width, and $Nsource$ is the number of source register operands per instruction. The SRAM array holds the information for instruction execution, including the instruction opcode and the source operands' data. Write accesses happen when instructions are issued to reservation stations, and search operations happen when results are available from the broadcast bus. Instructions will be woken up once the comparisons for source operand designators succeed. The CAM search operation serves as the wakeup logic for the reservation station. Since it is possible that multiple instructions become ready in the same cycle, selection logic is modeled according to [76]. In order to have dependent instructions issued consecutively, wakeup and selection must be done within one cycle. The worst case latency of the reservation-station-based design is shown in Equation 3.6, where $T_{CAMsearch}$ is the latency of search operation in the CAM, $T_{RSwrite}$ is the write latency of the whole reservation station array (both CAM and RAM), and $T_{InstSelection}$ is the latency of the selection logic. This latency must be less than the clock cycle time, and McPAT tries to optimize the reservation station to achieve this goal. A reservation station requires a large scheduling window in order to hold both tags and data in the same array structure. This does not make it easy for McPAT to satisfy the timing constraints of both the latency and throughput.

$$T = max((T_{CAMsearch} + T_{InstSelection} + T_{RAMread}), T_{RSwrite}) \tag{3.6}$$

3.1.3.2 Instruction Issue Window

In the physical-register-file-based model, the physical register file holds both the speculative and non-speculative data. Designators of both source and destination operands are stored together with the instructions in the scheduling window. The source operand designators are used to wake up instructions, and an instruction will read the register file to obtain data right before it is sent into the function units. The instruction issue queue in the physical-register-file-based model is modeled similarly to the reservation station in the reservation-station-based based model. The major difference is that the data portion of the instruction issue queue does not have the values of registers, which results in a smaller data array.

3.1.3.3 Result Broadcast Bus

As shown in Figure 3.5, reservation-station-based models combine result forwarding and instruction wake up into a single broadcast bus. In McPAT, the broadcast logic is modeled based on [76], including repeated wires, MUXes, and tri-state buffers. Also according to a die photo of the AMD Opteron processor (Hammer), we assume that broadcast buses route across all function units, the reservation station, the architectural register files, and the ROB. The buses are modeled using hierarchical wires, as described in Section 4. The total latency of the results bus must be less than that of the clock period.

As shown in the Figure 3.5, the physical-register-file-based model separates the wakeup path of the control flow from the forwarding path of the data flow. Therefore, McPAT models two different portions of the result broadcast bus. The data portion of the broadcast buses route over the functional units and physical register files, while only the designator portion of the broadcast bus needs to route over the instruction issue queue and ROB in addition to the functional units and physical register files.

3.1.3.4 Reorder Buffer (ROB)

The reorder buffer (ROB) is modeled as a circular RAM array in McPAT with the number of entries being equal to the number of in-flight instructions. McPAT assumes each issued instruction (including the load/store instructions) has been assigned an entry in the ROB. Each entry of ROB contains the designator, data from the destination register, and the instruction address. In McPAT's models, the reservation-station-based-model ROB has $3 * W$ read ports ($2 * W$ read ports are used to read source operands, and W read ports are used to read destination register values when the values need to be copied to architecture register file upon instructions commit) and W write ports, assuming issue width is equal to commit width. The physical-register-file-based-model ROB is modeled in a similar way except the ROB does not contain the register values. The smaller physical-register-file-based scheduling window has relaxed the timing constraints compared to the reservation-station-based models, which in turn enables a higher achievable processor/core clock rate. The latency of ROB must be less than the clock period set by the user.

3.1.4 Execution Unit

The execution unit and the scheduling unit are tightly coupled. The execution unit includes functional units and register files. It also shares the broadcast buses with the scheduling unit. The execution unit contains ALUs, FPUs, and register files. In our hierarchical framework, the ALU and FPU are mapped to the complex logic model at the circuit level.

3.1.4.1 Register Files

Register files are mapped to the array model in McPAT. The architectural register file and physical register file have the same number of ports, $2 * W$ read ports and W write ports, where W is the issue width. The access time of register file must be less than the that of clock period of the core, and the throughput must be at least equal to the clock frequency of the core clock rate. McPAT also models windowed register files. Windowed register files are not alternatives of register renaming, rather, they are techniques to improve the performance of a particularly common operation, the procedure call. These techniques can usually be found in Sun processor families. Unfortunately, windowed register files make register renaming extremely difficult to implement efficiently [93]. Without register renaming, there is little to be gained from OOO execution. Therefore, McPAT assumes that once the windowed register file is selected by user, the renaming unit and OOO logic will be disabled. The non-active context of windowed register file are modeled as a large RAM array with single read and write port. The windowed register file is accessed on every function call and return with multiple reads and writes, assuming 24 out of 32 registers will be stored or restored into the working architectural register files.

Since McPAT allows users to focus on high level architecture without worrying about low level configurations, it also provides default values for the physical register file size. The total number of physical registers in the worse case is shown in Equation 3.7. However, this worst case is too pessimistic. By default, McPAT assumes the total number of physical registers as shown in Equation 3.8.

According to [102], this assumption works well for processors with a reasonable numbers of threads.

$$
\begin{aligned}
Num_{PhysicalRegisters} &= Num_{In-flightInstructions} * 3 && (3.7)\\
Num_{PhysicalRegisters} &= Num_{threads} * Num_{ArchiRegs} + 100 && (3.8)
\end{aligned}
$$

3.1.5 Memory Management Unit

The memory management unit contains two main structures: an instruction TLB and a data TLB. Currently McPAT supports single level TLB structures. TLBs are mapped to array structures at the circuit level and modeled as fully associative caches. We assume that all L1 caches are virtually indexed and physically tagged. Therefore, the CAM portion of TLBs holds the virtual address, and the data portion holds the tag of the physical address for the cache. TLBs have the same number of ports as their corresponding caches. The timing requirement on a TLB is to finish an access within a single cycle.

3.1.6 Load and Store Unit

McPAT models the load and store unit (LSU) as three components: a data cache, a load queue, and a store queue. A data cache is modeled in the same way as the instruction cache in the IFU except that a write back buffer (WBB) is modeled as part of the cache controller if the data cache is set to be write-back. The WBB is used to store the evicted dirty cache lines. The evicted lines are streamed out to the lower level memory opportunistically. The WBB is divided into a RAM portion, which stores the evicted data until it can be written out, and a CAM portion, which contains the address.

Load queues and store queues are designed assist memory renaming and permit the out-of-order execution of memory instructions. Based on modern architectures such as the Intel P6 and Netburst, the load/store queue is modeled as two separate queues in McPAT and has three functions: (1) The load and store queues buffer and maintain all in-flight memory instructions in program order. (2) The load and store queues support associative searches to honor memory dependences. A load searches the store queue to obtain the most recent store value, and a store searches the load queue to find any premature loads (store-load order violation). (3) In multicore processors, the load and store queues support associative searches to enforce memory consistency.

McPAT assumes that for all memory instructions the ROB still holds all load and store instructions in-order; in other words, McPAT does not support the combined ROB and load-and-store-queues approach as in MIPS 10K. Similar to the Intel P6 and Netburst architecture, McPAT also assumes that the instruction window (reservation station or instruction issue queue) will hold the load and store instructions until the their operands are ready (both data and address for stores) and then dispatch the them into the LSU.

McPAT models two different types of load and store queues for in-order and out-of-order issued memory instructions. It is very important to model the two different types as this allows McPAT to reason about the statistics of load and store queues when the performance simulator cannot provide detailed information. McPAT models in-order issuing of memory instructions according to Intel P6 architecture. When a load enters the load queue to see if there are stores with conflicting addresses, all previous stores in program order are already in the store queue. Issuing loads and stores in-order ensures that all preceding stores to a load will be in the store buffer when a load executes. However, this limits the out-of-order execution of loads. Although a load instruction can be ready to be issued, a preceding store can hold up the issue of the load even when the two memory addresses do not alias since the in-order issue of the load and store must be maintained. McPAT also models OOO issuing of memory instructions based on the Alpha 21264 processor. Loads are allowed to issue out-of-order and be executed speculatively, and all speculatively finished loads are stored in the load buffer. If loads and stores issue out-of-order, it is possible for some of the stores that precede a load to still be in the scheduling window while the load has already enters the LSU, thus the load queue. Therefore, just searching the store buffer is not adequate for checking potential aliasing between the load and all its preceding stores. Instead, when a store issues, it performs an associative search in the load queue using its address. A match in the load queue indicates that an older store is issued after a younger load to the same memory address. Thus, the load queue must squash the load since the load got the wrong data. All the subsequent instructions stored in the ROB after the load also need to be flushed.

Since McPAT is designed to model multicore processors, load-load ordering also needs to be enforced because a remote store between loads can cause memory consistency problem. We assume that the load-load ordering is enforced at the hardware level as in the Alpha 21264 and MIPS R10K, although this may also be handled by software. When a load executes, it searches the load queue to compare its address to the addresses of all loads. If there is a match with a younger load issued out-of-order, then the out-of-order-issued load and subsequent instructions are squashed and fetched again.

The sizes of modeled load and store queues in McPAT are determined by the maximum number of in-flight load and store instructions. The numbers of ports of the load queue and the store queue are determined by the number of memory instructions that can be issued per cycle. Specifically, the store buffer has: $2*L$ read(compare) ports and L write ports, where L is the number of memory instructions can be issued per cycle. The load buffer has $2*L$ read(compare) ports and L write ports. In order to maintain load-store and load-load order, McPAT considers that for each memory instruction there are multiple inherent accesses to the load and store queue as shown in Table 3.1

TABLE 3.1

ACTIVITY FACTOR ASSUMPTIONS ON LOAD AND STORE QUEUE IN MCPAT

	Load instruction	**Store instruction**
In-order issue	1 write op (send the load to the load buffer); 1 read op (update the ROB or bypass); 1 CAM op in store buffer to check update values; 1 CAM op in load buffer to maintain load-load ordering;	1 write op (send in the store buffer); 1 read op (update main memory);
Out-of-order issue	Same as above	1 write op (send in the store buffer); 1 read op (update main memory); 1 CAM op in load queue to find younger loads that must be squashed

3.1.7 Pipeline Logic

Pipeline logic is modeled as a row of registers at each pipeline stage. By default, we assume that 5 and 10 stages per pipeline are used for in-order and OOO processors, respectively as shown in Figure 3.1. Users are also allowed to increase the number of pipeline stages. We assume that the main reason for increasing the number of pipeline stages is that the user defined target clock frequency cannot be satisfied during optimization. Therefore, we assume that increasing the number of pipeline stages for a certain function by X times will increase the number of inter-stage registers for the function by X times. Positive-edge triggered registers are modeled based on implementations using NAND2 gates, and negative-edge triggered registers are modeled based on implementations using NOR2 gates. In front of each group of pipeline registers, a NAND gate is modeled for clock gating.

3.1.8 Undifferentiated Core

In additions to the units modeled so far, there are also other glue/control logic units on silicon, such as the trap logic that handles exceptions. These logic units are hard to model since they are highly customized depending on the processor architecture and implementation. They usually are not as active as other components such as reaming logic, but they occupy die area and consume leakage power. They are modeled by the undifferentiated core function in McPAT.

An undifferentiated core is a core that does not have the components that are modeled analytically as mentioned above. The area of a native core is obtained by measuring the die photos of Sun processors including the 90nm Niagara [58] and the 65nm Niagara 2 [70] as well as Intel processors including the 90nm Pentium 4 [86], the 65nm Xeon Tulsa [83], the 65nm Merom [85] and the 45nm Penryn [30], and then curve-fitting to get equations with respect to processor type (in-order vs. OOO), number of pipeline stages and issue width. The transistor density of the undifferentiated core is obtained from the data of the Intel Penryn processor [30], which is used to calculate leakage power for the undifferentiated core.

3.1.9 Models of Multithreaded Processors

McPAT also models the power, area, and timing of multithreaded processors whether in-order (e.g., Sun Niagara) or out-of-order (e.g., Intel Nehalem [54]). There are three different types of multithreading: coarse-grained multithreading (CGMT), fine-grained multithreading (FGMT), and simultaneous multithreading (SMT). A CGMT processor issues instructions from one thread until the thread is blocked When a thread is blocked, then it switches to another ready thread. An FGMT processor switches threads every cycle and issues instructions from one thread every cycle. A CGMT processor is more likely to be built atop an in-order processor than an OOO processor because the in-order processor would normally stall the pipeline on a cache miss or branch. Although a FGMT processor can be implemented using an OOO processor as the baseline, it makes more sense to implement the FGMT processor based on an in-order processor since it only issues instructions from a single thread at one time. Current in-order multithreaded processors are usually implemented as chip multithreading (CMT) processors, such as Niagara, which is a combination of both CGMT and FGMT (Since the CMT processor performs FGMT over ready threads, and not-ready threads are dropped as in CGMT processor.) Finally, SMT processors such as as Intel Nehalem [54] leverage OOO superscalar processors.

McPAT already has all the models for in-order and OOO base processors.

TABLE 3.2

SHARING AND DUPLICATION ASSUMPTIONS FOR HARDWARE RESOURCES IN MCPAT FOR MULTITHREADED PROCESSORS

Architecture	**Private and Duplicated Units**	**Partitioned and Tagged Units**	**Share Units**
SMT	Instruction buffers, RAS, Architecture RF, RAT (FRAT and RRAT), inter-stage buffers	ITLB, DTLB, BTB, BPT, ROB, instruction decoders, load buffers, store buffers	Functional units, Icache, Dcache, L2cache, Reservation Station, instruction issue queue, physical register files
CMT	Instruction buffers, RAS, architecture RF, inter-stage buffers	ITLB, DTLB, BTB, BPT, instruction decoders, load buffers, store buffers	Functional units, Icache, Dcache, L2cache

Therefore, to model multithreaded processors, McPAT also must model the sharing and duplication scheme for hardware resources as well as the extra hardware overhead for the type of multithreading needed. McPAT models multithreaded architectures based on designs of Niagara processors [45, 51], Intel hyperthreading technology [52], and early research in SMT architecture [102]. Specifically, it is critical to recognize the shared units, partitioned units, and private units for both CMT and SMT processors and model the architectural level models in McPAT accordingly.

Table 3.2 shows the sharing and duplication assumptions for hardware resources in McPAT for multithreaded processors. All private units are modeled in a duplicated manner, with the number of copies being equal to the number of hardware threads. Partitioned units are modeled with extra thread ID in hardware. Each entry of the partitioned unit is assumed to include a CAM portion to store thread IDs as tags. Therefore some of these units, such as the branch predictor, are changed from RAM structures in single-threaded processors to cache-like structures with both tag and data arrays in multithreaded processors. Models of shared units in multithreaded processors are the same as that of single threaded processor models. Since McPAT computes energy per access to calculate final dynamic power, the duplicated units for non-active threads will not affect the dynamic power. However, these units will affect the leakage power and area of a multithreaded target design compared to a single threaded version.

3.2 Network on Chip (NoC)

An NoC has two main components: signal links and routers. For signal links between hops, we use hierarchical repeated wires, as described in Section 4. We use the same analytical approach as used in core modeling to model routers: breaking the routers into basic building blocks such as flit buffers, arbiters, and crossbars; then building analytical models for each building block. McPAT models power, area and timing consistently for NoCs.

3.2.1 Routers

As shown in Figure 3.6, McPAT models a traditional router with four stages. Although innovative research such as speculation and look ahead can reduce the number of pipeline stages, we choose to model traditional routers as we believe

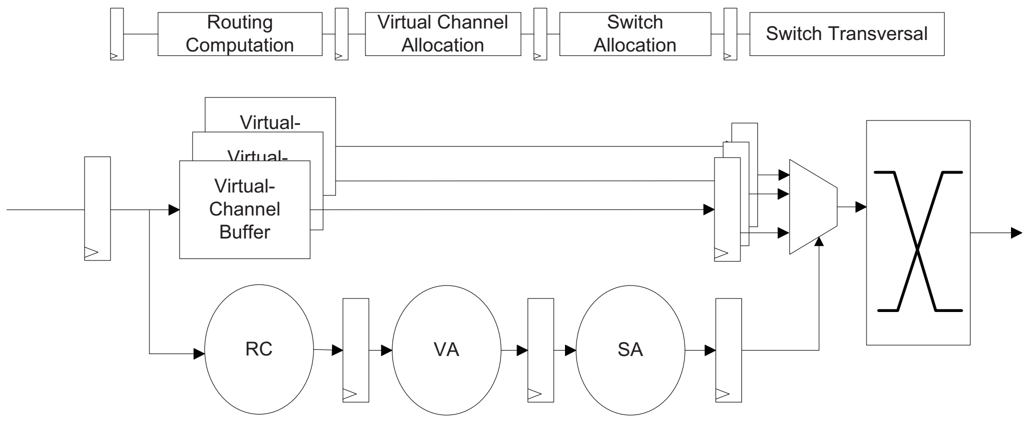


Figure 3.6. Router Model in McPAT

they are more generic. The four stages of a traditional router are routing computation (RC), virtual channel allocation (VA), switch allocation (SA), and switch transversal (ST). Each stage involves different hardware components. At the RC stage, information from the head flit is used by the routing computation block to select an output port (or ports). At the same time, the arriving flits are stored in the virtual-channel flit buffer. During the VA stage, the result of the routing computation is sent to the virtual-channel allocator. If successful, the allocator assigns a single output virtual channel on one of output channels. At the SA stage, each active virtual channel bids for a time-slice of the switch so that the buffered flit on the virtual channel can transverse the switch to the output unit containing its output virtual channel. If successful the flit traverses the switch at the ST stage. Also during the SA stage, a flit is read out from the flit buffer so that it will be ready to transverse the switch when the grant signal is set. However, it is also possible that the flit does not get a grant signal. In order to avoid wasting read operations and to have the buffer read in parallel with the switch allocation, we assume that all virtual channel buffers with a grant signal from the virtual-channel allocator have one flit read out and stored in a flip flop based buffer. The grant signal from the SA is used to choose which buffer gets to send its flit into the switch. Depending on the flow control mechanism, not all flits will access all hardware in each stage. McPAT requires activity factors on each component to compute total dynamic energy. If the performance simulator can not provide default statistics, McPAT assumes wormhole routing, where only head flits will go through all the pipeline stages. The body flits will bypass hardware such as routing computation logic. The latency of each pipeline stage is computed as in Equations 3.9, 3.10, 3.11, 3.12, where $Delay_{RC}$, $Delay_{VA}$, $Delay_{SA}$ and $Delay_{crossbar}$ are the latency of logic at each stage, respectively. $T_{virtual-channel-buffer}$ is the access time of the virtual channel buffer. $T_{pipeline}$ is the delay of a register. The maximum clock frequency is shown in Equation 3.13

$$\begin{aligned}
T_{RC} &= Max(Delay_{RC}, T_{virtual-channel-buffer}) + T_{pipeline} && (3.9)\\
T_{VA} &= Delay_{VA} + T_{pipeline} && (3.10)\\
T_{SA} &= Max(Delay_{SA}, T_{virtual-channel-buffer}) + T_{pipeline} && (3.11)\\
T_{ST} &= Delay_{crossbar} + T_{pipeline} && (3.12)\\
F_{router} &= 1/Max(T_{RC}, T_{VA}, T_{SA}, T_{ST}) && (3.13)
\end{aligned}$$

3.2.2 Flit Buffer

The flit buffers are one of the most important structures in a router. Not all flits use routing computation logic in the RC stage, but all flits need to access the flit buffer. We model the flit buffer as a SRAM FIFO using the internal CACTI module. Both the access time and cycle time of the flit buffer must be less than the clock period of the router to meet timing constraints. One important decision when designing a flit buffer is its sharing scheme between virtual channels or physical channels. The flit size of modern router can be large, and the number of ports of modern router can be high. Therefore, implementing large shared buffers with large numbers of read/write ports puts very high pressure on the flit buffer to satisfy the timing constraint and to achieve reasonable area and power results. According to the analysis in [23], McPAT uses the one flit buffer per physical channel implementation with one output port per virtual channel by default as shown in Figure 3.7.

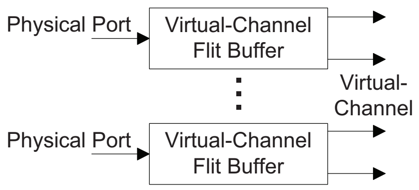


Figure 3.7. Flit Buffer organization Model in McPAT

3.2.3 Arbiter and Allocator

Both VA and SA stages need an allocator to map multiple requests to multiple available resources. An allocator performs matching between a group of resources and a group of requesters, and it can be implemented by two stages of arbiters. An arbiter assigns a single resource to one of a group of requesters. McPAT models a queueing arbiter and a matrix arbiter. Modeling a queuing arbiter is based on a RAM array. A matrix arbiter is modeled based on the circuit topology in

[23, 105].

McPAT models both input-first and output-first separable allocators. A separable allocator performs allocation as two sets of arbitration: one across the inputs and one across the outputs. In an input-first separable allocator, an arbitration is first performed to select a single request at each input port. Then, the outputs of these input arbiters are sent to a set of output arbiters to select a single request for each output port. In contrast, in an output first separable allocator, output arbitration is performed first and then the input arbitration is performed. The total latency of the two stages of arbiters in the allocator must be less than the target clock period to satisfy the timing constraints.

3.2.4 Crossbar

As shown in Figure 3.8, we model the matrix crossbar with the cross points being implemented as tri-state buffers. In a normal router architecture, although the whole router is pipelined we do not assume the crossbar itself is pipelined. The main reasons for not modeling pipelined crossbar switches are: (1) repeated wires in the crossbar route in both horizontal and vertical directions, makeing it very difficult to drop flip-flops underneath the wires. (2) only one cycle is allocated to switch transversal. Optimizing the crossbar is hard as there are multiple parameters that need to be optimized: the sizing of repeaters in the metal wires, the signal wiring pitch of the wire array, the sizing of tri-state buffers, and the sizing of input and output drivers. All these parameters depend on each other. McPAT first decides the signal wiring pitch. The tri-state buffers are then sized to have the same driveability as the optimized repeaters inside the wires of the crossbar. The tri-state buffers used in crossbars are shown in Figure 3.8. In order

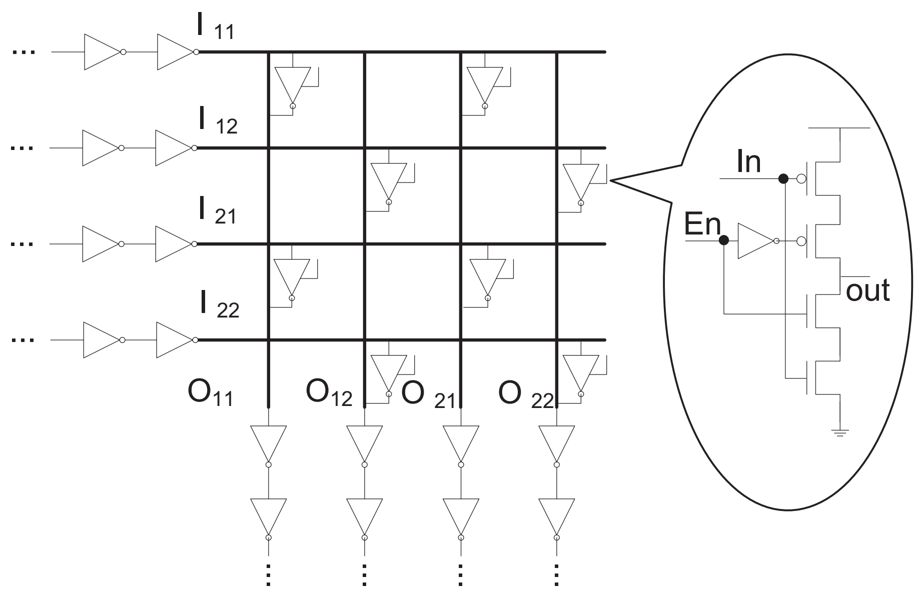


Figure 3.8. McPAT's crossbar model shown as a $2X2$ crossbar with data width of 2.

to provide same driveability, the size of PMOS and NMOS devices in the tri-state buffer are set to be double the size of PMOS and NMOS devices in the repeater. Then, input drivers and output drivers are sized using logical effort based on the gate load and the wire load of the "on" path in the crossbar.

McPAT also models a double-pumped crossbar [103] that reduces die area for on-chip interconnect intensive designs. McPAT assumes a five stage router when a double-pumped crossbar is used. The modeled router architecture with double pumped crossbar in McPAT is shown in Figure 3.9. The double-pumped crossbar is active on both positive and negative clock edges. Therefore, having virtual buffers read-ahead and using the SA grant signal to select data from multiple buffers will violate the timing requirements. The buffer read stage cannot be done in parallel with SA since otherwise the set-up time of the first flip-flop in a double-pumped crossbar cannot be satisfied. Unlike a single-pumped crossbar, a double

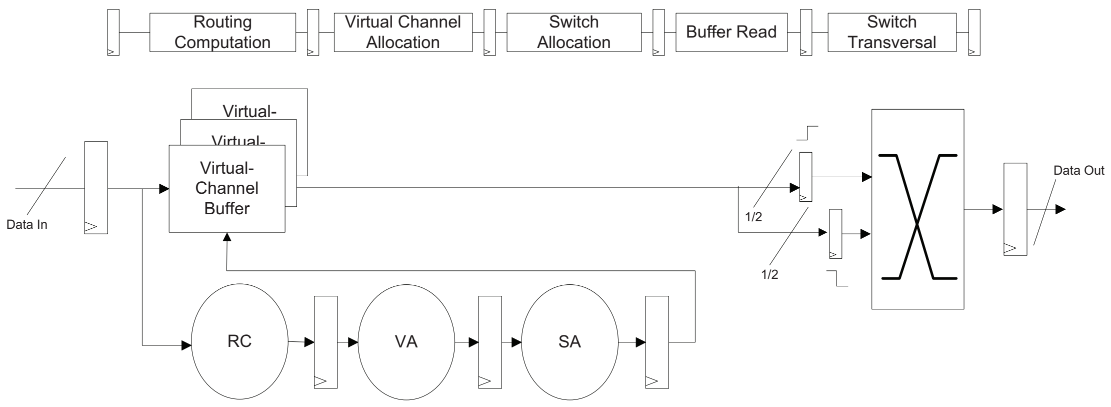


Figure 3.9. Router with double-pumped crossbar modeled in McPAT

pumped crossbar is a fully synchronized circuit. The delay of the double pumped crossbar is shown in Equation 3.14, which must be less than half clock cycle.

$$T_{xbar} = T_{M0} + T_{S0} + T_{Mux} + T_{xbarwire} \tag{3.14}$$

3.2.5 Inter-router Link

Inter-router links can consume 30% of total NoC power and need to be modeled carefully. McPAT assumes that all the inter-router links are repeated wires and appropriately pipelined when necessary. For the inter-router links, throughput is more important than latency since the links can be heavily pipelined. On the other hand, pipelining is not free as the flip-flops used in the pipelined occupy area and consume power. Therefore, McPAT first attempts satisfy the timing without adding flip-flops in the wire. Only when timing constraints cannot be satisfied, will McPAT insert flip-flops in the links.

During optimization, McPAT first attempts to assign the links to different metal layers in the hierarchical wire model at the circuit level and computes the delays of the link at each metal layer. If the timing constraint can not be satisfied at the current metal layer, McPAT will vary the signal wiring pitch, use double or triple wide metals, and move to higher levels of metal, trying to satisfy the timing requirement. It is important to note that the links are modeled as repeated wires, therefore, repeaters are included in all the computations. If all attempts fail, McPAT assumes the links must be pipelined in order to satisfy the timing constraints. The number of pipeline stages are computed according to Equation 3.15, where N_{stage} is number of pipeline stages needed for the link, $Latency$ is the latency of the unpipelined link, T_{Clock} is the target clock period for the NoC, and $Throughput$ is the user defined throughput per clock.

$$N_{stage} = (Latency/T_{Clock}) * Throughput \tag{3.15}$$

3.3 On-chip Caches

McPAT supports both private and shared/coherent caches. Private caches are modeled in the same way as instruction and data caches inside a core. It models coherent/shared caches by modeling the hardware that stores directory information associated with shared/coherent caches. Depending on the architecture, the hardware with directory information can be mapped to CAM structures at the circuit level as in Niagara processors [45, 51] or directory cache structures as in the Alpha 21364 [43]. The CAM implementation is usually used in shared cache structures, shadowing the tags of higher level cache to support quick lookup. However, this approach has high overhead since CAM is expensive and the size of the CAM is proportional to the total capacity of caches that share the same low level cache. The directory cache is designed as a dedicated cache for directory information. The complete information of the directory is assumed to be stored in main memory (or a special memory section). This approach is usually used with coherent caches or last level shared caches.

3.4 Memory controller

McPAT models on-chip memory controllers. The memory controller models follow a design from Denali [24] that contains three main hardware structures: 1) the frontend engine responsible for scheduling the memory requests, 2) the transaction processing engine that has the logic and sequencer to generate the command, address, and data signal, and 3) the physical interface (PHY) that serves as an actual channel of the memory controller for communicating off-chip to memory.

3.4.1 Front-end Engine

The front-end engine is responsible for reordering memory requests, buffering data, selecting memory requests, and sending them to the backend engine. There are three main components in the front-end engine: a request reorder buffer, a read request buffer, and a write request buffer. The request reorder buffer is mapped to a fully-associative structure in McPAT. Its CAM portion contains the memory request type, memory address, current page status in main memory, and memory channel ID. By associative searching in the request reorder buffer for each incoming memory request, the front-end can select appropriate memory requests based on a pre-defined scheduling policy. The data portion of the request reorder buffer contains an index of the memory requests that are stored in a read or write queue. It is possible that multiple memory requests are granted at the same time. Selection logic is needed to pick one request from the valid candidates. The selection logic is modeled in the same way as that used in wakeup logic in an instruction issue queue.

The read and write request buffers are modeled as RAM arrays. The data field of the buffers contains a physical memory address and memory data. Although each memory operation issued from a last level cache requests a whole cache block, we assume the memory controller supports a requested-word-first optimization [35]. Hence, the data widths of the read and write request queues are equal to the sum of the physical memory address, memory data bus width, and ECC bit width.

3.4.2 Transaction Processing Engine and Physical Link (PHY)

The transaction processing engine has the logic and sequencer to generate the command, address, and data signals, and is responsible for taking care of routine jobs such as refresh, ECC, and memory scrubbing. Modeling the transaction processing engine and PHY are difficult at an architectural level since they involve highly irregular logic, analog devices and transmission lines. We empirically model PHY and the transaction processing engine according to published data from Rambus [56] and AMD [6].

3.5 Clocking

Clocking circuitry has two main parts: the phase-locked loop (PLL) with fractional dividers to generate the clock signals for multiple clock domains, and the clock distribution network to route the clock signals. McPAT uses an empirical model for the power of a PLL and fractional divider, by scaling published results from Sun and Intel [4, 83]. The clock distribution network can be directly mapped to the clock network model at the circuit level.

CHAPTER 4

CIRCUIT LEVEL AND TECHNOLOGY LEVEL MODELING

This chapter presents the circuit level and technology level models we have developed for McPAT. Whenever possible, circuit level structures are modeled using analytical approaches including logical effort [99], Elmore delay, Horowitz model [40], and so on. For complex structures where analytical modeling is not possible, empirical models are used. Technologies in McPAT are modeled using MASTAR [88]/ITRS roadmap [88] and R. Ho's work [38, 39].

There are four categories of hardware structures in the circuit level models: arrays, hierarchical repeated wires, logic, and clock distribution networks. Combinations of the four categories of hardware structures form all the architectural level components.

4.1 Hierarchical Repeated Wires

Hierarchical repeated wires are used to model both local and global on-chip wires. Performance of wires is governed by two important parameters: resistance and capacitance. We model short wires using a one-section π-RC model [38] as shown in Figure 4.1, where R_{wire} and C_{wire} for a wire of length L_{wire} are computed by Equations 4.1 and 4.2. We use Equations 4.3 to 4.4 from [38, 39] to compute $R_{unit-length-wire}$ and $C_{unit-length-wire}$ of a plain wire. Figure 4.2 shows the wire capacitance model used in Equations 4.3 to 4.4 from [38].

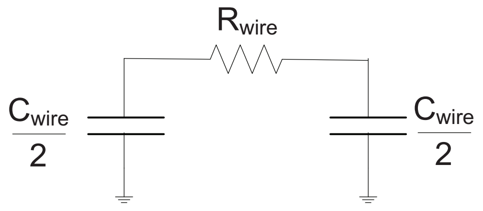


Figure 4.1. One-section π-RC model that we have assumed for plain wires.

$$R_{wire} = L_{wire} R_{unit-length-wire} \tag{4.1}$$
$$C_{wire} = L_{wire} C_{unit-length-wire} \tag{4.2}$$

$$R_{unit-length-wire} = \alpha_{\text{scatter}} \frac{\rho}{(thickness - barrier - dishing)(width - 2 * barrier)} \tag{4.3}$$
$$C_{unit-length-wire} = \epsilon_0 (2M \epsilon_{\text{horiz}} \frac{thickness}{spacing} + 2\epsilon_{\text{vert}} \frac{width}{ILD_{\text{thick}}}) + fringe(\epsilon_{\text{horiz}}, \epsilon_{\text{vert}}) \tag{4.4}$$

Wires scale more slowly than gates with respect to RC delays, and unbuffered wires cannot keep up with the improved transistor delay. For long wires, we use a repeated wire model [38] with optimized repeaters as in CACTI 6.0 [66]. The equations used to find optimal sizing of the repeaters are reproduced below, where c_0, c_p and r_s are constants for a given technology.

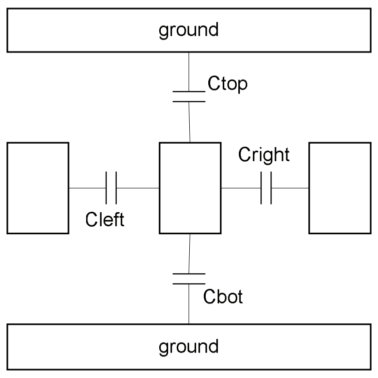


Figure 4.2. Capacitance model from [38].

$$L_{optimal} = \sqrt{\frac{2r_s(c_0 + c_p)}{R_{unit-length-wire}C_{unit-length-wire}}} \quad (4.5)$$

$$S_{optimal} = \sqrt{\frac{r_s C_{unit-length-wire}}{R_{unit-length-wire}c_0}} \quad (4.6)$$

We assume multiple metal planes for local, intermediate, and global interconnect, each with different wire pitches and aspect ratios. Our hierarchical wire model assumes ten metal planes, with four layers for local interconnect, and two each for intermediate, semi-global, and global interconnect.

Wires on different planes have different pitches and aspect ratios. The assignment of signals to wiring planes plays a key role in determining their power, area, and timing characteristics. McPAT's optimizer automatically assigns wires to planes to achieve specified objectives by trying different metal layers and varying effective wiring pitches. First, our model optimizes the buffers for each wiring plane. Then, the delays of buffered wires are calculated for the first assigned layer in the wiring plane hierarchy. If the current wiring layer cannot satisfy the delay constraints, the wires are moved to a higher wiring plane for lower latency, at the expense of a higher power and area penalty. The effective wiring pitches on semi-global and global planes are also increased to satisfy the target latency as described in [55]. Latches are also inserted and modeled when necessary to satisfy the target clock rate as mentioned in Section 3.2.5.

4.2 Logic

McPAT employs three different schemes for modeling logic blocks, depending on the complexity of the block. For highly regular blocks with predictable structures, such as memories or networks, McPAT uses the algorithmic approach of CACTI [101]. For structures that are less regular but can still be parameterized, such as thread selection or decoding logic, McPAT uses analytic models similar to those in [76] as explained in Section 3. Finally, for highly customized blocks such as functional units, McPAT uses empirical models based on published data for existing designs scaled to different technologies. For example, the ALU and FPU models are based on actual designs by Intel [64] and Sun [58].

4.3 Clock Distribution Network

A clock distribution network is responsible for routing clock signals of different frequencies to clock domains, with drops to individual circuit blocks. It is a special case of a hierarchical repeated wire network. We model it separately from the normal repeated wires not only because it has strict timing requirements with large fanout loads spanning the entire chip but also because it is wave-pipelined [106] so it is completely different from normally pipelined global interconnects.

We represent a clock distribution network using a separate circuit model that has three distinct levels: global, domain, and local. We assume an H-tree topology for global-level and domain-level networks and a grid topology for the local networks as shown in both the Niagara processors [51, 70] and the Intel Itanium processors [28, 62]. We assume the global-level clock distribution network can only be implemented using global metal layers. The domain-level and local clock distribution network can be implemented using both semi-global and global metal layers. NAND gates are used at all final clock heads to enable clock gating.

4.4 Arrays

McPAT models three basic arrays at the circuit level: RAM-, CAM-, and DFF-based arrays. RAM models are basic building blocks of both pure scratch-pad-based memory arrays such as buffers, queues and set-associative caches. McPAT uses internal CACTI [101] modules when modeling RAM structures. CAMs and fully-associative caches are also extremely important. As mentioned in Chapter 3, they are key building blocks for many components of multicore processors, including cache controllers, renaming units, load and store units, and memory management units. Therefore, the accuracy of CAM and fully-associative cache models play important role to overall accuracy of McPAT. McPAT implements a new CAM model to accurately reflect multi-banked and multi-ported CAM structure and also adds a write operation model and a non-associative-search read model. Cycle time of CAM is also added in McPAT. McPAT also adds a detailed DFF array model. Gate leakage models are also added to all array structures in McPAT

4.5 CAM—An Example of Circuit Level Modeling

As a detailed example, this section describes the circuit level for a content-addressable memory or CAM. Although CAM models are array models, they also incorporate examples of logic and hierachical wires as it sub-components. The CAM models developed for McPAT also incorporate significant changes from the original models in CACTI. A CAM is a memory that implements the lookup table function in pure hardware as shown in Figure 4.3. A CAM can perform three different operations: read, write, and search. Read and write operations are as the same as the corresponding operations in a RAM. A search operation in a CAM compares an input search data token against data stored in the memory structure, and returns a hit/miss signal and the address containing the matching data. When there are multiple matches, all addresses containing the matching data pass through a priority encoder and the address with highest priority is returned as the final result. Although a CAM mostly performs search operations, read and write operations are also important since write operations are used to update data and read operations are used in dual-mode components such as a renaming units of OOO processors. Hence, unlike CACTI that only models the search operation, McPAT models all three operations of a CAM in a unified and consistent manner.

A search operation in a CAM needs to search all addresses to find the matching data, which consumes lots of power. A multibanked CAM as shown in Figure 4.4 can reduce power significantly since each search operation only needs to search a bank. The target bank is decided by extra bank selection bits during search. The bank selection bits are also used to choose which bank should be selected for storing data in a write operation. The multibanked CAM array modeled in

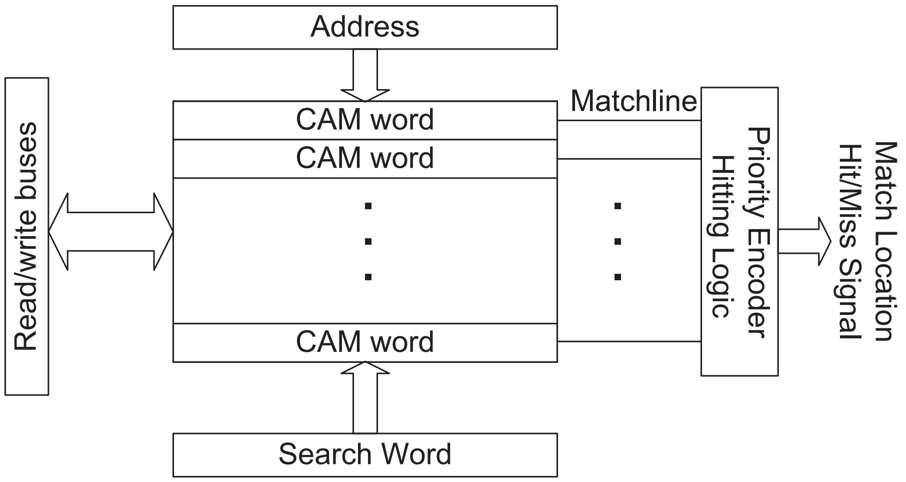


Figure 4.3. Conceptual view of a CAM

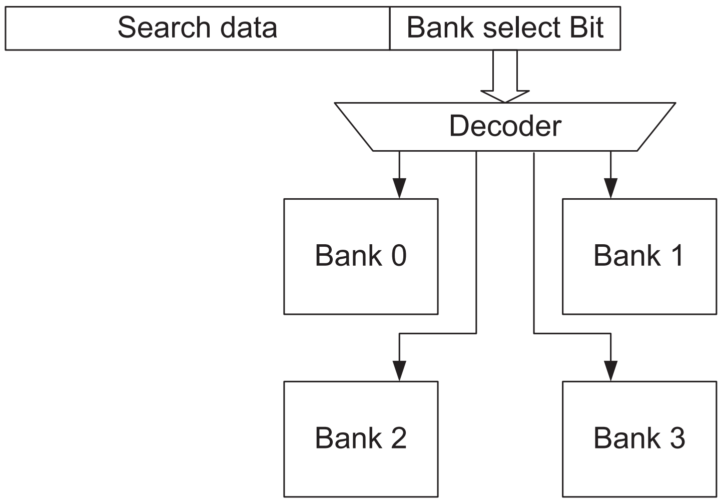


Figure 4.4. Multibanked CAM

McPAT is similar to the RAM array modeled in CACTI [101]: a CAM array consists multiple identical banks (N_{banks}). And each bank can be concurrently accessed and has its own address, data, and search buses.

In order to perform optimizations of a CAM array, McPAT's CAM model uses similar partitioning parameters as in its RAM model: N_{dwl}, N_{dml}, N_{dbl}, N_{dsl} and N_{spd} for each bank, where N_{dwl} = number of segments in a wordline of a bank, N_{dml} = Number of segments in a matchline of a bank, N_{dbl} = number of segments in a bitline of a bank, N_{dsl} = number of segments in a searchline of a bank, and N_{spd} = number of sets mapped to each bank wordline/matchline. Since an access to a CAM always needs the whole wordline/matchline, N_{spd} is always 1. When $N_{dwl} > 1$, multiple sets of address buses and row decoders need to be placed alongside each segment of wordlines. This increases the length of the matchline significantly and complicates the CAM array, which results in a longer latency search operation. Since search operations are very important for a CAM, we consider $N_{dwl} = 1$. McPAT supports N_{dml} as both 1 and 2. The detailed explanation about two-segmented matchlines, $N_{dml} = 2$, can be found later in this section. N_{dbl} and N_{dsl} are always the same and vary together.

Each block with continuous bitlines is a subarray that forms the smallest building block of a CAM. As shown in Figure 4.6, a CAM subarray is essentially a RAM subarray with matching circuitry inside each memory cell and special peripheral logic, including searchlines, matchlines, precharge circuit, hit/miss detection logic, and priority encoders.

As a CAM also needs to perform read and write operations as well as search operations, it has all the components of a RAM except the bitline muxes and sense amplifier muxes. Since only N_{dbl} and N_{dsl} can be changed freely, a CAM array can only grow subarrays in the vertical direction. This can lead to non-optimized aspect ratios, with the CAM array being tall and thin. A re-organization of subarrays in a bank as shown in Figure 4.5 is performed for each configuration.

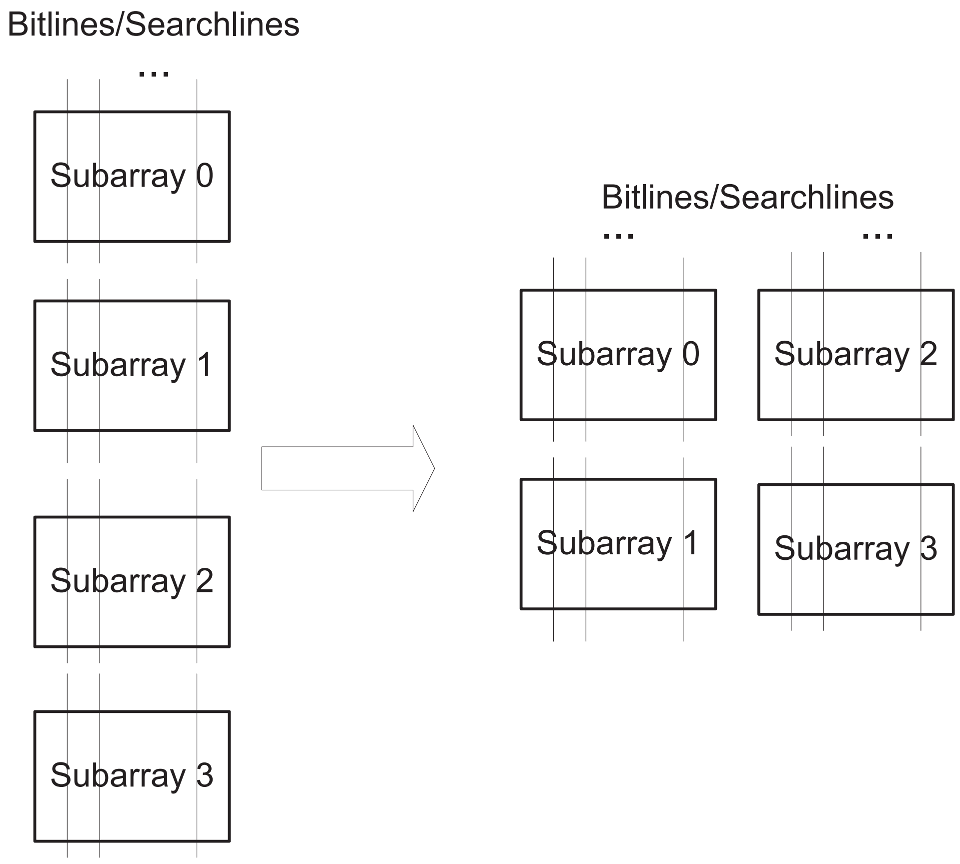


Figure 4.5. Re-organization of subarray for better aspect ratio

The number of subarrays in the horizontal direction and vertical direction after the re-organization are shown in Equations 4.7 and 4.8.

A read or write operation to a CAM bank only activates a subarray, while a search operation to CAM bank needs to activate all subarrays to perform parallel search. Thus, CAM models do not have the concept of subbank as in RAM models. On the other hand, the concept of mats as in RAM models is kept in the CAM models of McPAT. As shown in Figure 4.7, four identical subarrays and associated predecoding logic form a mat, the basic self-contained building block of a CAM array. It is important to note that unlike in RAM models where all

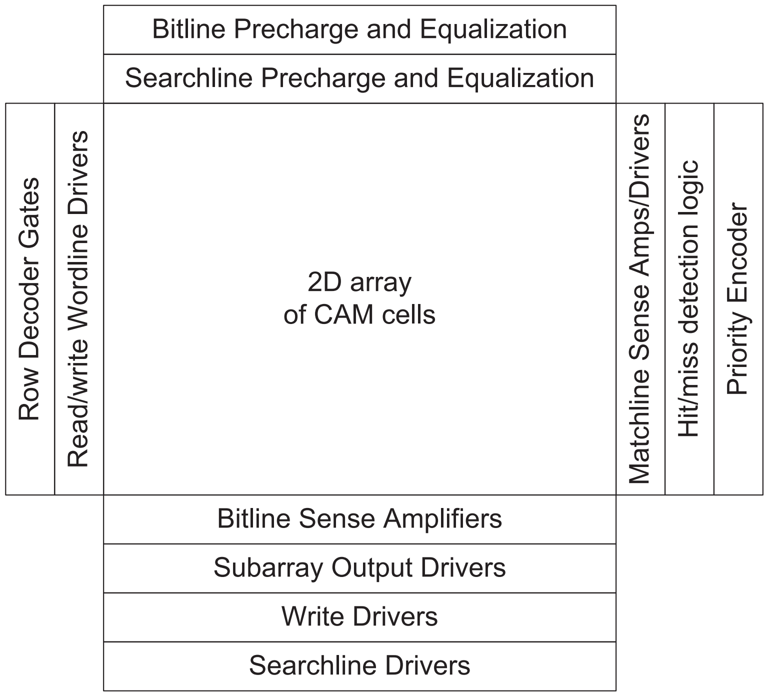


Figure 4.6. Subarray organization of CAM

subarrays in a mat are active during an access, read and write accesses to a CAM mat only activate one subarray in a CAM mat.

$$N_{Hsubarray} = \lceil \sqrt{N_{dbl}} \rceil \tag{4.7}$$

$$N_{Vsubarray} = N_{dbl}/N_{Hsubarray} \tag{4.8}$$

Inside a mat, a predecoding block shared by four subarrays is used for read and write operations. The main difference between CAM predecoding logic and RAM predecoding logic is that the CAM predecoding logic needs complete address bits, while the RAM predecoding logic only needs the address bits of a subbank. In a CAM mat, the higher order bits of the address are used to enable the appropriate

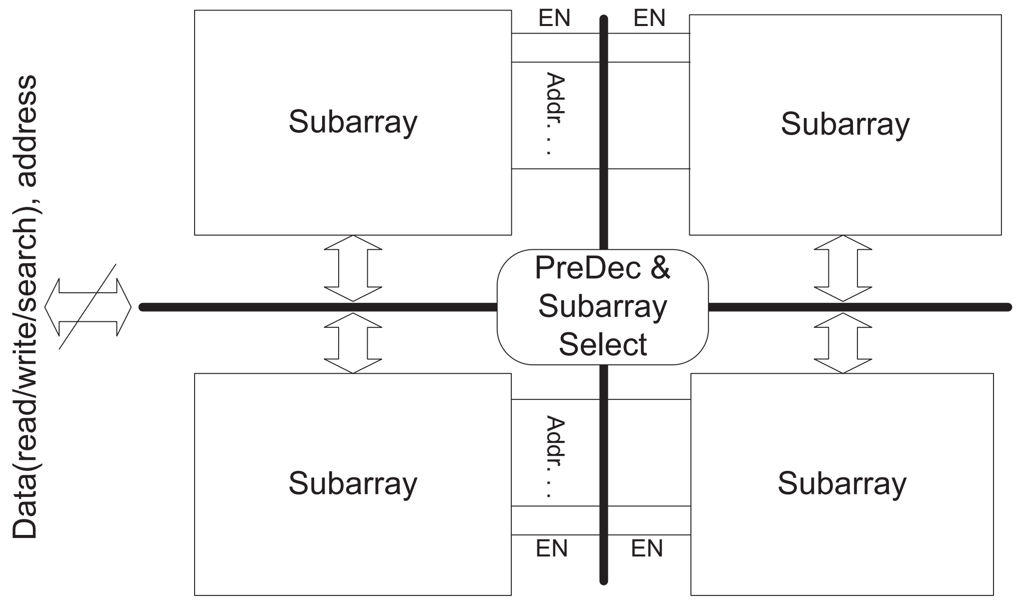


Figure 4.7. Mat organization of CAM

subarray, and the lower order bits of the address are decoded and used (together with the row decoder at the subarray side) to activate the proper wordline in the target subarray. As a result, the address Htree of a CAM needs a few extra bits in the vertical direction, compared to a RAM with the same layout of subbarrays.The read/write data Htrees of a CAM are properly gated for each access, while the search data bus broadcasts the search data to all subarrays.

4.5.1 CAM Circuit Models

Since CAM models are similar to RAM models at the mat level and beyond, except for the search data buses that broadcast data to all subarrays, we focus our description on the models of specialized CAM circuitry as well as the modeling of search operations at the subarray level. Figure 4.8 shows the CAM array with 10T CAM cells. The 10T cell provides a rail-to-rail voltage for all the transistors

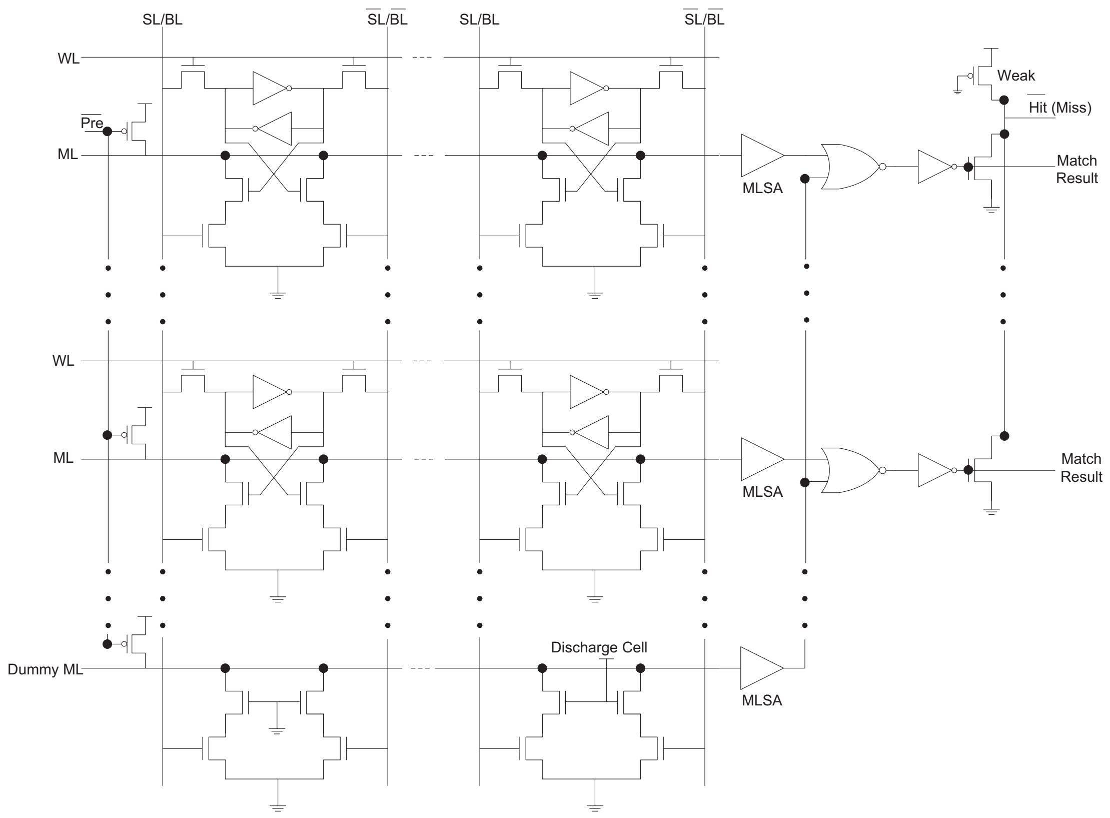


Figure 4.8. CAM subarray circuit

in the comparator part of the cell, which provides better noise immunity than the 9T CAM cell alternative. Therefore, we model CAM structures using 10T CAM cells in McPAT.

As shown in in Figure 4.8, A CAM has two key structures for each search port (besides the normal RAM parts): differential searchlines (SL and SLbar) and matchlines. A search operation has four major phases: searchline precharge, matchline precharge, applying search data, and matchline evaluation. First, the pair of searchlines (both SL and SLbar) are precharged low to shut off the pull down paths in all CAM cells. Thus, matchlines are disconnected from the ground.

Second, matchinelines are precharged high. Third, the data bits to be searched is driven by searchline drivers onto searchlines. Finally, the matchline evaluation phase is triggered. If the search word matches the word stored in the cells, the matchline stays high and indicates a match. Searchlines need to maintain a stable voltage level until the evaluation phase is over. A single bit miss will open a discharge path for the match line, causing the matchline to be low after the evaluation phase, which indicates a miss for the current comparison.

If there are multiple matches, a priority encoder selects the location with highest priority (usually the location with the lowest address) from all locations containing matched data. Then, the priority encoder generates the address of the match location. If there is always no more than one match in the CAM, the priority encoder will degrade to a simple encoder. Moreover, for systems such as fully-associative cache that do not need to send out the address, the encoder will degenerate to a driver chain.

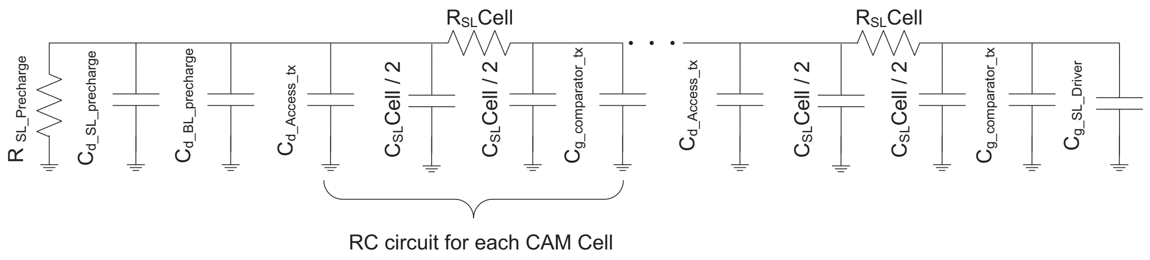
Searchlines can multiplex with bitlines of read-and-write ports or exclusive write ports as shown in Figure 4.8. This sharing reduces the overhead for maintaining multiple ports, but it also limits the CAM's capability to support search and read/write accesses simultaneously. McPAT supports both shared searchlines and bitlines as well as separate searchlines and bitlines.

Peripheral circuits, including the row-decoders for read/write, bitline sense amplifiers, write-drivers, and bitline precharge and equalization circuits are not shown in Figure 4.8, but are modeled using the same models as in CACTI. Although seachlines and bitlines can be multiplexed, an extra set of precharge and equalization circuits is needed for each pair of multiplexed searchlines since searchlines need to be precharged low and bitlines need to be precharged high.

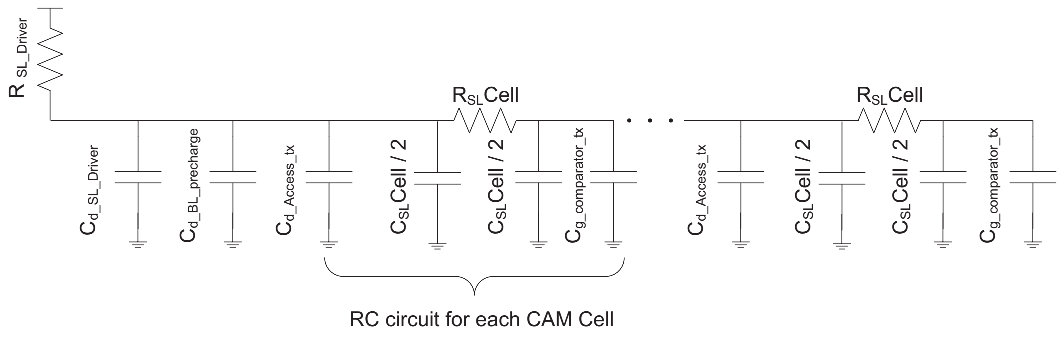
Usually, a CAM search operations are pipelined with a priority encoder in a different pipeline stage. Hence, the critical-path latency is mostly dictated by the access time of the matchlines and searchlines. In this section, we present analytical circuit models of the key parts of a search operation: matchlines and searchlines.

Figure 4.9 shows the equivalent circuit of CAM searchlines in their precharge phase and the data driven phase, where $R_{SL\_precharge}$ and $C_{d\_SL\_precharge}$ are the resistance and drain capacitance of the precharge transistor, R_{SL_driver} is the on-resistance of the PMOS in the searchline driver, $C_{d\_SL\_driver}$ is the capacitance of the searchline driver, including the both the PMOS and NMOS drain capacitance. R_{SL_cell} and C_{SL_cell} are the resistance and capacitance of searchline metal in a CAM cell. The equivalent circuit models assume that searchlines and bitlines are multiplexed. Therefore, the drain capacitance of the access transistors ($C_{d\_Access\_tx}$) and the capacitance of the bitline precharge and equalization circuit($C_{d\_BL\_precharge}$) appear in the models. McPAT also supports separate searchlines and bitlines, where $C_{d\_Access\_tx}$ and $C_{d\_BL\_precharge}$ are not in the circuit model. Unlike read operations that can use sense amplifiers to reduce voltage swing, searchlines use rail-to-rail voltage swing.

Matchline delay is another key components of the CAM access time. As shown in Figure 4.8, the number of discharge paths that a matchline can have depends on the number of unmatched bits between the search word and the stored word. Therefore, the equivalent resistance of all discharge paths is $R = R_{comparator}/M$, where $R_{comparator}$ is the on-resistance of a single discharge path in each CAM cell and M is number of unmatched bits. As a result, the latency of the matchline varies from one match operation to another. This variation creates timing control difficulties for the clocked circuits such as the searchline precharge circuit, match-



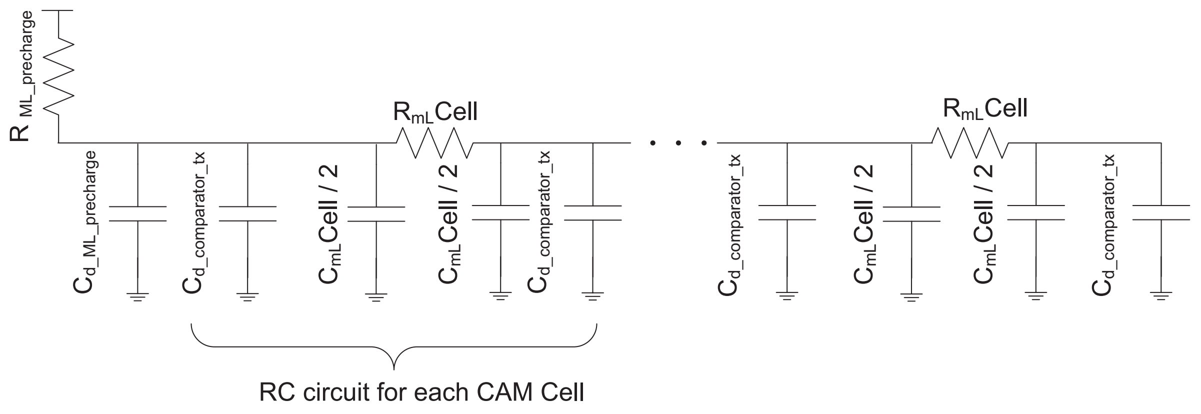
(a) Searchline circuit model in precharge phase



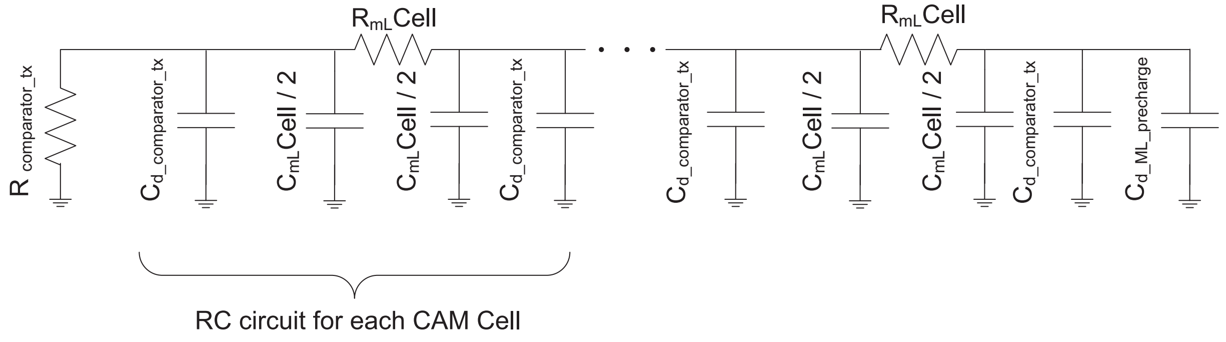
(b) Searchline circuit model in data driven phase with 1 is the bit to be searched

Figure 4.9. Searchline Circuit Model

line precharge circuit, and matchline sense amplifiers. A dummy matchline is used to control the precharge and evaluation timing. This dummy matchline also tracks the PVT (power, voltage, and temperature) variation of the whole CAM array. Figure 4.8 shows the circuit of the dummy matchline. The dummy line does not have the SRAM part in each cell. All but one discharge paths are always off as shown in Figure 4.8. Hence, the dummy line always models the slowest matching operation (one-bit miss). The dummy line signal is used as the signal to shut off the evaluation phase of matchlines. The circuit models of matchlines during precharging and evaluation phases are shown in Figure 4.10, where the matchline latency is in fact the dummy line latency.



(a) Matchline circuit model in precharge phase



(b) Matchline circuit model in evaluation phase, assuming only one CAM cell contains the mismatched bit

Figure 4.10. Matchline Circuit Model

Since matchline delay is on the critical path of overall latency, it is important to reduce it. High performance CAM designs have often used segmented matchlines so that multiple segments of the matchlines can perform comparisons simultaneously and the final result is determined by combining the results of all segments. Although this can reduce the matchline latency effectively, a heavily segmented matchline can reduce the density of the CAM array. Furthermore, the multi-level NAND tree used to combine results from all segments will also add

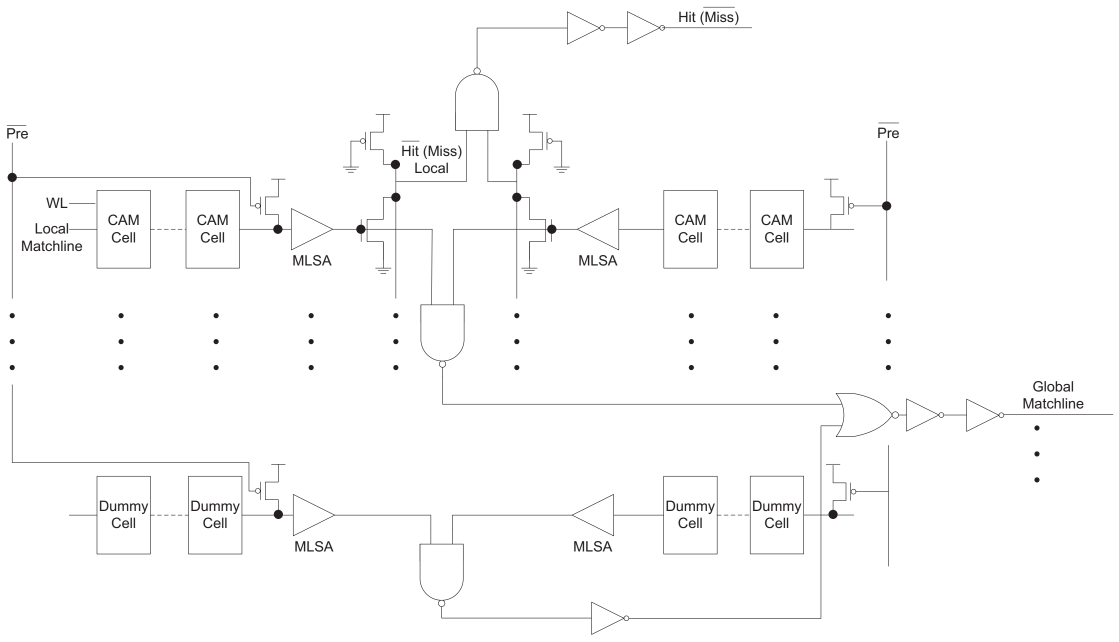
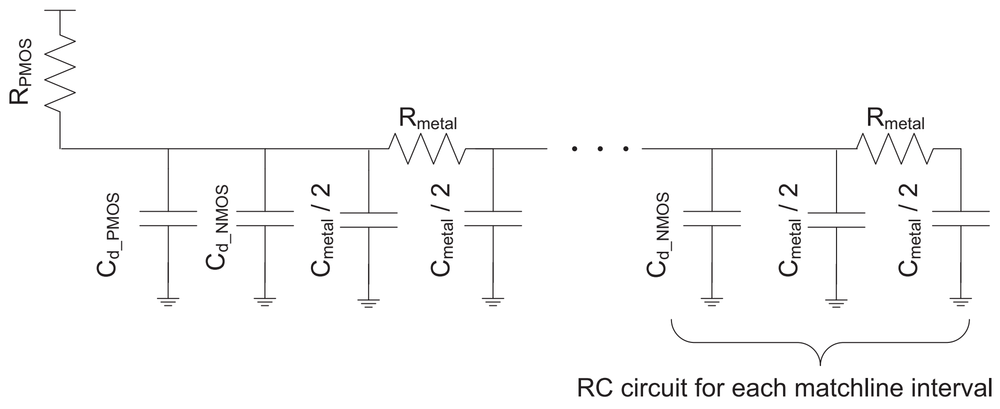


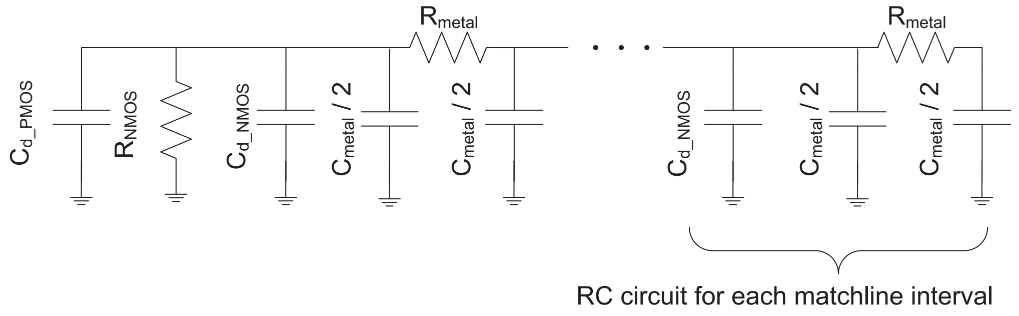
Figure 4.11. CAM with two-segmented Matchlines

latency to the critical path delay. Therefore, current implementations from Intel and Sun [34, 89, 90] only have two segments per matchline. McPAT follows this assumption in its CAM model. The two-segmented matchline model used in McPAT is shown in Figure 4.11, where a matchline becomes two local matchlines that are connected by a NAND gate. The two local matchlines perform comparison simultaneously, which reduces the total matchline latency by approximately half.

Figure 4.8 also shows the hit/miss detection circuit of a CAM. The hit/miss detection circuit contains a weak PMOS transistor (therefore with large on-resistance), that always connects to Vdd. Multiple strong NMOS are controlled by corresponding matchlines and behave as a pull down network. Hits on matchlines open pull down paths of the hit/miss detection circuit and generate a hit signal. If all matchlines are at ground level (all matchlines miss), the hit/miss circuit is charged high



(a) Hit/Miss Detection Circuit in Precharge Phase



(b) Hit/Miss Detection Circuit in Evaluation Phase, assuming the match happens on the matchline with lowest address

Figure 4.12. Hit/Miss Detection Circuit Model

through the weak PMOS, which indicates a miss for the whole subarray. Figure 4.12 shows the circuit model of hit/miss detection circuit.

4.5.2 Power, Area, and Timing Models of CAM

Using the modeling methodology stated in Chapter 2, run time power consumption of a CAM array in McPAT is modeled as shown in Equation 4.9.

$$P_{\text{read/write/search}} = \frac{E_{read/write/search-per-access} * Total\_accesses}{T_{execution}} + P_{leakage} \tag{4.9}$$

where $E_{read/write/search-per-access}$ is the dynamic energy per read/write/search access of the array, $T_{execution}$ is the measured time period, $Total\_accesses$ is the total access in $T_{execution}$, and $P_{leakage}$ is the leakage power of the CAM array. Dynamic energy per access and leakage power are critical to compute final power. $T_{execution}$ and $Total\_accesses$ are passed from performance simulators to McPAT through the XML interface.

The dynamic energy per search access consumed in a CAM array is the sum of the dynamic energy consumed in all active circuit parts involved in the search access, with three major components: the energy consumed in the input network for applying search data, the energy consumed in all active mats/subarrays, and the energy consumed in the output network. Since McPAT models multibanked CAMs, we assume only one bank of a CAM structure is active in a multi-banked CAM during a search access. Equations 4.11 to 4.16 are used to compute dynamic energy per search access per search port.

$$\begin{aligned}
E_{search} &= E_{search-Htree-outside-bank} + E_{search-Htree-inside-bank} + \\
&\quad E_{search-subarrays} * N_{number-subarrays-in-bank} + \\
&\quad E_{results-Htree-inside-bank} + \\
&\quad E_{results-Htree-outside-bank} \qquad (4.10) \\
E_{search-subarrays} &= E_{searchline-precharge-drivers} + \\
&\quad E_{matchline-precharge-drivers} \\
&\quad E_{searchlines} + E_{matchlines} + E_{pri-encoder} \qquad (4.11)
\end{aligned}$$

$$\begin{aligned}
E_{searchlines-in-subbarray} &= (E_{searchline-drivers} + E_{searchline}) * \\
&\quad N_{row-per-subarray} \qquad (4.12) \\
E_{searchline} &= C_{searchline} * V_{DD}^2 \qquad (4.13) \\
E_{matchlines-in-subbarray} &= E_{matchline} * N_{column-per-subarray} \qquad (4.14) \\
E_{matchline} &= C_{matchline} * V_{matchline-swing} * V_{DD} + \\
&\quad E_{NAND2} + E_{MLSA} + \\
&\quad E_{matchline-driver} + \\
&\quad C_{hit-miss-detection} * V_{DD}^2 \qquad (4.15) \\
V_{matchline-swing} &= 2V_{MLSA} \qquad (4.16)
\end{aligned}$$

$C_{searchline}$, $C_{matchline}$, and $C_{hit-miss-detection}$ are the total capacitance of a searchline, a matchline, and the hit-miss-detection circuit as shown in Figure 4.9, Figure 4.10, and Figure 4.12 , respectivly. Equation 4.14 assumes that a rail-to-rail voltage swing needs to be used in searchlines. Low-swing voltage can be used in matchlines as shown in 4.16, and we assume the voltage swing in the matchline rises up to twice the signal that can be detected by the sense amplifier.

Both subthreshold leakage and gate leakage are modeled using the modeling methodology stated in Chapter 2. While we assume ports are independent when calculating dynamic energy, leakage power is computed for all ports of a CAM array. Thus, idle ports still consume leakage power.

At the whole array level, the area of a CAM is estimated based on the area occupied by all banks and the area reserved for I/O networks. The area occupied by I/O networks is determined by the wiring pitch of the network, I/O data widths, and number of banks. We assume that Htees are used for both inter-bank I/O networks and intra-bank I/O networks. The area of a CAM array is modeled using Equations 4.18 to 4.21. Equations 4.20 and 4.21 calculate the overhead caused by routing inter-bank I/O networks and assume that the number of banks in the horizontal direction is always either equal to or twice the number of banks in the vertical direction. The inter-bank Htree is assumed to enter the array horizontally.

$$A_{cam-arr} = H_{cam-arr} * W_{cam-arr} \tag{4.17}$$

$$H_{cam-arr} = N_{banks} * H_{Bank} + H_{Inter-bank-Networks} \tag{4.18}$$

$$W_{cam-arr} = N_{banks} * W_{Bank} + W_{Inter-bank-Networks} \tag{4.19}$$

$$H_{Inter-bank-Networks} = \sum_{n=1}^{\lceil \log N_{banks}/2 \rceil} \frac{Wiring\_Pitch * Total\_bits}{2n} \tag{4.20}$$

$$W_{Inter-bank-Networks} = \sum_{n=1}^{\log N_{banks} - \lceil \log N_{banks}/2 \rceil} \frac{Wiring\_Pitch * Total\_bits}{2n} \tag{4.21}$$

As described earlier in this section, each CAM bank has multiple mats connected by intra-bank htree I/O networks. The area of a single bank is calculated in a similar way as in Equations 4.18 to 4.21. It is worth noting that I/O of a RAM bank consists I/Os of multiple mats in it, while the I/O of a CAM bank always equal to that of a single mat.

The height and width of a mat are estimated using same methodology as in CACTI with the overhead of circuits for search operations being added. The equations for mat area estimation are reproduced here with CAM related changes.

$$W_{mat} = \frac{H_{mat}W_{mat-initial} + A_{mat-center-circuitry}}{W_{initial-mat}} \tag{4.22}$$

$$H_{mat} = 2H_{subarr-cam-cell-area} + H_{mat-non-cell-area} \tag{4.23}$$

$$W_{initial-mat} = 2W_{subarr-cam-cell-area} + W_{mat-non-cell-area} \tag{4.24}$$

$$A_{mat-center-circuitry} = A_{row-predec-block-1} + A_{row-predec-block-2}$$

$$H_{subarr-cam-cell-area} = (N_{subarr-rows} + 0.5) * H_{cam-cell} \tag{4.25}$$

$$\begin{aligned} W_{subarr-cam-cell-area} = {} & N_{subarr-cols}W_{cem-cell} + \\ & \lfloor \frac{N_{subarr-cols}}{N_{mem-cells-per-wordline-stitch}} \rfloor W_{wordline-stitch} + \\ & \lceil \frac{N_{subarr-cols}}{N_{bits-per-ecc-bit}} \rceil W_{cam-cell} + \\ & W_{two-segment-connector} \end{aligned} \tag{4.26}$$

$$\begin{aligned} H_{mat-non-cell-area} = {} & 2H_{subarr-bitline-peri-circ} + 2H_{subarr-searchline-peri-circ} + \\ & H_{hor-wires-within-mat} \end{aligned} \tag{4.27}$$

$$
\begin{aligned}
H_{hor-wires-within-mat} &= \frac{H_{number-mat-addr-bits}}{2} + \frac{H_{mat-write-datain-bits}}{2} + \\
&\quad \frac{H_{mat-read-dataout-bits}}{2} + \\
&\quad \frac{H_{mat-search-datain-bits}}{2} + \\
&\quad \frac{H_{mat-search-dataout-bits}}{2} \qquad (4.28) \\
W_{mat-non-cell-area} &= \max(2W_{subarr-row-decoder}, W_{row-predec-out-wires}) \\
& \qquad (4.29) \\
H_{subarr-bitline-peri-cir} &= H_{bitline-pre-eq} + H_{write-driver} \qquad (4.30) \\
H_{subarr-searchline-peri-cir} &= H_{searchline-pre-eq} + H_{search-driver} \qquad (4.31)
\end{aligned}
$$

As shown in equation 4.29, we assume that two metal planes are used for signal routing within a mat, with each metal plane takes half of the address, read/write data, and search data signals. The areas of lower-level circuit blocks such as the search and write drivers are calculated using the gate area model as in CACTI while taking into account pitch-matching constraints.

Although McPAT models read, write, and search accesses, search accesses are crucial to a CAM structure. We describe the modeling of search access time to a CAM structure in this section, and access times of read or write accesses to a CAM array are computed similarly to a RAM array[1]. Access time of a search operation in a CAM is computed using Equations 4.33 4.37. Search access time is defined as the time spent from search word being applied onto the input network to the search results being sent out via the output network. It is important to note that

[1]Therefore, the variables in following timing equations for access time/latency and cycle time refer to the timing of a search opeartion unless denoted otherwise.

circuits are assumed to be ready (properly precharged), when computing latency.

$$\begin{aligned}
T_{Access} &= T_{In-network} + T_{subarray} + \\
&\quad T_{Out-network} \qquad (4.32)\\
T_{In-network} &= T_{arr-edge-to-bank-edge-htree} + \\
&\quad T_{bank-edge-to-mat-htree} + \\
&\quad T_{mat-edge-to-subarray-network} \qquad (4.33)\\
T_{subarray} &= T_{searchline} + T_{matchline} + T_{MLSA} + \\
&\quad T_{matchline-driver} + \\
&\quad \max(T_{hit-miss-detection}, T_{pri-encoder}) \qquad (4.34)\\
T_{Out-network} &= T_{subarray-to-mat-edge-network} + \\
&\quad T_{mat-to-bank-edge-htree} + \\
&\quad T_{bank-edge-to-arr-edge-to} \qquad (4.35)
\end{aligned}$$

Unlike read and write accesses that involve predecoders in the center of a mat, search accesses pass search data tokens from the mat edge to the subarray edge. The input and output network delays include the delay on network between mat edge and mat edge as shown in Equation 4.34 and 4.35. Latency of simple circuits such as drivers are computed using the static gate delay models as in[40]. In order to compute latencies of searchline ($T_{searchline}$), matchline($T_{matchline}$), and hit-miss-detection logic ($T_{hit-miss-detection}$), we first compute time constants of the three circuit structures ($\tau_{searchline}$, $\tau_{matchline}$, and $\tau_{hit-miss-detection}$) respectively using the Elmore delay with the RC-tree topologies of the working phase as shown in Figure 4.9 (b), 4.10 (b), and 4.12 (b). Then, the latencies of the three circuit structures with step inputs can be computed using Equation 4.36. Finally, the latencies with slow input of the three circuit structures are computed using Equation 4.37 based on the model described in [107]. The model considers the effect of the rise time of the signal in previous stage by considering the slope (m) of the signal.

$$T_{step} = \tau_{working-phase} * ln(\frac{V_{start}}{V_{end}}) \tag{4.36}$$

$$T_{with\_slow\_input} = \begin{cases} \sqrt{2T_{step}\frac{VDD-V_{TH}}{m}} & if\ T_{step} <= 0.5\frac{VDD-V_{TH}}{m} \\ T_{step} + \frac{VDD-V_{TH}}{2m} & if\ T_{step} > 0.5\frac{VDD-V_{TH}}{m} \end{cases} \tag{4.37}$$

Cycle time determines the achievable clock frequency of a CAM. We have chosen to model conventional structure with precharge phase, where cycle time is the sum of latencies of all non-pipelinable operations plus the sum of all non-overlap precharge time. Thus, inside a CAM subarray where major operations are non-pipelineable, the cycle time is usually a modest percent larger than the access time because of extra precharge time. However, for a large capacity CAM where the latency of I/O networks dominates the overall CAM latency, the cycle time is smaller than the access time. This is because the I/O networks can be heavily pipelined. Therefore, although they consume significant portion of access latency, their portions of cycle time are negligible. The cycle time of a CAM (for search access) is computed using Equations 4.38 to 4.39.

$$\begin{aligned} T_{cycle-time} = {} & T_{Access} - T_{In-network} - T_{Out-network} + \\ & T_{Searchline-precharge} + T_{matchline-precharge} + \\ & T_{hit-miss-detection-reset} \end{aligned} \tag{4.38}$$

Since the precharge circuits of searchlines, matchlines, and hit-miss-detection logic are very simple, we assume that the prechage time of the three circuits is the step-input response time. Therefore, the precharge time is computed using Equations 4.39, where T_{step} is the precharge time of searchline, matchline, or hit-miss-detection logic, and $\tau_{precharge-phase}$ is the correspondent time constant of the circuit structure computed using Elmore delay with the RC-tree topologies of the precharge phase as shown in Figure 4.9 (a), 4.10 (a), and 4.12 (a).

$$T_{step} = \tau_{precharge-phase} * ln(\frac{V_{start}}{V_{end}}) \tag{4.39}$$

4.5.3 Multiported CAM Model

Multiported CAMs are very important for modeling processors and key components of modern processors including renaming units, load-and-store units and instruction schedulers. McPAT models a multiported CAM based on a single-ported CAM. Each search port requires a set of searchlines and matchlines. The extra search ports are modeled as an increase to CAM cell size, which results in longer searchline and matchline lengths. Each search port needs a set of auxiliary circuits including sense amplifiers, precharge circuits, drivers, etc. These auxiliary circuits are duplicated for each search ports. Although the duplicated auxiliary circuits have an impact on area and leakage power, they do not affect timing and dynamic energy per access of the CAM. (The total dynamic energy will double if two search ports are active simultaneously).

Figure 6 demonstrates a four port CAM cell configuration. It consists of a read/write port (BL0) and three search ports (SL0, SL1, and SL2). The read/write port (BL0) and search port 0 (SL0) multiplex bitlines and searchlines. The impact

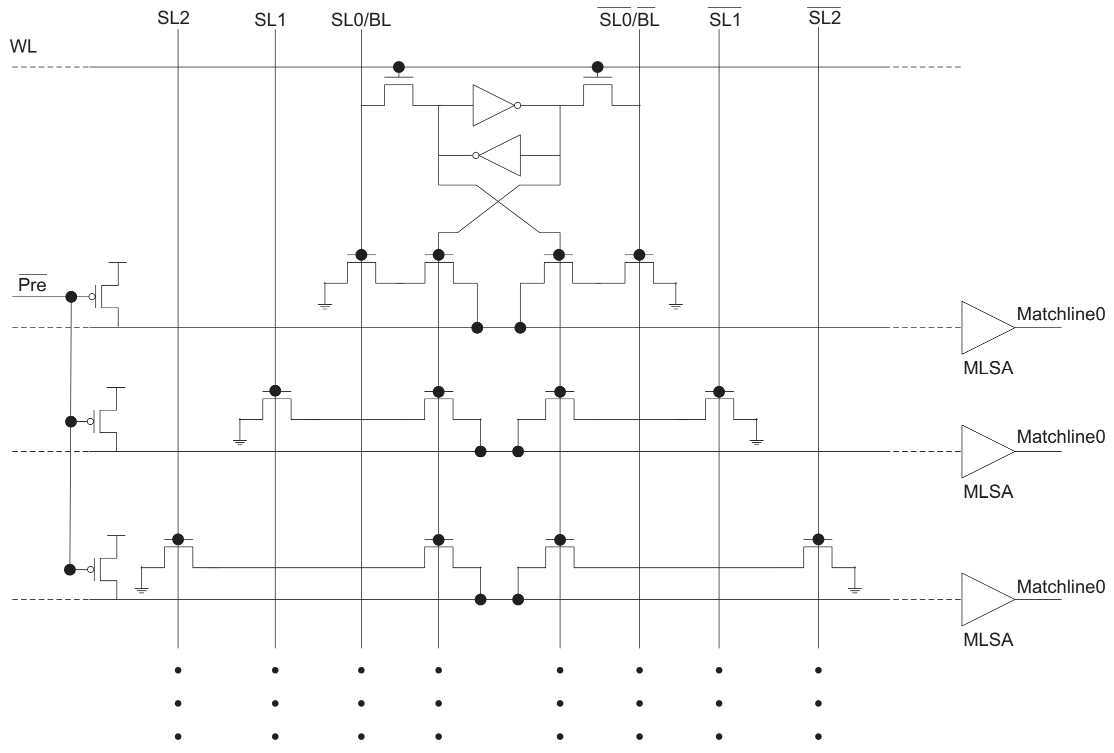


Figure 4.13: Multiple port CAM cell example. BL0 is a read/write port. SL0 is a search port that shares circuits with the read/write port. SL 1 and 2 are search only ports. Each additional search port impacts the cell size and wire lengths of searchlines and matchlines

of extra search ports on cell size is as follows:

If the extra search port multiplexes with a read/write or write port:

- increase the CAM cell size by 2 wire pitches in the y direction (affects searchline/bitline metal)
- do not increase the CAM cell size in the x direction (assuming the overhead caused by read/write or write port is already considered)

If the extra search port is exclusively used for searching:

  - increase CAM cell by 2 wire pitches in both the x and y directions (affects both searchline/bitline and matchline/wordline metal).

4.6 Technology Level Modeling

Following the same methodology as in CACTI5[100], McPAT uses MASTAR [88] to derive device parameters from the ITRS roadmap [88] and R. Ho's work [38, 39] to derive wire parameters for different technology nodes. The current implementation of McPAT includes data for the 90nm, 65nm, 45nm, 32nm, and 22nm technology nodes, which covers the ITRS roadmap through 2016. The ITRS assumes that planar bulk CMOS devices will reach practical scaling limits at 36nm, at which point the technology will switch to silicon-on-insulator (SOI). Below 25nm, the ITRS predicts that SOI will reach its limits and that double-gate devices will be the only option. McPAT captures each of these options, making it scalable with the ITRS roadmap. McPAT shares the same technology level modeling with CACTI 5. The main improvements at technology level modeling are: (1) the technology generations have been extended to 22nm with double gate (DG) technology; (2) gate leakage is modeled based on MASTAR; (3) the interconnect also has been extended to 22nm technology; (4) and double/triple/quad- width wires are added into the interconnect technology model.

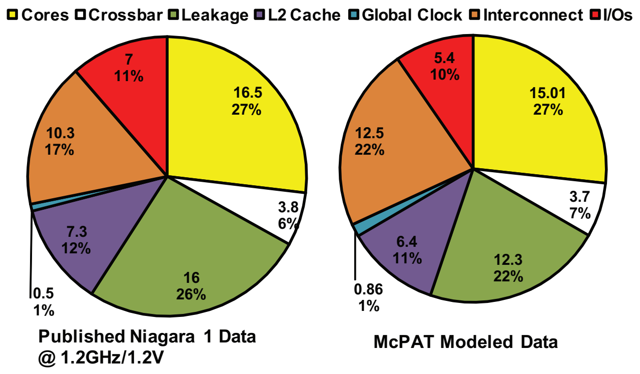
CHAPTER 5

MCPAT VALIDATION

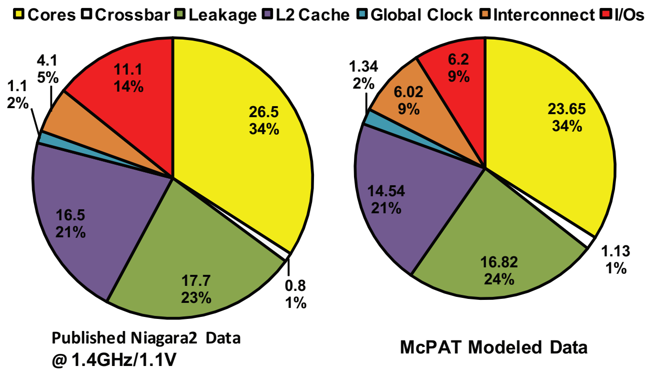
The main focus of McPAT is accurate power and area modeling at the architectural level when timing is given as a main design constraint. Both *relative* and *absolute* accuracy are important for architectural level power modeling. Relative accuracy means that changes in modeled power as a result of architecture modifications should reflect on a relative scale the changes one would see in a real design. While relative accuracy is critical, absolute accuracy is also important if one wants to compare results against thermal design power (TDP) limits, or to put core power savings in the context of the whole processor or whole system.

We compare the output of McPAT against published data for the 90nm Niagara processor [58] running at 1.2GHz with a 1.2V power supply, the 65nm Niagara2 processor [70] running at 1.4GHz with a 1.1V power supply, the 65nm Xeon processor [83] running at 3.4GHz with a 1.25V power supply, and the 180nm Alpha 21364 processor [43] running at 1.2GHz with a 1.5V power supply. There are in-order and out-of-order processors as well as single-threaded and multithreaded processors in the validation targets. Therefore, the validations stress McPAT in a comprehensive and detailed way. The comparisons against Niagara and Niagara2 processors also test our ability to cross technology generations and retain accuracy. The configurations for the validations are based on published data on the target processors in [43, 58, 70, 83, 97], including target clock rate, working temperature, and architectural parameters. The target clock rate is used as the lower bound for the timing constraint in McPAT to determine the basic circuit properties and must be satisfied before other optimizations and trade-offs can be applied. Therefore, the generated results must match the clock rate of the actual target processor. Because timing (target clock rate) is already considered when computing and optimizing power and area, only power and area validation results are shown in this section.

Figure 5.1 and 5.2 shows the validation results. Unfortunately, the best power numbers we have for these processors are for peak power rather than average power. Fortunately, McPAT can also output peak power numbers based on maximum activity factors and maximum switching activity. These results show that modeled power numbers track the published numbers well. For the Niagara processor, the absolute power numbers for cores and crossbars generated by McPAT match very well with the published data. Over all seven components, the average difference in absolute power between the modeled power numbers and published Niagara data is just 1.47W, for an average error per component of 23%. That number seems high, but the two big contributors are clock power (72% error, but a small contributor to total power) and leakage power (23% error). Both are significantly more accurate for the Niagara2. For Niagara2, the average error is 1.87W (26%), but by far the biggest contributor (4.9W error) is the I/O power. This arises because Niagara2 is a complicated SOC with different types of I/Os including memory, PCI-e, and 10Gb Ethernet [70] while McPAT only models on-chip memory controllers/channels as I/Os. If we were measuring average power instead of peak power, this difference would shrink given the expected activity factors of those components. The modeled power number of the OOO issue logic,



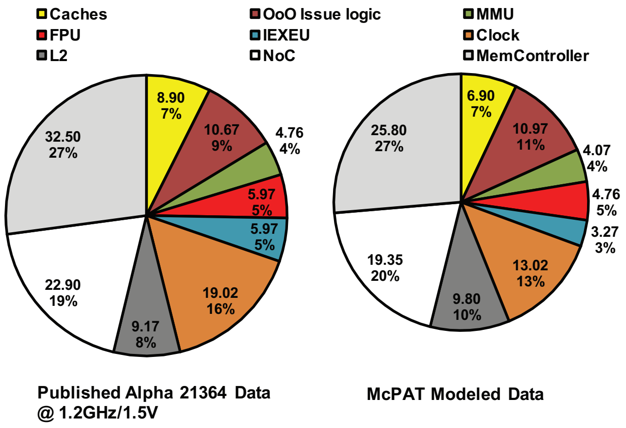
(a) Validation against Niagara processor.



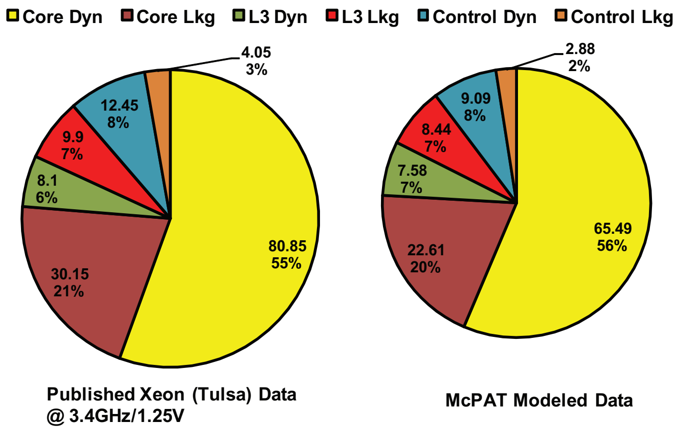
(b) Validation against Niagara2 processor.

Figure 5.1: McPAT validation against niagara processors. The numbers in all charts are the reported and modeled power numbers of the components. The percentages denote the ratios of the component power to the total power. There are miscellaneous components such as SoC logic and I/O that McPAT does not model in detail because their structures are unknown. Therefore, 61.4W out of the total 63W are modeled for Niagara, 77.9W out of the total 84W are modeled for Niagara2.

key component of the OOO core in the Alpha 21364 processor, is very close to the reported power with only a 2.78% difference. Although there are no detailed power breakdowns of both the core and uncore parts of Xeon Tulsa, the modeled bulk power of core and uncore comes close to reported data, with -20.63% and -11% error respectively.



(a) Validation against Alpha 21364 processor.



(b) Validation against Xeon Tulsa processor.

Figure 5.2: McPAT validation against OOO procesors. 119.8W out of the total 125W are modeled for Alpha 21364, and 145.5W out of the total 150W are modeled for Xeon Tulsa.

Table 5.1 shows the comparison of total power for validations against the target processors. Differences between the total peak power generated by McPAT and reported data are 10.84%, 17.02%, 21.68%, and 22.61% for Niagara, Niagara2, Alpha 21364, and Xeon Tulsa, respectively. It is worth noting that these differences include the unknown parts that we do not model in detail. Chip-to-chip power variation in recent microprocessor designs [15] is also comparable to the errors reported in Table 5.1.

TABLE 5.1

VALIDATION RESULTS OF MCPAT WITH REGARD TO TOTAL POWER OF TARGET PROCESSORS.

Processor	Published total power	Modeled total power	McPAT results	% McPAT error
Niagara	63W	61.4W	56.17W	-10.84
Niagara2	84W	77.9W	69.70W	-17.02
Alpha 21364	125W	119.8W	97.9W	-21.68
Xeon Tulsa	150W	145.5W	116.08W	-22.61

TABLE 5.2

VALIDATION RESULTS OF MCPAT WITH REGARD TO CHIP DIE AREA OF TARGET PROCESSORS.

Processor	Published die size (mm^2)	McPAT results (mm^2)	McPAT error %
Niagara	378	295	-21.8
Niagara2	342	248	-27.3
Alpha 21364	396	324	-18.2
Xeon Tulsa	435	362	-16.7

Table 5.2 compares the published die sizes of the target processors with the McPAT results, which shows that the modeled area numbers track the published numbers well. The area error is higher for Niagara2 because it has more types of I/O components than others as mentioned above, which are not modeled in McPAT. It is important to point out that even if we use empirical models for highly irregular logic, we still see a reasonable match between McPAT results and

reported data. The area validation of Alpha is a good example. Although we do not use Alpha data when building our empirical models, the area validation of Alpha 21364 is only off 18.2% from the reported data. This shows that both the analytical models and empirical models for area modeling have good accuracy. Given the generic nature of McPAT, we consider these power and area errors acceptable.

CHAPTER 6

SCALING TRENDS AND CLUSTERING TRADE-OFFS OF FUTURE MANYCORE PROCESSORS

As technology keeps scaling, the same die area can accommodate more and more cores. There are two important questions that need to be answered: (1) how new device and interconnect technologies can help to shape future architectures; (2) how cores can be organized efficiently when the total core count is large. In order to answer these two questions, we use McPAT together with a performance simulator to evaluate the following aspects of a manycore processor designed for throughput computing: (1) the scaling trends of power, area, and timing of the proposed manycore architecture, and (2) the benefits of organizing cores into clusters with local interconnect. We evaluate this architecture across five technologies—90, 65, 45, 32, and 22nm—which covers years 2004 to 2016 of the ITRS roadmap. Clustering will bring interesting tradeoffs between area and performance because the interconnects needed to group cores into clusters incur area overhead, but many applications can make good use of them due to synergies of cache sharing.

This chapter also introduces the metrics of energy-delay-area^2 product (EDA^{2}P) and energy-delay-area product (EDAP) as examples of comprehensive metrics that include both performance and cost, the operational cost (energy) and the capital cost (area). While a chip vendor may favor EDA^{2}P as *area*^{2} provides an approximation to die cost in practice [35], a system vendor could prefer EDAP as other fixed system costs such as memory and I/O reduce the overall system cost dependence on chip multiprocessor cost. As shown in this chapter, the new metrics are proved to be able to reveal new design sweet spots that cannot otherwise be found using current metrics.

6.1 Proposed Architecure

Figure 6.1 shows the manycore architecture we assume targeting future high throughput computing. It consists of multiple clusters connected by a 2D-mesh on-chip network. A cluster has one or more multithreaded Niagara-like [51] cores and a multi-banked L2 cache. Each core has up to 4 active threads and 32KB 4-way set-associative L1 instruction and data caches. All cores in a cluster share a multi-banked 16-way set-associative L2 cache. The number of L2 banks equals the number of cores per cluster. The size of an L2 bank is 256KB. All caches have 64B cache lines. A crossbar is used to connect cores and L2 cache banks for intra-cluster communications. A two-level hierarchical directory-based MESI protocol is used for cache coherency. Within a cluster, the L2 cache is inclusive and filters coherency traffic between L1 caches and directories. Between clusters, a cache directory is implemented by using directory caches that are associated with the on-chip memory controllers, similar to the implementation in the Alpha 21364 processor. The 2D-mesh networks have a data width of 256 bits. We use minimal dimension-order routing and two virtual channels per physical port. Each virtual channel has a 32-deep flit input buffer. Double-pumped crossbars [103] are used in the routers to reduce the die area. Routers in the networks have a local port that connects the hub of a cluster as well as ports that connect the neighboring routers.

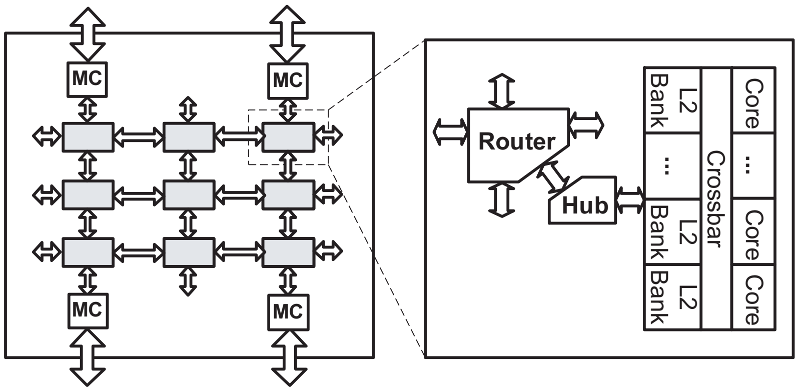


Figure 6.1: Manycore system architecture. MCs refer to memory controllers.

TABLE 6.1

PARAMETERS OF THE MANYCORE ARCHITECTURE ACROSS TECHNOLOGY GENERATIONS. EACH MEMORY CONTROLLER HAS ONE MEMORY CHANNEL.

Parameters	90nm	65nm	45nm	32nm	22nm
Clock rate (GHz)	2.0	2.3	2.7	3.0	3.5
The number of cores	4	8	16	32	64
The number of memory controllers	2	3	4	6	8
Memory capacity per channel (GB)	2	4	4	8	8
Main memory type	DDR2-667	DDR3-800	DDR3-1066	DDR3-1333	DDR3-1600

Table 6.1 shows the parameters of the manycore architecture at each technology generation. We start from a conservative 2.0GHz clock rate at 90nm technology, which is about the average of that of the Niagara processor and Intel processors fabricated in 90nm processes. The intrinsic speed of high-performance

transistors increases by 17% per year according to the ITRS. However, increasing clock frequency at this pace will lead to unmanageable chip power density. Moreover, unlike Intel's approach of changing micro-architectures of their processors during technology scaling, we increase the number of cores and memory controllers aggressively, while keeping the same micro-architecture for new generations. Therefore, we increase the clock frequency conservatively by around 15% every generation. We also start from a conservative die size around 200mm^2 at 90nm technology and use McPAT to optimize power, area, and timing. The results show that four cores can be placed within the specified area at 90nm. Then, we double the number of cores for each generation. It is difficult to increase the number of memory controllers and channels linearly with the increased core count because of the limited pin count of the chip [88]. We assume that the number of memory controllers grows proportional to the square root of the cluster count since the bandwidth of each controller also increases over time. The memory channels are shared by all clusters through the on-chip network and placed at the edge of the chip to minimize the routing overhead as shown in Figure 6.1. As shown in Table 6.1, we also scale the bandwidth of main memory based on the expected availability of major DIMM products at each technology node.

6.2 Experimental Setup

We developed a manycore simulation infrastructure where a timing simulator and a functional simulator are decoupled, as in GEMS [63]. We modified a user-level thread library [77] in the Pin [60] binary instrumentation tool to support more pthread APIs, and use it as a functional simulator to run applications. In-order

TABLE 6.2

SPLASH-2 DATASETS.

Application	Dataset	Application	Dataset
Barnes	16K particles	Cholesky	tk17.O
FFT	1024K points	Radiosity	room
FMM	16K particles	Raytrace	car
LU	512×512 matrix	Volrend	head
Ocean	258×258 grids		
Radix	8M integers		
Water-Sp	4K molecules		

cores, caches, directories, on-chip networks, and memory channels are modeled in an event-driven timing simulator, which controls the execution flow of a program running in the functional simulator and effectively operates as a thread scheduler.

SPLASH-2 [108], PARSEC [12], and SPEC CPU2006 [36] benchmark suites are used for the experiments. The number of threads spawned in a multithreaded workload is the same as the number of hardware threads so each thread is statically mapped to a hardware thread. We use all SPLASH-2 applications and 5 of the PARSEC applications (only canneal, streamcluster, blackscholes, fluidanimate, and swaptions currently run on our infrastructure.) The simlarge dataset is used for PARSEC applications while the datasets used for SPLASH-2 applications are summarized in Table 6.2. For each application, the same dataset is used throughout all process generations. We use the SPEC CPU2006 benchmark suite to measure the system performance on consolidated workloads. The bench-

TABLE 6.3

SPEC 2006 APPLICATION MIXES FOR HIGH, MED, AND LOW MEMORY BANDWIDTH.

Set	Applications
CINT	
high	429.mcf, 462.libquantum, 471.omnetpp, 473.astar
med	403.gcc, 445.gobmk, 464.h264ref, 483.xalancbmk
low	400.perlbench, 401.bzip2, 456.hmmer, 458.sjeng
CFP	
high	433.milc, 450.soplex, 459.GemsFDTD, 470.lbm
med	410.bwaves, 434.zeusmp, 437.leslie3d, 481.wrf
low	436.cactusADM, 447.dealII, 454.calculix, 482.sphinx3

mark suite consists of integer (CINT) and floating-point (CFP) benchmarks, all of which are single-threaded. We pick 12 applications from both CINT and CFP, and make 3 groups each, 4 applications per group, based on their main-memory bandwidth demand [36] as shown in Table 6.3. We find the representative simulation phases of each application and their weights using Simpoint 3.0 [92]. Each hardware thread runs a simulation phase and the number of instances per phase is proportional to its weight. We skip the initialization phases of each workload and simulate 2 billion instructions unless it finishes earlier.

Table 6.4 shows the area and maximum power of the proposed architecture with four cores per cluster across five technology generations. Core area varies from 43% to 62% of total die size across technologies. Core area scales worse than uncore area since uncore components, especially L2 caches, crossbars, and routers, are more regular and easier to scale across generations than cores. According to McPAT's area modeling results, the double-pumped crossbars reduce the area of intra-cluster crossbars and 2D mesh routers by 54.1% and 35.6% respectively, compared to the single-pumped crossbar implementations. Although we save significant area on interconnects within and between clusters, uncore components

TABLE 6.4

AREA AND MAXIMUM POWER OF CONFIGURATIONS WITH 4 CORES PER CLUSTER ACROSS TECHNOLOGY GENERATIONS.

	90nm	65nm	45nm	32nm	22nm
Core area (mm^2)	81.9	96.4	113.4	133.5	157.1
Uncore area (mm^2)	104.3	111.3	102.7	101.6	93.5
Die area (mm^2)	186.3	207.7	216.2	235.1	250.6
Max core dynamic power (W)	24.1	30.7	41.7	48.3	56.4
Max uncore dynamic power (W)	20.6	36.1	45.9	54.5	61.8
Total subthreshold leakage (W)	6.5	11.2	17.6	21.5	25.8
Total gate leakage (W)	2.6	6.7	0.7	1.6	2.5
Chip max power (W)	53.8	84.8	106.0	125.9	146.7

still occupy a big portion of the total die area.

Short-circuit power is around 10% of the total dynamic power, with fluctuations within 3.1% across all the technology generations. The main reason for the stable short-circuit power is that we use ITRS technology models that have stable V_{th} to V_{dd} ratios. Cores burn about half of the total maximum dynamic power across generations. Gate leakage is an important component in 90nm and 65nm technology, being 37.6% of the total leakage power at 65nm technology. Hi-k metal gate transistors [9] are introduced at 45nm, which reduces the gate leakage by more than 90%. SOI technology and double gate (DG) devices that are used at 32nm and 22nm technology also help to keep the subthreshold leakage under control.

Table 6.5 shows the die areas of various manycore architectures when the number of cores per cluster is varied from 1 to 8 at the 22nm technology node. Since the total number of cores is fixed at 64, NoC size decreases as the number of cores per cluster increases. We keep the same NoC bisection bandwidth on all configurations. The 1 core per cluster design is the smallest in size because it does not need crossbars between cores and L2 caches. Even though it needs more routers to connect clusters, the size of each router is smaller since we keep the same bisection bandwidth. When the size of a mesh network is not square, its bisection bandwidth is limited by a cut through its smaller dimension. So we need the same link width for 8×4 and 4×4 networks. That is why the die size of the 2 core per cluster design is very close to that of the 4 core per cluster design, even though the latter has much bigger crossbars in its clusters.

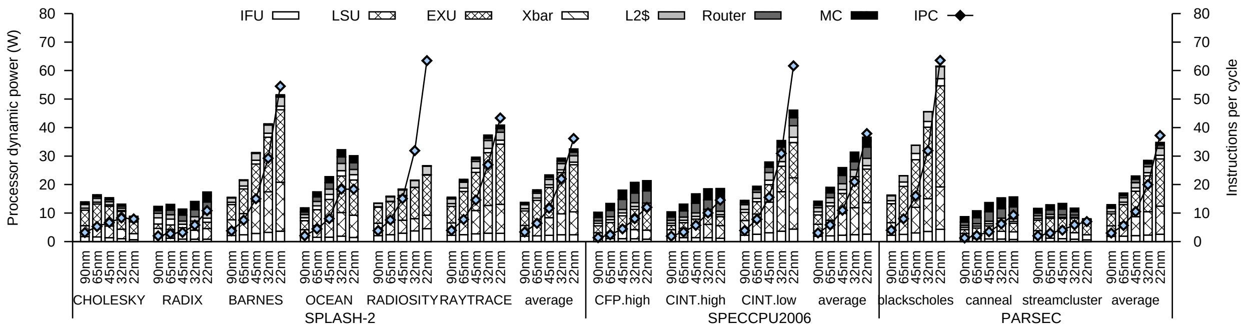
TABLE 6.5

NOC SIZES AND DIE AREAS OF THE MANYCORE ARCHITECTURE AT 22NM. THE NOC BISECTION BANDWIDTH IS KEPT CONSTANT ACROSS CONFIGURATIONS.

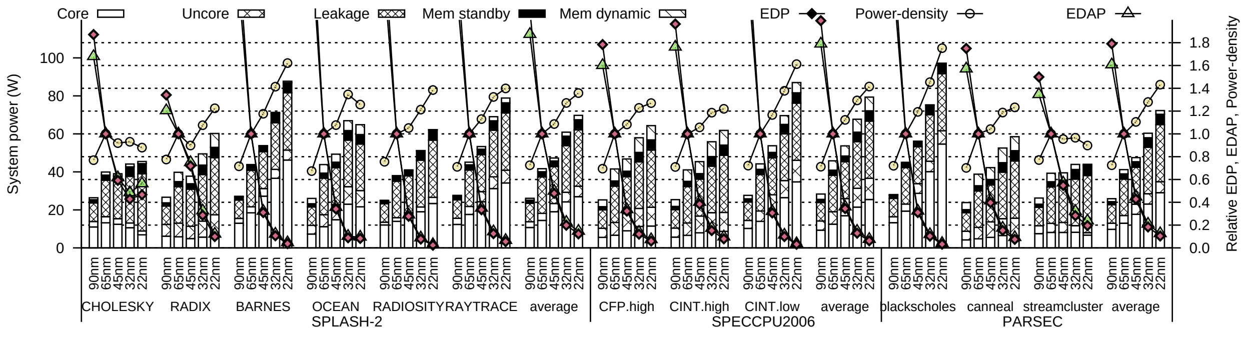
	NoC size	Die size (mm^2)
1 core per cluster	8×8	239.1
2 cores per cluster	4×8	246.3
4 cores per cluster	4×4	250.6
8 cores per cluster	2×4	278.6

6.4 Performance and Efficiency Tradeoffs in Technology Scaling and Clustering

Because McPAT provides an integrated power, area, and timing model, when combined with performance data, chip multiprocessors can be analyzed using several metrics previously unavailable in architecture studies. We think energy-delay-area^2 product (EDA^{2}P) and energy-delay-area product (EDAP) are particularly interesting metrics. These metrics include both an operational cost component (energy) as well as a capital cost component (area). Although the die yield is proportional to the fourth power of the area [79], in practice due to good die yield and when combined with die per wafer, die cost is roughly proportional to the square of the area [35]. So when designing and manufacturing a chip multiprocessor, we believe EDA^{2}P is a good way of including chip cost into the optimization process. However, other fixed system costs such as memory and I/O reduce the overall system cost dependence on chip multiprocessor cost. Thus we believe EDAP could



(a) Processor dynamic power and IPC

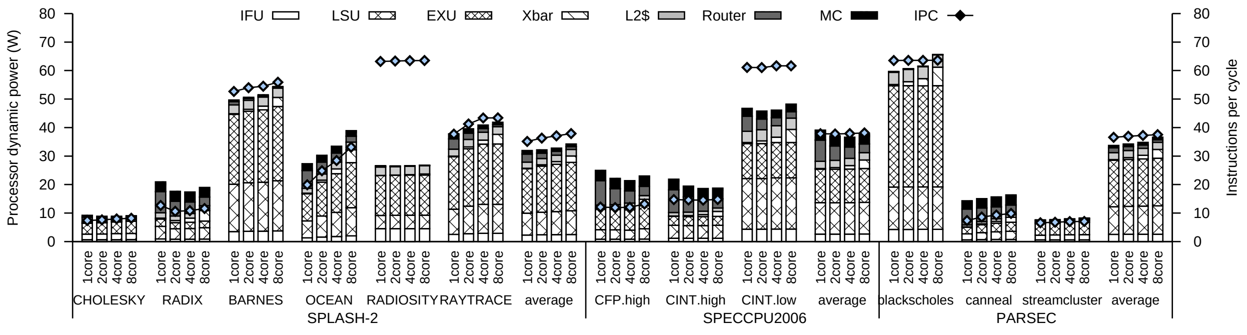


(b) System power, EDP, and EDAP

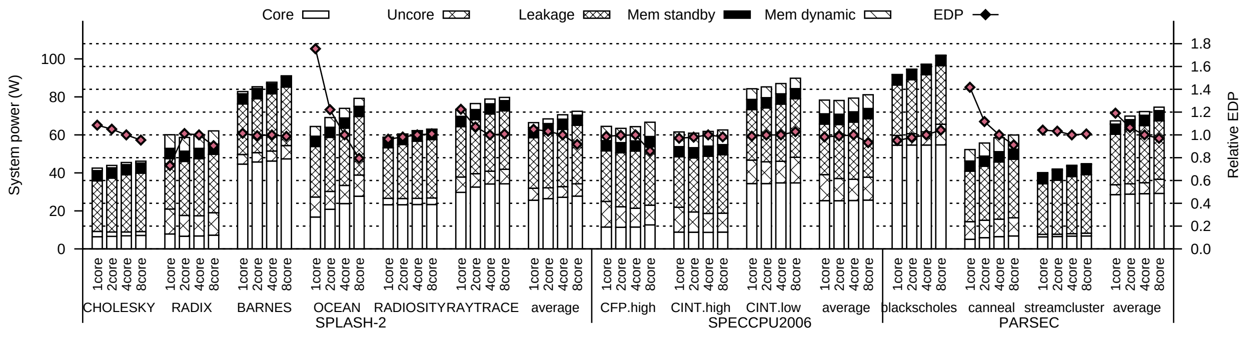
Figure 6.2: Power, power-density, IPC, EDP, and EDAP of the manycore systems while the technology nodes are changed from 90nm to 22nm. For each suite, there are applications which are not listed due to space limitations, but they are included when average values are computed.

be a more useful metric for chip multiprocessors at the system level than EDA^{2}P. Hence, a chip vendor may favor EDA^{2}P, while a system vendor could prefer EDAP. Finally, given McPAT's area models, another interesting metric is power density: chip power divided by area. Cooling a microprocessor becomes substantially more difficult as power density increases, and this can add significant capital cost for more advanced packaging.

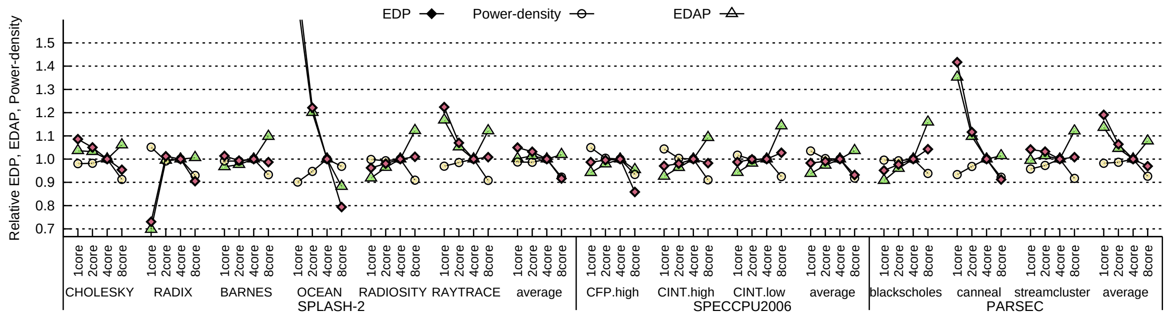
Figure 6.2 shows power, instructions per cycle (IPC), system energy-delay product (EDP), and EDAP of the 5 system configurations where the technology nodes are changed from 90nm to 22nm. These are all 4-core/cluster configura-



(a) Processor dynamic power and IPC



(b) System power and EDP



(c) System EDAP, EDP, and power-density

Figure 6.3: Power, power-density, IPC, EDP, and EDAP of the manycore systems while the number of cores per cluster is changed from 1 to 8 on the 22nm technology node. For each suite, there are applications which are not listed due to space limitations, but they are included when average values are computed.

tions. Applications are grouped by three benchmark suites, and in each group there are applications which are not listed due to space limitations, but they are included when average values are computed. Figure 6.2(a) shows the IPC and the dynamic power of 5 configurations. Applications such as RADIOSITY, CINT.low, and blackscholes are not limited by main memory bandwidth and scale close to linearly with the number of cores in the system. The IPCs of RADIX, CFP.high, CINT.high, and canneal improve relatively slowly since they are limited by main memory bandwidth, which scales worse than computation power. CHOLESKY and OCEAN are the applications with limited IPC scaling because of the insufficient parallelism within applications or small datasets, which in turn leads to the decrease of power density over technology generations. Figure 6.2(b) shows the system power breakdown, the power density, the EDP, and the EDAP of 5 configurations. The power density, the EDP and the EDAP values are normalized to the values of the 65nm configuration. Many of the applications have inflection points in power density at 65 nm because gate leakage is reduced by more than 90% from 65nm to 45nm. On many applications, the EDP values improve rapidly as process technology improves. But on CHOLESKY and OCEAN the EDP stops improving after the 32nm process because of limited IPC scaling. The EDAP values scale in a similar way to the EDP values, but change less since the die area increases as shown in Table 6.4.

In Figure 6.3, the number of cores per cluster is varied from 1 to 8 on the 22nm technology node. The layout of Figure 6.3 is the same as that of Figure 6.2, but here the power density, the EDP, and the EDAP values are normalized to the 4-core per cluster configuration. Figure 6.3(a) shows that throughout the applications, the intra-cluster crossbar power (Xbar) increases and inter-cluster power decreases as the number of cores per cluster increases. This is expected because the size and power of a crossbar scales super-linearly with the number of cores per cluster. The effects of this clustering on IPC depend heavily on applications.

As more cores are grouped into a cluster, the size of the L2 cache that a core can access increases even though more cores share the multi-banked cache, so if multiple cores in a cluster share data (i.e., cores use a shared L2 cache synergistically), then an L2 cache can retain more of the combined working sets, hence lowering L2 misses. On OCEAN, RAYTRACE, and canneal, the IPC increases noticeably because of this synergistic cache sharing effect. As a result, the EDP of these applications improves as the number of cores per cluster increases, as shown in Figure 6.3(b). In contrast, there are applications like RADIOSITY, CINT.low, and blackscholes whose performance is not affected by cache sharing. On these applications, the EDP gets worse as more cores are grouped since the intra-cluster crossbar power increases.

On average, clustering more cores together improves the system energy-delay product. However, if we take the area of the processors into account, the benefits of cache sharing are negated, especially when the number of cores per cluster is 8. In that configuration, the system energy-delay-area product is worse than the configuration with 4 cores per cluster on all benchmark suites on average (Figure 6.3(c)). RADIX and CFP.high are two applications showing interesting performance characteristics, which in turn is reflected in the EDP. On RADIX, when the number of cores per cluster changes from 1 to 2, the L2 miss percentage increases noticeably because cache lines in the L2 caches are evicted more frequently, meaning that cores in a cluster interfere with each other. On CFP.high, cache sharing does not affect performance when the number of cores per cluster is changed from 1 to 4, but the L2 miss rate drops considerably as 8 cores are grouped into a cluster. This is because with fewer cores per cluster, there are noticeable miss rate fluctuations between L2 caches, which are mostly attenuated

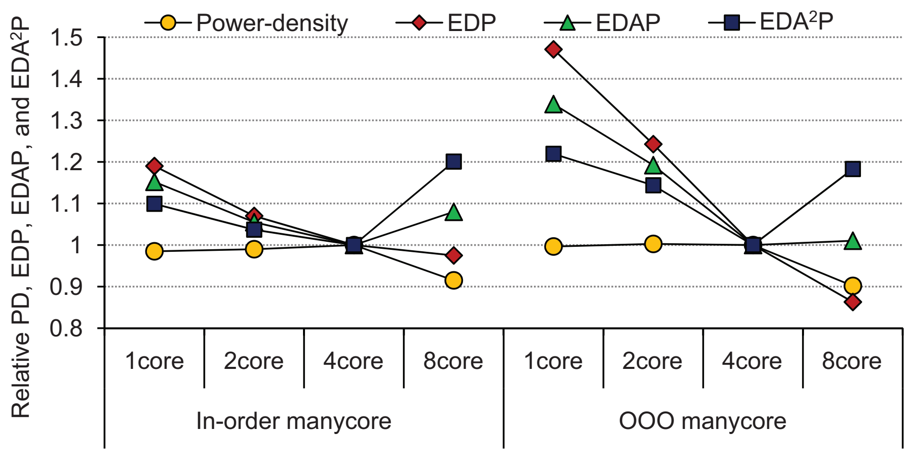


Figure 6.4: Averaged power-density, EDP, EDAP, and EDA^{2}P of both in-order and OOO manycore architectures at the 22nm technology node running PARSEC benchmarks. Total numbers of cores/threads are 64/256 and 16/16 for in-order and OOO processors, respectively. The number of cores per cluster is changed from 1 to 8 for both in-order and OOO processors.

when 8 cores share an L2 cache.

By using McPAT together with M5 [13], another cycle-accurate performance simulator, we also studied clustering trade-offs of manycore processors with 16 OOO cores at 22nm running at 3.5GHz. Each OOO core is similar to the single-threaded four-issue Alpha 21264 processor but with 32KB 4 way set-associative instruction and data caches. The total capacity of on-chip L2 caches of the OOO manycore is as same as the in-order manycore. The L2 caches are shared within a cluster and coherent among different clusters. The OOO manycore architecture (with fewer, larger cores) can provide similar peak performance and occupy similar die area to the previously described in-order manycore architecture. Specifically, both architectures' ideal IPCs are 64, and the area of the OOO manycore is about 3.5% larger than the in-order manycore when using the same number of cores per cluster. The same subset of PARSEC benchmark suite is used for the OOO simulations.

Figure 6.4 shows that clustering trade-offs with OOO cores have similar, but magnified, trends as when using in-order cores. For both the in-order and OOO cores running PARSEC benchmarks, if manycore die cost is not taken into account 8-core clusters provide the best EDP. However, 4-core clusters are best for both in-order and OOO cores when manycore die cost is taken into account by using EDA^{2}P and EDAP.

CHAPTER 7

LIGHTWEIGHT CHIP MULTI-THREADING (LCMT): MAXIMIZING FINE-GRAINED PARALLELISM ON-CHIP

Coarse-grained Multithreaded applications benefit greatly from multithreaded and multicore processors that have already become the mainstream. For example, embarrassingly parallel applications can achieve almost linear speedup as the number of cores increases, when memory starvation is not present. There are many important applications that are readily divided into coarse-grained parallel sections. For these, popular multithreaded programming interfaces such as Pthreads or OpenMP are sufficient. However, because such heavyweight threads accumulate a large memory footprint with complex data structures and require interaction with the operating system (OS), the operations of starting, stopping, suspending, and synchronizing threads incur significant overhead and thus must be used sparingly. As a result, this approach is often ineffective for irregular and dynamically-parallel applications, such as one finds in many graph problems [10, 26, 68, 72, 75] or agent-based simulations [2, 44]. For these applications, the performance of multicore processors is limited because of the lack of architectural support for fine-grained parallelism. Thus, one downside of current multicore processors is that they only provide architectural support for coarse-grained parallel interfaces such as Pthreads[17] or OpenMP[22].

To solve this problem, researchers have proposed software-based, fine-grained multithreaded environments, such as MIT's Cilk[29] and Intel's threading building blocks (TBB) [80]. These environments rely on coarse-grained standard threading libraries, such as Pthreads, where fine-grained parallel threads (called as procedures in Cilk and tasks in TBB) become work units in the work queues of the Pthreads. When the work queues of the Pthreads become unbalanced, these environments use work stealing to distribute the workload.

While the software-based, fine-grained multithreaded environments have demonstrated superior performance over heavyweight, OS-level threads for irregular applications, software-based lightweight multithreading has its limits where the overhead of thread management becomes significant. In this chapter, we propose an architectural approach, called *lightweight chip multithreading* (LCMT), to maximize the on-chip fine-grained parallelism by providing direct hardware support for fine-grained threads.

The LCMT architecture inherits two innovative ideas proposed by Kogge et al. [50]: *frames in memory* and *extended memory semantics*. The technique of frames in memory replaces a fixed set of architectural registers in the processor with main memory frames to hold thread state and can support an "unlimited" number of threads. The extended memory semantics (EMS) yield significant improvement over traditional full/empty bits (FEB)[20] by providing hardware support for the most frequent synchronization and thread management operations.

7.1 Contributions

The main contributions of this work include:

- The LCMT architecture is the *first* attempt to combine a mainstream processor architecture and the direct hardware support for fine-grained parallelism that were previously proposed mainly for massively parallel processing by using a cache hierarchy for frames and a *frame prefetch* mechanism. LCMT is based on a mainstream multithreaded processor, specifically Niagara [51], with very small additional hardware overhead. This enables it to evolve with the mainstream designs with minimal additional cost. Moreover, by using the cache hierarchy for frames and the *frame prefetch* mechanism, LCMT ensures that, unlike architectures such as the Cray MTA/XMT [20, 27] and IBM Cyclops 64 [113] that require specialized "smart" memory controllers at the main memory, commodity DRAM chips without specialized memory controllers can be used for the proposed LCMT architecture.

- A methodology to let LCMT leverage legacy software with few changes is proposed. Therefore, current applications can be easily modified to run on the LCMT architecture and achieve better performance. This significantly reduces the learning curve for a programmer working on the proposed architecture.
- We qualitatively evaluate the advantages of the LCMT architecture from power, area, and performance perspective using McPAT and a performance simulator.

The proposed LCMT architecture not only significantly improves performance for irregular and dynamic applications but also provides comparable performance for regular applications on mainstream processors. Using a cycle-accurate simulator, we compare the power and performance of a Niagara-like baseline processor with the proposed LCMT processor running benchmarks written in Cilk. We show that for dynamic and irregular applications the LCMT processor achieves up to up to 1.8X better scalability, 1.91X better performance, and more importantly, 1.74X better performance per Watt, than the baseline processor without hardware support for lightweight multithreading. For regular applications, both the LCMT processor and the baseline processor achieve similar performance.

7.2 Related Work

The related work can be categorized into two areas: hardware support for fine-grained parallelism in customized, early dataflow and multithreaded architectures and software-based techniques for supporting fine-grain parallelism on conventional multicore processors and symmetric multiprocessing (SMP) systems.

7.2.1 Data-flow and Multithreaded Machines

Prior work on hardware support for fine-grain multithreading can be traced back to the early dataflow and multithreaded architectures of the 1970s and 80s. Early dataflow machines, such as the MIT TTDA [7], Monsoon [78], and Hybrid P-RISC [73], provide efficient frame-based parallel execution models. Although some of these dataflow concepts did evolve into the instruction level parallelism (ILP) found in today's superscalar, out-of-order execution processors, the organization of these early dataflow machines differs greatly from current mainstream architectures and they require special programming environments such as the Id [7] language.

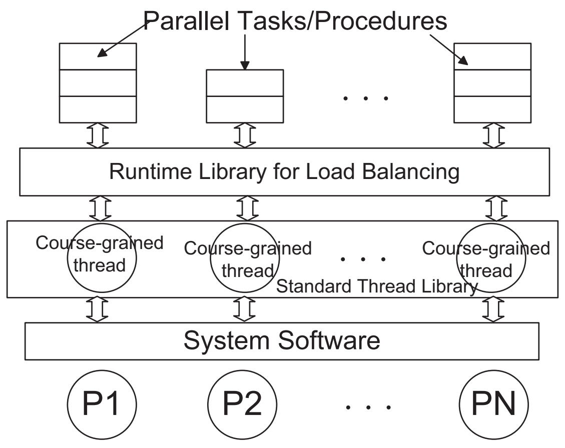
A number of multithreaded architectures, including the Denelcor HEP [96], its successors the Tera/Cray MTA [20], and the newer Cray XMT [27], are inspired by these early data-flow machines. Although the organization of these multithreaded architectures is closer to that of mainstream architectures, they sacrifice single-thread performance for overall throughput. Therefore, although they provide fine-grained parallelism and perform well for irregular and dynamic applications, they cannot be used for general-purpose computing where performance for regular applications and single-threaded applications is also important.

7.2.2 Software-based Approaches for Fine-grained Parallelism

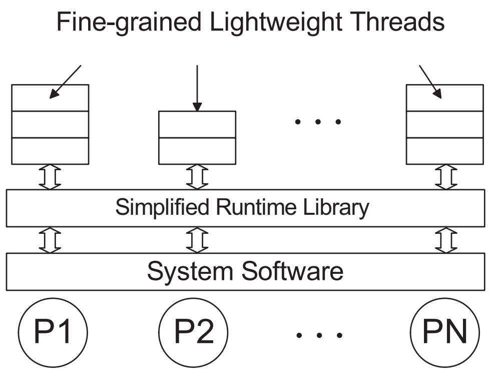
Current multicore processors, especially those that focus on throughput computing, rely on thread level parallelism so that they can be clocked at a lower frequency to avoid unmanageable chip power density and reduce total chip power dissipation while still achieving better overall performance. Unfortunately, none of these processors provide architectural support for fine-grained parallelism which is critical to dynamic and irregular applications. As a result, current multicore processors exhibit degraded performance when running irregular and dynamic applications. In order to solve this problem, software-based lightweight multithreaded environments are proposed. These environments include earlier systems such as Split-C [21] and Earth [71] as well as the latest environments such as Cilk [29] and Intel's threading building blocks (TBB) [80]. Although software-based lightweight multithreading environments narrow the gap between the requirements of irregular and dynamic applications and the hardware support provided by multicore processors, they are still not sufficient to fully exploit the inherent parallelism of those applications.

7.3 Lightweight Chip Multithreading (LCMT)

Software-based lightweight multithreaded approaches, such as Cilk and Intel's TBB, rely on coarse-grained threading libraries to schedule fine-grained parallel tasks or procedures. As shown in Figure 7.1 (a), a runtime library maintains the information for parallel tasks. The standard coarse-grained threads such as Pthreads serve as the real workers from the hardware/OS's perspective, and fine-



(a) A software-based fine-grained lightweight multithreaded program running on a conventional multicore and/or multithreaded architecture



(b) The same lightweight multithreaded program running on the LCMT architecture.

Figure 7.1: Architectures running programs with fine-grained lightweight multithreading. A processing unit is the hardware that can process a hardware thread. In a), a Niagara core can be viewed as four processing units. In b), when a fine-grained lightweight multithreaded program runs on the LCMT architecture, the standard course-grained thread library such as Pthread library is not needed and not used, and the parallel procedures or tasks are cast to the lightweight threads directly supported by the LCMT architecture. Moreover, the runtime library that supports the lightweight multithreaded program is much simpler on the LCMT architecture than that on conventional multicore and/or multithreaded architecture.

grained parallel tasks become work units in the work queues of the standard coarse-grained threads. The runtime library works as the glue between the parallel tasks and the standard coarse-grained threads.

Figure 7.1 (b) shows a conceptual view of the proposed LCMT architecture. Applications executing on the LCMT architecture do not require the use of the standard coarse-grained thread library. In contrast, the parallel tasks are cast to lightweight threads that are directly supported by the LCMT architecture. These fine-grained threads directly communicate with the hardware in the LCMT architecture, which significantly reduces overhead by eliminating the need for coarse-grained threading libraries to act as middleware. This hardware-based approach, including hardware support for lightweight threads and frames as well as Extended Memory Semantics (EMS), provides very efficient thread scheduling, synchronization and management.

To some extent, software-based approaches, such as Cilk and Intel's TBB, can be viewed as a "virtual machine" that tries to create the illusion of an architecture that supports fine-gained parallelism using conventional multicore processors or symmetric multiprocessing (SMP) systems. In contrast, the LCMT architecture provides direct architectural support for fine-gained parallelism, and, as a result, this virtualization overhead is eliminated.

The LCMT architecture aims to not only provide architectural support for fine-grained parallelism but also to achieve this goal with minimum hardware overhead while leveraging existing legacy code. In order to realize this goal, we start from a CMT design similar to the Sun Niagara and incorporate three key enhancements:

- the use of *frames in memory*, in place of a fixed set of architectural registers in the processor, to support an "unlimited" number of fine-grained program threads;

- *extended memory semantics (EMS)* to reduce thread management and synchronization overhead;

- a cache hierarchy for frames and a *frame prefetch* mechanism to further reduce the thread switch overhead and to allow the use of commodity DRAM chips without specialized memory controllers.

The rest of this section will describe the concepts of lighweight multithreading and EMS in LCMT as well as the implementation of the LCMT architecture.

7.3.1 Lightweight Threads and Frames

In a program that exploits fine-grained parallelism, the number of threads can greatly exceed the number of physical register files in the processors. As a result, the system software needs to maintain complicated data structures to keep track of threads' states. When a thread is swapped out of the processor, its state must be packed into the data structures. When a thread is brought into the processor, its state must be unpacked from the data structures maintained by system software and copied to the register file. The cost of copying thread state in and out of registers can quickly become a performance bottleneck.

To alleviate this bottleneck while still supporting a very large number of program threads, the LCMT architecture replaces processor register files with *frames* in memory that contain the local variables and a program counter. Logically, the pool of threads is a list of frames in the memory. Operations to spawn or terminate threads simply add frames to or remove frames from the list and are performed by hardware at the memory. When a thread has all the data it needs to issue the next instruction in its frame, such as the values of operands, that thread is said to be `ready`. Otherwise, the thread is `blocked`. When a thread is running, its frame must be buffered within the LCMT core in a hardware *frame buffer*. We may think of the frame buffer as a tagged register file that is logically part of the memory system. If a two-level caching scheme is used for the frames, it makes the overhead of thread management extremely low.

From the perspective of the lightweight threads, all the LCMT cores are the same and serve only to process instructions for threads in the cores. As long as there are enough `ready` threads so that every LCMT core can issue an instruction from some thread at every clock cycle, the system will run at peak performance. Because the threads' frames are part of the memory system rather than the processor state, the LCMT architecture can theoretically support an "unlimited" number of threads to maximize the inherent parallelism of a program, bounded only by the size of the memory segment used for storing the threads' frames.

7.3.2 Extended Memory Semantics (EMS) for Low Overhead Synchronization

Synchronization is always a challenge for parallel applications, especially for multithreaded applications with intensive inter-thread communication. It becomes even more challenging as the thread granularity becomes smaller, since any synchronization overhead will become more significant. The LCMT architecture uses extended memory semantics (EMS) to provide a mechanism for extremely fine-grained, lightweight synchronization and management of threads. EMS is based on the Full/Empty bit(FEB)[20] technique used in the Cray MTA[20], but goes a step beyond by removing the spin phase of FEB.

FEBs rely on an additional bit associated with each word in memory to identify it's state. This supports synchronization by allowing memory operations to execute conditionally, depending on the state of the memory they are attempting to access. When a memory operation cannot execute because the location being accessed is not in the correct state, the thread issuing that memory operation blocks and the memory controller repeatedly retries the memory operation. The overhead of this spin phase in network and memory bandwidth can be huge.

Let us explore a traditional FEB implementation using a simple example as illustrated in Figure 7.2. A consumer thread, `T2`, wants to consume the value in memory location `A` by issuing the instruction `load.fe R2, A`. This instruction

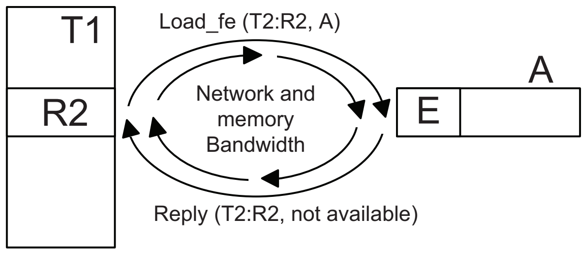


Figure 7.2. FEB Spin Phase

attempts to load the value from memory location `A` to register `R2` of thread `T2` and leave memory location `A` in an `Empty` state. However, memory `A` is already in the `Empty` state and cannot provide the value. Thread `T2` first blocks on memory location `A` to wait for the data while the memory controller repeatedly retries the instruction. If the data is not available within a pre-defined time period, a timeout occurs, and the thread traps to a kernel level software handler—the FEB handler. The thread state is saved to memory and the thread blocks until `A` is marked `Full`. When `A` becomes `Full`, the thread state is reloaded from memory and the `load.fe R2, A` instruction is restarted. The spin phase increases network traffic and wastes memory bandwidth, both of which are very valuable in a shared memory multicore processor. Moreover, there is significant overhead involved in blocking and waking up operations that trap to the kernel level FEB handler.

The fundamental storage unit in the 64-bit LCMT architecture is the *extended double word* or `Xdword`. It consists of a 64-bit double word (`dword`) together with a 65th bit called the *extension bit (Xbit)*. By using the extension bit in tandem with mode fields within the `dword` itself, we define a set of extended memory states for the `dword`. If the extension bit indicates the memory is full, then the 64-bit field contains valid data. Otherwise, the 64-bit field contains metadata needed for

TABLE 7.1

EMS STATES.

State	Name	Description
`F`	Full	The `dword` contains a valid value.
`E`	Empty	The `dword` is empty of a value. This typically indicates that a value is pending for the `dword`.
`FVE`	Forward Value leave Empty	It indicates a store to this `dword` should be forwarded to the `dword` indexed by the metadata, and the state of this `dword` should be changed to the E after forwarding data.
`FVF`	Forward Value leave Full	It is the same as the FVE, except the state after forwarding data is F.
`FRK`	Memory Fork	This invokes a software EMS handler when the location is accessed.

hardware to process the memory references. While the FEB has only two states, there are 13 defined extended memory states in EMS (some of these states are used by OS kernel only), but for the purpose of this discussion, we will reference only five states, shown in Table 7.1.

Load and store instructions are characterized by their synchronization behavior when they encounter memory with non-full EMS states. Table 7.2 summarizes these instructions. The synchronization behavior describes both the precondition and postcondition of the extended state of the memory location, where the precondition is the state that the memory location must be in before the operation

TABLE 7.2

LOAD AND STORE INSTRUCTIONS WITH EMS.

Instruction	Description
`load.ff`:	Load the data when the memory location is full.
`load.fe`:	Load the data when the memory location is full and leave it empty. Block until the memory location is full.
`store.xf`:	Store the data regardless the memory state.
`store.ef`:	Store the data when the memory location is empty. Block until the memory location is empty.
`store.xe`:	Change the memory to state empty regardless its prior state.

can take place and the postcondition is the state in which the memory location is left after the operation takes place. When a memory operation fails to satisfy the precondition (assuming no other error condition exists), it causes the thread to block until the precondition is met. For example, a `load.fe` instruction will block until a memory location is `Full` and then leave it `Empty`, while a `store.ef` instruction will block until a memory location is `Empty` and then leave it `Full`.

Threads block by "queuing up" on the memory location at which they seek to read or write data. The architecture supports this by maintaining a pointer to a thread or a list of threads blocked on that location. Note that this memory location can be a thread register if it is part of a frame, since a lightweight thread's registers "are" the first 32 `Xdwords` in its frame; in other words, there are no general purpose registers in hardware that exist in a namespace separate from memory.

Figure 7.3 illustrates a simple case of the synchronization between a single producer and single consumer. Suppose that `T2` is a consumer thread that wants to read from memory location `A`. When `T2` issues a `load.fe R3, A` instruction to request a load and finds location `A` in the empty `(E)` state, the location changes its state to forward value, leave empty `(FVE)` and its contents are set to the target address of the forwarding operation, which in this case is register `R3` of thread `T2`. `T2` is not placed back on the ready list and is hence blocked on location `A`. Next, the producer thread `T1` issues the `store.ef R2, A` instruction to store the contents of its register `R2` in location `A`. The system detects that location `A` is in the `FVE` state, writes the value in `R2` to `T2`'s register `R3`, and leaves location `A` in the `E` state. Both threads `T1` and `T2` are now reinserted on the ready lists, unblocking the consumer `T2`. While it takes several steps, the common case of a single producer and single consumer can be readily handled by a hardware EMS controller (finite state machine) at the cache bank. Compared to FEB, EMS significantly reduces both the network and memory traffic for the single producer and single consumer scenario.

For more complex situations such as multiple producers and consumers, software or extra hardware support is needed to manage a queue of blocked threads. In our LCMT implementation we use a kernel-level EMS handler for such situations. The full state transition diagram is shown in Figure 7.4. Assume that thread `T1` from Figure 7.3 is not the producer but rather another consumer that wants to consume the data. It issues a `load.fe` instruction, trying to load the data and leave the memory empty. When the EMS controller at the cache bank attempts to execute the instruction, it realizes that the state transition is beyond the hardware's capability and traps to the EMS handler. The EMS handler

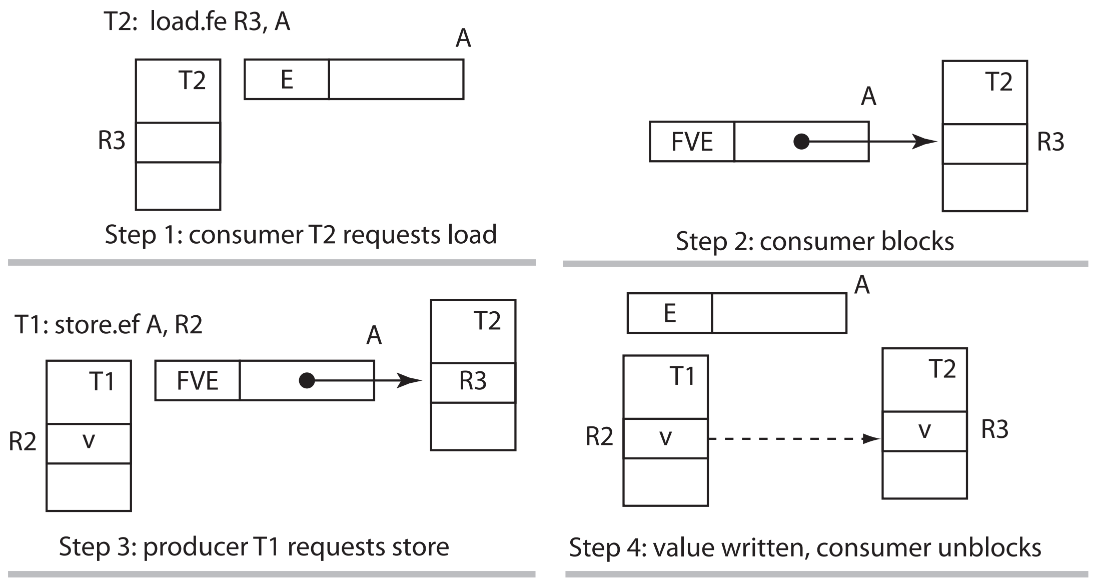

T2: load.fe R3, A
A
T2
E
R3
Step 1: consumer T2 requests load
A
T2
FVE
R3
Step 2: consumer blocks
T1: store.ef A, R2
A
T2
T1
FVE
R3
R2
v
Step 3: producer T1 requests store
A
E
T1
T2
R2
v
v
R3
Step 4: value written, consumer unblocks

Figure 7.3. Single producer, single consumer synchronization

queues the memory requests of both `T2` and `T1` at the memory location. As shown in Figure 7.4, all transitions between `E`, `F`, `FVE` and `FVF` (white circles) can be handled by hardware directly. Only when transitions involve the `FRK` state, will the hardware trap to the EMS handler. This approach leverages the advantages of both hardware and software to balance hardware overhead and performance benefits.

7.3.3 LCMT Processor Architecture

The LCMT processor uses a CMT processor similar to the Sun Niagara processor as its baseline and incorporates lightweight multithreading and EMS to support "unlimited" fine-grained parallelism with minimum hardware overhead. The LCMT processor consists of eight LCMT cores, an on-chip crossbar network, and a shared unified L2 cache, as shown in Figure 7.5 (a). There are three major differences between the LCMT processor and the Niagara processor: 1) the LCMT

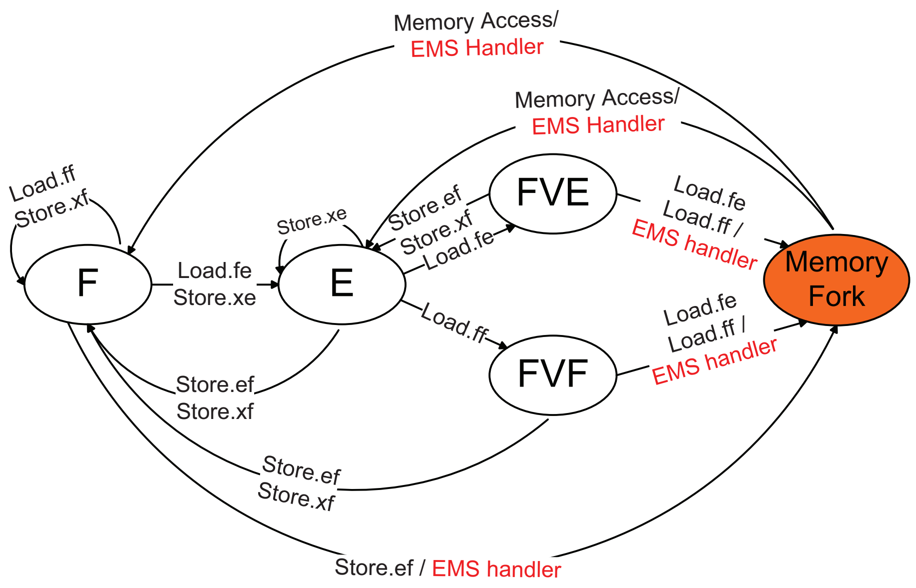
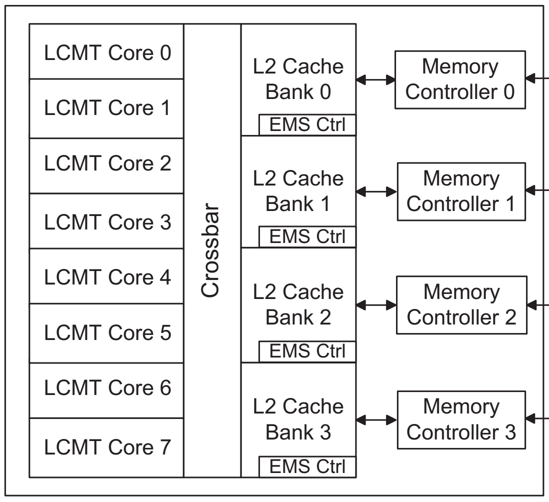


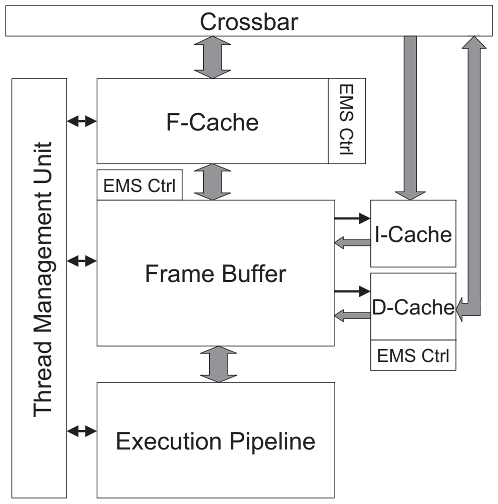
Figure 7.4: EMS State Transition Diagram. A circle represents the state of a memory location. The edge shows the transition between two states. All transitions between white circles can be handled by hardware directly. Only the transitions involving the colored circle need to trap to the EMS handler. Such transitions are denoted by "EMS handler" at the right side of "/" symbol.

processor has an EMS controller attached to each L2 cache bank, 2) in the LCMT architecture the frames share the L2 unified cache, and 3) a *frame prefetching* mechanism is used to further reduce the frame management overhead and ensure the use of commodity DRAM chips without special memory controllers. Since frames are also in the same memory space as instructions and data, there is no change in organization and policies, including the cache coherence policy, of the L2 cache.

Figure 7.5 (b) shows the block diagram of an LCMT core. The LCMT core is similar to the Niagara-like CMT baseline with a six-stage pipeline, but has three major differences. First, an EMS controller is attached to the L1 D-cache to support EMS operations. Second, the 4-way register file in the CMT processor is replaced by a 4-way *frame buffer* that also has an EMS controller. Third, the



(a) LCMT Processor Block Diagram



(b) LCMT Core Block Diagram.

Figure 7.5: LCMT Processor and LCMT Core Block Diagram. Each LCMT core has an FPU, which does not show in the figures.

LCMT core includes thread-management logic and an L1 Frame-cache (F-cache), along with its TLB.

For every LCMT core, there are 4 linked lists in the frame segment in main memory to store the ready frames. Each linked list provides the thread pool for one entry in the frame buffer. The L1 F-cache provides the frame buffer with a pool of fast-access, ready threads that is a subset of all ready threads. In this way, the thread swap between the frame buffer and L1 F-cache is very fast. When a thread swap is requested, the LCMT core will first check the L1 frame cache to find available frames/threads. The frame buffer only updates the L1 F-cache during thread swap. Since the frames are in the same address space as program instructions and data, they could also be cached in the L1 D-cache. However, the frames have different access patterns from instructions and data. Data in frames has great temporal and spatial locality, since they are treated as register values in the frame buffer. Caching frames in the L1 D-cache would increase the conflict rate and affect performance. As a result, a separate L1 F-cache is used to serve the frame buffer. The L1 F-cache is inclusive of the frame buffer, with write-through and allocate on load policies.

The thread-management logic is responsible for swapping threads between the frame buffer and frame caches. The EMS controller attached to the frame buffer records the number of blocked threads. If the number of EMS blocked threads is greater than the threshold (currently, the threshold is set to two and can be easily changed), the EMS controller will trigger the thread-management logic to swap out the first blocked thread to the L1 F-cache. The thread will stay in the L1 F-cache and wait on the data for which it is blocked. The thread may also be swapped out to L2 unified cache or main memory, if the data does not become available for a extended period of time. When the number of blocked threads is lower than the threshold, threads may remain in the frame buffer to wait for data. By changing the threshold, one can easily balance the pipeline throughput and thread swap overhead to achieve optimal performance.

When the data required by a waiting thread (or threads) is produced by another thread, the EMS controller will forward it to the destination frame(s). Since the L2 cache is inclusive of the L1 F-cache and the L1 F-cache is inclusive of the frame buffer, the current residence of a frame can be easily found by checking the directories of the L2 cache and possibly the L1 F-cache. If the frames reside in the on-chip cache hierarchy, EMS controllers at cache banks perform the resulting EMS operation and handle the frames as described in Section 7.3.2. If the frames are written back from the L2 cache to main memory for reasons such as cache conflicts, they will be brought into the L2 cache so that the EMS operation can be processed by the EMS controller at the L2 banks. Bringing these frames into the L2 cache is controlled by the EMS controller and L2 cache controller, without the need for attention from processor cores. This is called *frame prefetching*.

Like data and instruction prefetching, frame prefetching ensures that the frame is in the processor and in the ready state before the cores have empty slots to run them. This reduces the overhead of bringing the frames/threads from main memory to the processor. It also eliminates the need for EMS controllers at the main memory side, which ensures that commodity DRAM chips can be directly used to build DIMMs for the proposed processors. The frame prefetching policy could be further refined to increase performance. Currently, we use the following basic policy: when data for a pending frame arrives, the corresponding frame is fetched into the L2 cache except when this prefetch causes conflicts with L1 caches (since the L2 cache is inclusive of the L1 caches). When conflicts occur, the EMS controller sends an interrupt signal to the processor core to allow it to properly handle the conflicts and EMS state transitions.

When performing a thread swap, the thread-management logic switches a frame in the frame buffer with a frame on a ready list. This encourages thread swaps in the L1 F-cache or the L2 unified cache which have shorter access times than accessing main memory. A prioritized work-stealing policy is implemented in the thread-management logic for lightweight thread management. It is similar to the *THE* [29] policy used between Pthreads in the Cilk runtime library. The main difference is that, when a ready list is empty, the thread-management logic will pick a frame from one of the ready lists belonging to the same core. Only when all four lists of the same core run out of ready frames does the thread-management logic relinquish control to system software. The system software then steals frames from the lists of other cores. This hardware-based, prioritized, work-stealing ap-

TABLE 7.3: PARAMETERS AND AREA ESTIMATION

Parameters	Baseline processor	LCMT processor
Process Technology (nm)	32 (Year 2013)	32 (Year 2013)
Simulated Clock Rate	1.2GHz	1.2GHz
Threads/Core	4	4
Cores/Die	8	8
Ideal Instruction Per Cycle (IPC)	8	8
L1 I-cache	16KB, 4-way, 32B-line	16KB, 4-way, 32B-line
L1 D-cache	8KB, 4-way, 16B-line	8KB, 4-way, 16B-line, EMS enhanced
L1 F-cache	N/A	16KB, 4-way, 32B-line, EMS enhanced
Miss penalty to L2 Cache	23 cycles	23 cycles
Crossbar bandwidth (GB/S)	134.4	134.4
L2 Cache	3MB, 12-way, 64B line, 4 bank	3MB, 12-way, 64B line, 4 bank, EMS enhanced
Miss penalty to Memory	110 cycles	110 cycles
Simulated System Memory	4GB	4GB
Simulated Memory bandwidth (GB/S)	25.6	25.6
Total EMS Overhead (mm^2)	N/A	Less than 0.33
Thread-management logic Overhead (mm^2)	N/A	Less than 0.14
Core (1 FPU/core) Area (mm^2)	4.2	4.9
Area of all Cores (mm^2)	33.6	39.2
Crossbar Area (mm^2)	4.5	4.5
L2 Cache Area (mm^2)	21.6	21.7
Area of all Memory Controllers (mm^2)	8.6	8.6
Estimated Die Area (mm^2)	68.3	74

proach makes the best use of the available threads in the L1 F-cache in each LCMT core.

7.4 Methodology

7.4.1 Simulated Architecture

We use a Niagara-like CMT processor as the baseline in our simulation and compare the LCMT processor against it. The parameters of both processors, including our area estimates, are listed in Table 7.3. In order to remove the bottleneck presented by the floating point operations, the shared floating point units

(FPU) are duplicated in all the cores for both processors. This same modification has been made on the Niagara 2 processor [45] by Sun. We estimate the area of both designs using a combination of methods including: 1) McPAT [59], 2) Cacti5 [101], and 3) RTL models of both the EMS controller and the thread-management logic. The L1 F-cache of the LCMT processor can hold 64 frames/core. The area estimation of an LCMT core reflects replacement of the register file with the frame buffer, the area of L1 F-cache, the EMS controllers, the EMS enhancements on L1 D-cache, and the thread-management logic. The area estimation of the LCMT L2 cache reflects the EMS overhead. Our comparison shows that an eight-core LCMT processor results in only an 8.3% increase in die area over the baseline processor.

7.4.2 Programming Model

In order to evaluate the proposed LCMT architecture, we chose Cilk [29], a multithreaded programming language and runtime environment developed at MIT, as our programming model. Other software-based environments such as Intel's TBB can also be easily ported to the LCMT architecture. Two things that make Cilk especially attractive are the simplicity of its programming model and its demonstrated performance. Like the general case shown in Figure 7.1, Cilk uses Pthreads as the underlying hardware threads. It supports parallel programming via non-blocking procedure calls identified with a `spawn` keyword. A runtime library manages the parallel procedure calls as work units in the work queues of hardware supported Pthreads. In terms of performance, a `spawn/return` in Cilk is over 450 times faster than a Pthread `create/exit` on a modern Intel processor [57]. Because the LCMT architecture provides architectural support for fine-grained threads, the Pthread library is not needed to execute applications coded in Cilk on the LCMT processor. The Cilk runtime library on the LCMT architecture is also much simpler than that on conventional multicore/multithreaded architectures.

When running on conventional architectures, Cilk uses three different synchronization mechanisms to alleviate the synchronization overhead. Spin locks are used among fine-grained Cilk procedures, since a heavyweight mutex can be a bottleneck for fast synchronization. Although the spin lock is cheap, it is effectively polling memory which wastes both CPU time and power. Between the parent and child procedures, a "sync" [29] mechanism is used to keep track of the states of the children. The "sync" mechanism works like a local memory barrier to maintain the correct behavior between the parent and child. In the runtime library, Pthreads use heavyweight Pthread-mutexes to synchronize.

When executing on the LCMT architecture, the lightweight threads are scheduled by the LCMT hardware and the EMS provides a very efficient synchronization mechanism. Instead of spinning on memory locations, threads are blocked by hardware and "queued up" at the memory location where they seek to read or write data. When a lightweight thread competes for a lock and fails to grab it, it is blocked by the EMS hardware automatically without spinning. This approach saves both CPU time and power. The communication pattern between a parent and its child thread is the same as that between a single producer and a single consumer. Since the single producer and consumer scenario can be easily handled by the EMS, the synchronization between parent and child threads in the LCMT architecture is very efficient.

TABLE 7.4: SUMMARY OF THE BENCHMARKS.

Benchmarks	Problem Size	Thread Behavior	Degree of Load Balance	Contention
Fibonacci	32	Irregular	Unbalanced	None
Queens	16	Irregular	Unbalanced	None
SAT-solver	154309 variables, 3230738 clauses, Unsatisfiable	Irregular	Very Unbalanced	High
Barnes-Hut	1M Particles	Irregular	Unbalanced	Normal
STREAM	Total data size is 1152MB for all 3 vectors	Regular	Balanced	None
RandomAccess	Total data size is 2048MB	Regular	Balanced	None
FFTE	Total data size is 1024MB for the 2 vectors	Regular	Balanced	None

7.4.3 Application Benchmarks

The benchmarks used in these experiments cover both regular and irregular problems with varying degrees of load balance and contention. As detailed in Table 7.4, the benchmarks include Fibonacci, Queens[25], SAT-solver [26], Barnes-Hut[11], and a subset of the High Performance Computing Challenge (HPCC) benchmark suite[61] consisting of STREAM, RandomAccess and FFTE. We implement a zChaff-based, highly multithreaded SAT-solver kernel in Cilk using the techniques described in [49]. The data set used in SAT-solver is the second CNF formula in the VLIW-UNSAT-4.0[104] suite. The data set sizes of the HPCC benchmarks are set according to the requirements in [1]. The listed contention only refers to the contention between the fine-grained threads and does not include the possible contention during thread management and work-stealing.

In the benchmarks with irregular thread behavior, threads are generated and terminated dynamically, and each thread communicates with random timing. Efficient thread management is very important to these irregular problems. In particular, the SAT solver benchmark produces a combination of high irregularity and high contention and there are many instances in the SAT Solver where one thread's operation causes other remote threads to terminate. As a result, the SAT solver benchmark has a very unbalanced workload, and work-stealing happens frequently. The SAT solver also uses locks to achieve mutually exclusive access among the fine-grained threads when modifying shared data structures. The Barnes-Hut benchmark has characteristics similar to the SAT-solver benchmark but with somewhat less irregularity and contention.

Although HPCC benchmarks have an irregular memory footprint [61], they are conveniently parallel with very regular thread behavior that allows each thread to work on a large chunk of the workload independently. In general, benchmarks like HPCC do not need Cilk's fine-grained parallelism as they are able to achieve good performance with the course-grained parallelism provided by Pthreads, openMP, or MPI (when running on a message-passing system). We include these benchmarks here to demonstrate that the proposed LCMT architecture significantly improves the performance for irregular and dynamic applications, without sacrificing the performance for regular applications that perform well on conventional architectures.

7.4.4 Simulation Framework

Our simulations are performed using the Structural Simulation Toolkit (SST) [81], a SimpleScalar based full-system cycle-accurate simulation framework developed by Sandia Labs. The McPAT [59] power, area, and timing modeling framework is paired up with SST to study the power consumption of both architectures. The

McPAT framework has been modified to accurately model the power of special components, such as the EMS controllers in the LCMT processor. A variant of the PowerPC ISA (user-level) is used as the front-end ISA of the simulator. We extend the basic PowerPC ISA to include the `load` and `store` instructions in Table 7.2, as well as a `spawn` instruction that inserts a free frame on the active list, and a `trip` instruction that moves a frame from the active list to the free list.

We use the same Cilk coded benchmarks on both architectures. On the simulated baseline architecture, we run Cilk benchmarks with the original runtime library, where the fine-grained, parallel Cilk procedures are managed by the runtime library and the Pthread library. The Cilk benchmarks are processed through the Cilk tool chain[98] to get the executables. Since the baseline processor has 32 simultaneous threads, there are 32 Pthreads to manage the parallel Cilk procedures during execution.

On the simulated LCMT architecture, the parallel Cilk procedures are cast to hardware supported lightweight threads. Running the benchmarks does not require the use of the Pthread software layer. In order to achieve this goal, we modify the cilk2c source-to-source compiler in the Cilk package, and redirect the parallel Cilk procedures to hardware supported lightweight threads on the LCMT architecture. Some of the lightweight threads contain sequential procedures (i.e. the normal function calls that are not marked by the `spawn` keywords in the benchmarks). Although these sequential procedures could be easily implemented as parallel procedures and cast to lightweight threads in the LCMT architecture by adding the `spawn` keyword, we choose not to do so to ensure that identical Cilk benchmark suites are used in both the LCMT architecture and the baseline architecture. For these threads, a small fixed-size stack (4KB) that contains only one memory page is allocated for each lightweight thread when the thread is spawned. Since all the recursive procedures are marked by the `spawn` keyword and become parallel lightweight threads, the 4KB stack is sufficient to hold the sequential procedures. When allocating these small stacks, a memory pool technique is used to avoid bottlenecks. The modified cilk2c source-to-source compiler first scans all the parallel Cilk procedures in the code and marks those lightweight threads needing the 4K stack. Lightweight threads are then created accordingly.

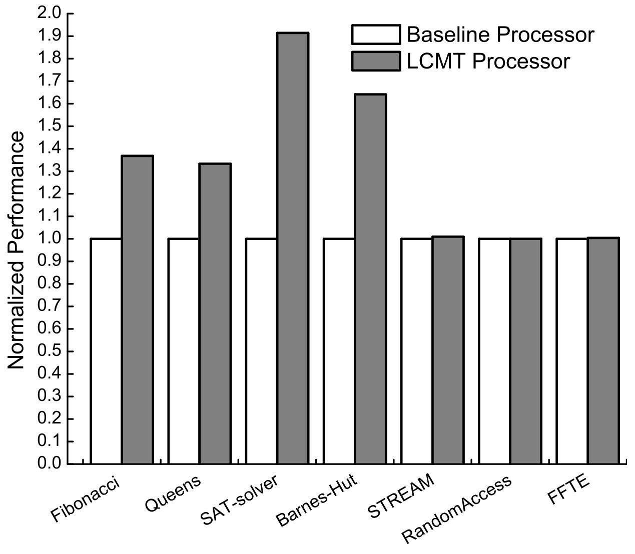
7.5 Results

This section presents the results of simulations comparing the Niagara-like CMT baseline processor and the LCMT processor. A number of different metrics are used to evaluate the processors including performance, efficiency, instructions per cycle (IPC), instruction count (IC), and scalability.

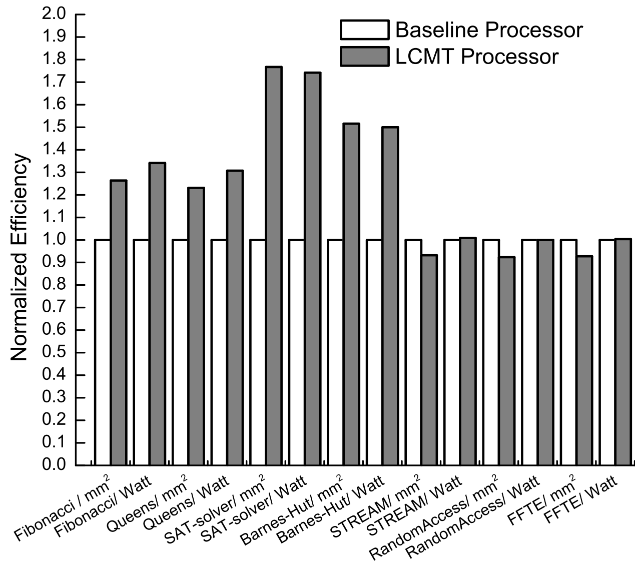
7.5.1 Performance and Efficiency

In order to compare performance, we measure the number of execution cycles for each of the benchmarks running on both simulated processors. By dividing the number of execution cycles by the target processor clock rate, we get the effective execution times and calculate performance as the reciprocal of execution time.

Figure 7.6 (a) shows the performance of the two processors normalized to the baseline processor. The irregular and dynamic benchmarks, including Fibonacci, Queens, Barnes-Hut, and SAT-solver, are shown to achieve better performance on the LCMT processor than on the baseline processor. The LCMT processor outperforms the baseline processor by 91.4% for the SAT-solver benchmark, which has highly unbalanced workloads and high contention. The Barnes-Hut benchmark has characteristics similar to the SAT-solver but with less irregularity, resulting



(a) Normalized Performance for LCMT and baseline architectures



(b) Normalized Performance/mm^2 and Performance/watt for LCMT and baseline architectures

Figure 7.6: Processors showing their performance and efficiency. All the numbers are normalized to the the Niagara-like baseline processor—which is therefore at 1 for all the benchmarks in both figures.

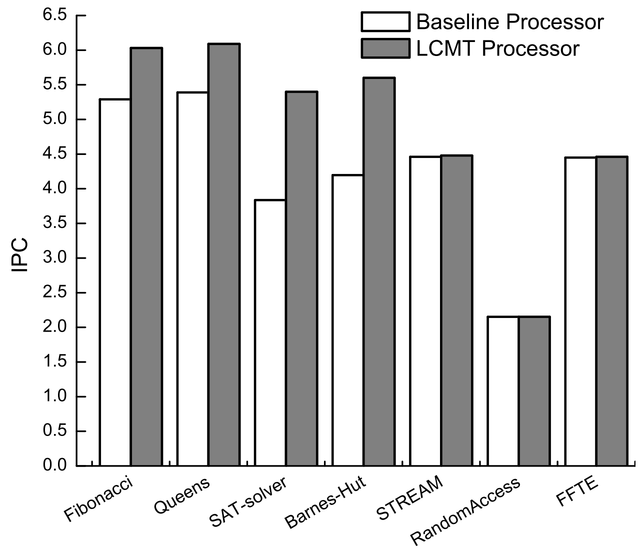


Figure 7.7. IPCs for LCMT and Niagara-like baseline Processors

in a 64.2% performance improvement on the LCMT processor. For the regular HPCC benchmarks, including STREAM, RandomAccess, and FFTE, both processors achieve similar performance.

Figure 7.6 (a) also gives us insight into single-thread performance of the LCMT processor. The STREAM benchmark [1, 61] from HPCC is an embarrassingly parallel benchmark for which no particular effort is needed to segment the problem into parallel tasks. In STREAM, all the parallel tasks work independently on their local chunk of data, and there is little thread management overhead. Therefore, the performance of the STREAM benchmark on both the LCMT processor and the baseline processor can be viewed as just the sum of the performance of each of the independent threads on their eight cores, respectively. The similarity of performance for the STREAM benchmark on both processors reveals that the LCMT processor has single-thread performance comparable to the baseline processor. The features of the LCMT architecture can also be implemented atop

Sun's Niagara 2 [45] processor which has much better single-threaded performance than the Niagara 1 processor. In this way, the single-threaded performance of the LCMT can leverage improvements in the baseline mainstream processor.

Overall, when compared to the baseline architecture, the LCMT architecture performs better for irregular and dynamic parallel applications and has similar performance for regular parallel or single-threaded applications. Together, these results demonstrate the suitability of the LCMT architecture for general-purpose computing that includes a variety of application types.

Figure 7.6 (b) shows the efficiency of the two processors in terms of performance per unit die area and performance per Watt, again normalized to the baseline processor. The die area estimation details were presented in Section 7.4.1. Power estimations are generated by the McPAT framework paired up with the SST framework as mentioned in Section 7.4.4. The LCMT processor achieves up to 74.2% better performance/Watt and 76.7% better performance/mm^2 than the Niagara-like baseline processor. Our experiments clearly demonstrate that the LCMT processor is more power and area efficient than the baseline processor for irregular and dynamic benchmarks. Being power efficient is a significant advantage of the LCMT architecture as power has become one of, if not the most important design constraints. Since the LCMT processor is more power efficient than the baseline processor, it can deliver the same performance while consuming less power and generating less heat than the baseline processor. This can also reduce the cost of packaging and cooling of the LCMT processor.

7.5.2 Instructions Per Cycle (IPC) and Instruction Count (IC)

The performance of a specific benchmark running in a system is determined by the average number of instructions executed per cycle (IPC) and the total

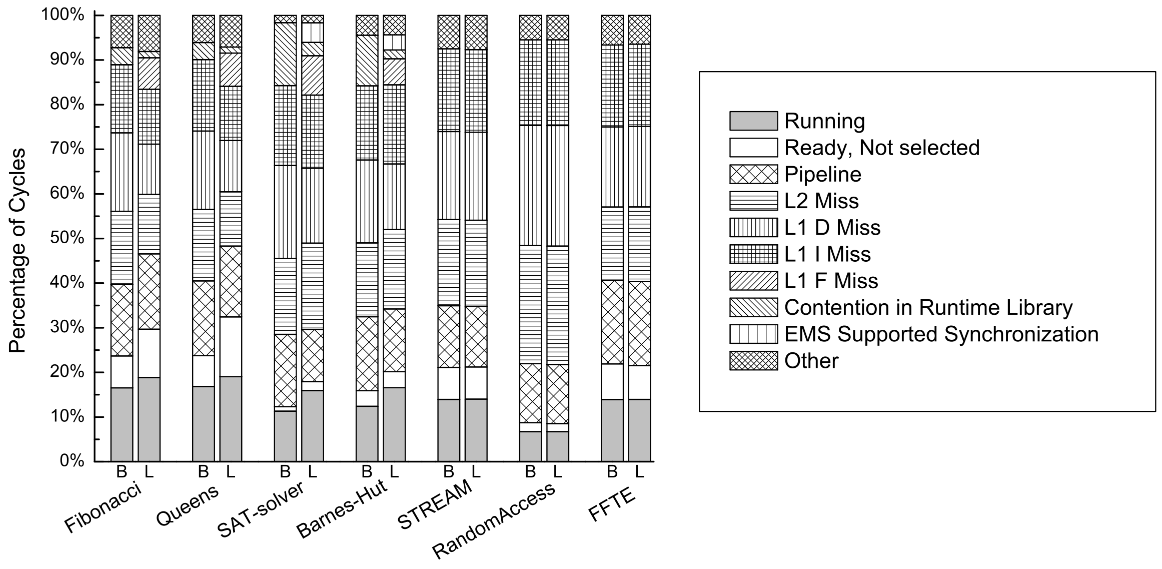
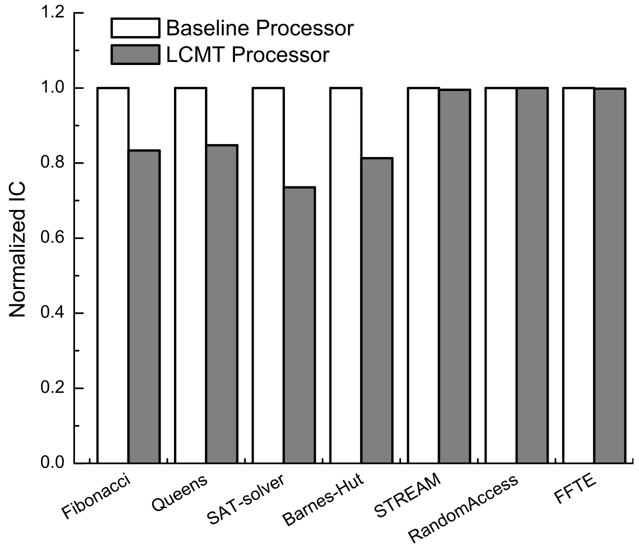


Figure 7.8: Breakdown of the status of an average thread and the causes for the thread being not ready. The "B"s and "L"s on X axis mean the baseline processor and the LCMT processor respectively. For the LCMT processor, the status of an average thread is actually the average status of all threads that have resided in the same entry of the frame buffer during execution. For the Niagara-like baseline processor, the status of an average thread is the average status of all heavyweight Pthreads that have resided in the same thread slot in the processor. The causes for a thread being "not ready" overlap with the not ready state which is not shown. EMS fine-grained synchronization only refers to the synchronization among fine-grained threads. The contentions in the runtime library for the LCMT processor have already included the impact of using EMS in thread management and synchronization in the runtime library.

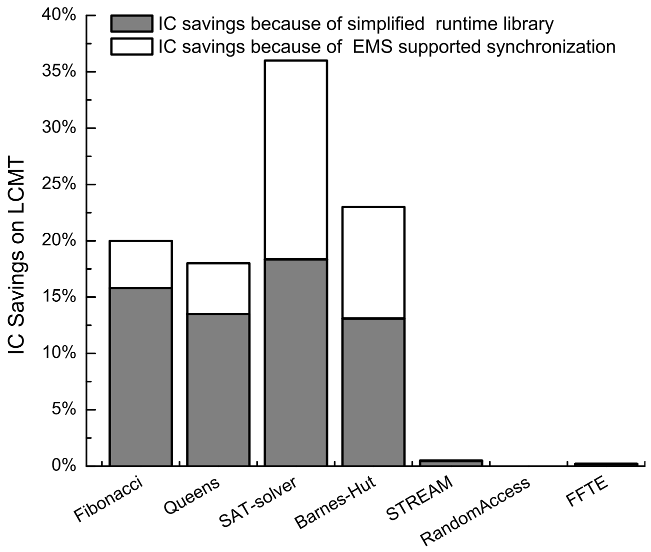
instruction count (IC). Figure 7.7 shows the effective instructions per cycle (IPC) for both processors. For both the baseline processor and the LCMT processor, the ideal per-core IPC is one. The effective IPC for the eight-core processor is the sum of the IPC for all eight cores, which is 8 in the ideal case.

In general, irregular and dynamic benchmarks have better IPCs running on the LCMT processor than on the baseline processor. The LCMT processor achieves 13% to 41% higher IPCs than the baseline processor for irregular and dynamic benchmarks. In particular, for the SAT-solver and Barnes-Hut benchmarks the LCMT processor shows 41% and 33% higher IPCs than the baseline processor. Furthermore, the advantage provided by the LCMT processor is underestimated since the IPC does not reflect the inefficiency of the memory polling performed by spin locks used in the baseline processor. The best effective IPC for the LCMT processor is 76% of the ideal, while the best effective IPC of the baseline processor is 67% of the ideal. For the regular HPCC benchmarks, the IPCs are similar on both processors.

Looking at the behavior of an average thread for both architectures can reveal the reasons for the differences in IPCs. Figure 7.8 shows the breakdown of the status of an average thread. On both processors, an average thread can be in one of the following states: running, ready but not selected, and not ready. A not-ready thread will not stall the core containing that thread; it is only when all four threads are not ready that the core will stall. The higher the frequency of an average thread being not ready, the lower the IPC for the processor. Figure 7.8 also shows the relative frequency of various causes for a thread being not ready. These include cache misses, pipeline delays, and thread contention. In the LCMT architecture, a thread-management logic directed thread swap also causes a thread to be in the state of not ready. This factor is implicit and included in the causes shown in Figure 7.8. When the thread-management logic swaps threads between the frame buffer and the L1 F-cache, the overhead of the thread swap is reflected in the pipeline delay. When the thread swap involves the L2 unified cache, the overhead of the thread swap is reflected in L1 frame misses. When the thread swap involves the threads' frames in main memory, the overhead of the thread swap is reflected in L2 cache misses.



(a) Normalized dynamic instruction counts (ICs) for both Processors. All ICs are normalized to the the Niagara-like baseline processor—which is therefore at 1 for all the benchmarks.



(b) LCMT IC savings breakdown.

Figure 7.9: Normalized dynamic instruction counts (ICs) for both Processors and IC savings breakdown for the LCMT processor.

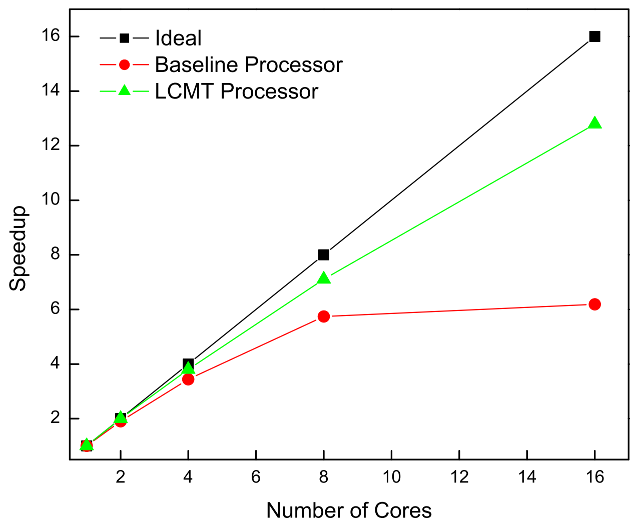
For irregular and dynamic benchmarks, the LCMT processor has fewer cache misses and pipeline delays that cause a hardware thread to be in the state of not ready than the baseline processor. The main reason for this is that the LCMT processor has both enhanced hardware and a simplified runtime library to perform the lightweight thread management and work-stealing while the baseline processor relies on different software layers to achieve the same goal. For the LCMT processor, letting frames share the L2 cache results in a small increase in the L2 cache misses, but the L1 data cache miss rate drops significantly because dedicated L1 F-caches can hold the private variables. This is especially true for the Fibonacci and Queens benchmarks which have few private variables. Irregular benchmarks, especially SAT-solver and Barnes-Hut, show that contention in the runtime library causes a thread to be not ready much more frequently on the baseline processor than on the LCMT processor. This observation confirms that the hardware supported thread management is more efficient than the software alternative. For the HPCC benchmarks, various cache misses are the major reason for the thread being not ready. Since each thread in HPCC benchmark requires very large local data sets, the L1 cache misses are comparable on both the baseline and LCMT processors. Potential delays from cache misses are most severe in the RandomAccess benchmark, which is due to the poor spatial and temporal locality in its memory footprint.

Dynamic instruction count (IC) reduction is another major contributor to the improvements in performance and energy efficiency of the LCMT processor. Figure 7.9 (a) shows the normalized ICs for both processors. The LCMT processor achieves 18% to 36% IC savings for the irregular benchmarks, as compared to the baseline processor. For regular benchmarks, both processors have approximately the same ICs because the IC savings on the LCMT processor are not significant, compared to the huge workload per thread in such benchmarks. The LCMT processor has lower ICs than the baseline processor for the irregular and dynamic benchmarks for the following reasons. (1) EMS and hardware supported lightweight threads are used in the runtime library of the LCMT processor whenever possible to replace the software routines in the original runtime library of the baseline processor, simplifying the runtime library on the LCMT processor. (2) EMS is used for synchronization among fine-grained threads whenever possible on the LCMT architecture while the baseline architecture uses spin locks. In the Fibonacci and Queens benchmarks, about 80% of the IC savings on the LCMT processor are due to the use of EMS and hardware supported lightweight threads in the runtime library whenever possible. In SAT-solver and Barnes-Hut, the same reason contributes 51% and 57% of the total IC savings respectively, and the rest of IC savings are due to the use of EMS for synchronization among fine-grained threads.

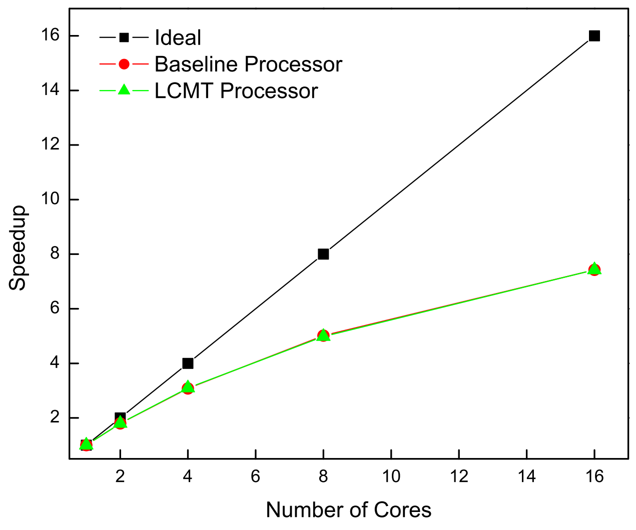
7.5.3 Scalability

Like the Niagara processor family, the proposed LCMT architecture relies on a large number of hardware threads and cores, 32 threads and 8 cores in our example, to achieve high throughput and low power dissipation. Therefore, it is important to study the on-chip scalability of the LCMT architecture. Figure 7.10 shows the speedup of both processors with up to 16 cores. [1]

[1]When a processor has more than 8 cores and uses a crossbar as the on-chip network, the quadratic cost of the crossbar becomes an issue. This is one of the key reasons that Sun chose to have 8 dual-pipelined cores instead of 16 single-pipelined cores when going from Niagara to Niagara2. The 16-core LCMT processor can be easily implemented as eight dual-pipelined cores as in the Sun Niagara2 processors [45]. Therefore, although we show the speedup of 16-core processors, we do not model the chip in detail as we only want to examine the scalability of applications on different core counts.



(a) Speedup of SAT-Solver



(b) Speedup of RandomAccess

Figure 7.10: Multicore Speedups for both Processors. SAT-solve and RandomAccess are used as representatives for irregular and regular applications respectively.

As shown in Figure 7.10, the LCMT processor achieves better speedups than the baseline processor for the irregular and dynamic benchmarks. Thread management and synchronization on the LCMT processor is done primarily by the enhanced hardware while the baseline processor uses software layers to do the job. Moreover, the prioritized work-stealing policy mentioned in Section 7.3.3 also helps to reduce the thread management overhead on the LCMT processor. Therefore, the LCMT processor has much lower thread management and synchronization overhead and achieves better speedups than the baseline processor. Figure 7.10 shows that the more irregular a benchmark is, the worse it scales on the baseline processor. For example, when running the SAT-solver, the baseline processor achieves 1.9X, 3.4X, 5.7X, and 6.1X speedup for 2 cores (8 threads), 4 cores (16 threads), 8 cores (32 threads), and 16 cores (64 threads), respectively, when compared with the single core (4 threads) configuration. In contrast, the LCMT processor's single-chip scalability is less sensitive to the irregularity of the benchmarks than that of the baseline processor. Specifically, for the SAT-solver benchmark the LCMT processor achieves speedups of 2X, 3.8X, 7.1X, and 12.8X when using 2 cores, 4 cores, 8 cores, and 16 cores, compared with the single core LCMT configuration. For regular benchmarks, both processors have similar performance. For both processors, the RandomAccess benchmark shows worse speedup than the other benchmarks. This is due to the fact that it has very poor spatial and temporal locality and therefore a very high cache miss rate which is a major cause for a thread being not ready and available to execute.

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

This chapter summarizes our contributions and research results and suggests future research directions.

8.1 Conclusions

In this dissertation, we have made four major contributions. First, we have developed McPAT to fulfill the needs of current and future computer architecture research. Second, we introduced new and comprehensive metrics, using energy-delay-area^2 product (EDA^{2}P) and energy-delay-area product (EDAP) as examples, for architectural design space exploration. Third, we have used McPAT to study important design points of multithreaded and multi/manycore processors. Specifically, we first study the scaling trends and clustering trade-offs multi/-manycore processors. Fourth, we propose LCMT as an architectural solution for irregular and dynamic applications that are hard to achieve good performance when running in parallel.

McPAT is the *first* tool to *integrate* power, area, and timing models for a *complete* processor, including components needed to model multithreaded and multicore/manycore systems. Unlike prior tools which were tightly integrated with specific performance simulators, McPAT uses an XML interface to decouple the architectural simulator from the power, area, and timing analysis, so that it can be readily used with a variety of simulators.

McPAT's power models account for dynamic, subthreshold leakage, gate leakage, and short-circuit power. It is also the first processor power modeling environment to support clock gating and power management schemes with multiple power-saving states. Like CACTI and Orion, McPAT uses MASTAR [88] to calculate device models based on the ITRS roadmap [88], giving it the ability to model not only bulk CMOS transistors, but also SOI and double-gate devices that will become increasingly important at the 32 nm technology node and beyond. Finally, as CACTI did for memory structures, McPAT includes the ability to determine power- and/or area-optimal configurations for array structures and interconnects, given specified targets for the timing and optimization function. Validation shows reasonable agreement between McPAT's predictions and published data for both in-order and out-of-order processors. We believe McPAT, by providing an integrated power, area, and timing modeling framework for complete manycore processors, will be critical for future architecture research.

McPAT also enables architects to evaluate manycore processor design from a new perspective by using new metrics. Introduced in our work, the metrics of energy-delay-area^2 product (EDA^{2}P) and energy-delay-area product (EDAP) are examples of comprehensive metrics that include both performance and cost, the operational cost (energy) and the capital cost (area). While a chip vendor may favor EDA^{2}P as *area*^{2} provides an approximation to die cost in practice [35], a system vendor could prefer EDAP as other fixed system costs such as memory and I/O reduce the overall system cost dependence on chip multiprocessor cost. The new metrics are proved to be able to reveal new design sweet spots that cannot be found using current metrics otherwise. Moreover, because its advanced device and leakage models adhere to the ITRS roadmap, McPAT can model important transitions between technology generations that happen in industry: for example, it can show that gate leakage peaks at 65nm—reaching 37.6% of the total leakage power—because of the transition to Hi-k metal gate transistors at 45nm. Thus, McPAT can be used, not only by academics to evaluate research ideas, but also by industry designers and researchers to evaluate critical design decisions.

By combining power, area, and timing results of McPAT with performance simulation, we explored the interconnect options of future manycore processors by varying the degree of core clustering over several generations of process technologies. At the 22nm technology node when running PARSEC benchmarks for manycores built from both in-order and out-of-order cores, we found that when cost is not taken into account, clusters of 8 cores provide the best EDP, but when cost is included clusters of 4 cores provide the best EDA^{2}P and EDAP.

The proposed LCMT architecture targets irregular and dynamic applications, without negatively impacting the performance of regular applications. Irregular and dynamic applications are difficult to parallelize efficiently on current multicore and multithreaded architectures that only provide hardware support for coarse-grained parallelism. Our simulations clearly demonstrate that for these types of applications, the LCMT architecture provides better performance and is much more area and power efficient than a CMT architecture using a software-based approach to lightweight multithreading. Specifically, the LCMT architecture achieves up to 1.8X better scalability, 1.91X better performance, and 1.74X better performance per Watt than the baseline architecture for irregular and dynamic applications. For regular applications, the LCMT architecture achieves performance and power efficiency equivalent to or better than the Niagara-like baseline processor, with only minimal differences in area efficiency. Thus, the proposed LCMT architecure is suitable for general-purpose computing and delivers much better performance for irregular and dynamic applications.

The addition of hardware supported lightweight multithreading and EMS eliminates the need for software to “virtualize” the execution of fine-grained threads on the CMT. Furthermore, because the thread-management logic in the LCMT architecture makes full use of the F-caches, the LCMT architecture is able to leverage the locality of the lightweight threads. Thus, the LCMT architecture achieves better scalability than a software-based approach which must perform load-balancing randomly on conventional mutithreaded architectures. Finally, since it can be easily implemented atop a mainstream architecture such as the Niagara processor family, the proposed LCMT architecture can leverage existing software with minimal changes, as discussed in Chapter 7.4.2.

8.2 Future Work

Modeling and design of multithreaded and multi/manycore processors are both challenging and rewarding. New architecture ideas demand improvements of modeling frameworks, and improved modeling frameworks in turn help to foster innovative designs. The work carried out in this dissertation can be extended in several ways.

McPAT supports current major on-chip components including cores, caches, NoCs, and memory controllers. As technology keeps scaling, more and more hardware resources can be put on the same chip. Therefore, components previously off-chip can placed on-chip to achieve better integration and performance. For example, Ethernet modules and cryptography engines have been included in Niagara processor family [45, 51]. Although these new on-chip components are not standard parts now, we believe they will be more and more important for future chip multiprocessors. Therefore, it would be good to extend McPAT to support these components. Adding GPGPU models into McPAT would also be a good direction since GPUs are not only important for graphical tasks but also demonstrate big potential for general purpose computing. It is also crucial to constantly update McPAT's technology level models for the latest technologies such as air-gap wires and 3D stacking technology since new technologies are the working horses that make Moores' Law still valid in deep submicron process.

With the added capability of McPAT, many evaluations that were previously not feasible can be conducted now. One interesting future work direction would be to study the cost of resource sharing, including the cost of coherence, the cost of memory bandwidth allocation, and the cost of on-chip message-passing. Sharing a modest number of resources among cores in a manycore processor can lead to improved performance through load balancing. However, sharing resources efficiently across a large number of cores is becoming increasingly difficult because of two trends. First, global wires scale poorly with shrinking process technology. Second, the overhead of maintaining a coherent global memory space using hardware technique grows dramatically with core count. Another important future direction would be power-efficient processors, as these processors are crucial to achieving power-efficient systems and building green computing infrastructures. New power management techniques such as the turbo mode in Intel Nehalem [54] and IBM power7 [95] have demonstrated improved power efficiency. However, current power managements techniques focus on processor cores. Power-aware NoCs and memory controllers will also be important to improve power-efficiency of multi/manycore processors.

BIBLIOGRAPHY

1. HPC Challenge awards: Class 2 specification. http://www.hpcchallenge.org.

2. OpenMP parallelization of agent-based models. *Parallel Computing*, 31(10-12):1066 – 1081, 2005.

3. K. Agarwal, H. Deogun, D. Sylvester, and K. Nowka. Power Gating with Multiple Sleep Modes. *ISQED*, 2006.

4. H.-T. Ahn and D. Allstot. A Low-jitter 1.9-V CMOS PLL for UltraSPARC Microprocessor Applications. *JSSC*, 35(3):450–454, 2000.

5. H. Al-Hertani, D. Al-Khalili, and C. Rozon. UDSM subthreshold leakage model for NMOS transistor stacks. *Microelectron. J.*, 39(12):1809–1816, 2008.

6. AMD. AMD Opteron Processor Benchmarking for Clustered Systems. *AMD WhitePaper*, 2003.

7. Arvind and V. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3), 1990.

8. T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2), 2002.

9. C. Auth, M. Buehler, A. Cappellani, C. hing Choi, G. Ding, W. Han, S. Joshi, B. McIntyre, M. Prince, P. Ranade, J. Sandford, and C. Thomas. 45nm High-k+Metal Gate Strain-Ehanced Transistors. *Intel Technology Journal*, 12, 2008.

10. D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 547–556, 2005. ISBN 0-7695-2380-3.

11. J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324(4):446–449, Dec. 1986.

12. C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

13. N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

14. B. Bishop, T. P. Kelliher, and M. J. Irwin. The Design of a Register Renaming Unit. In *Great Lakes VLSI '99*, 1999.

15. S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter Variations and Impact on Circuits and Microarchitecture. In *DAC*, 2003.

16. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

17. D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-63392-2.

18. B. H. Calhoun, F. A. Honore, and A. Chandrakasan. Design methodology for fine-grained leakage control in mtcmos. In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, 2003.

19. J. Chang, M. Huang, J. Shoemaker, J. Benoit, S. Chen, W. Chen, S. Chiu, R. Ganesan, G. Leong, V. Lukka, S. Rusu, and D. Srivastava. The 65-nm 16-MB Shared On-Die L3 Cache for the Dual-Core Intel Xeon Processor 7100 Series. *IEEE Journal of Solid-State Circuits*, 42(4), Apr 2007.

20. C. Corporation. Cray mta-2 system [online].

21. D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93*, pages 262–273, 1993.

22. L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 05 (1):46–55, 1998.

23. W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. 2003.

24. Denali. Using Configurable Memory Controller Design IP with Encounter RTL Complier. *Cadence CDNLive!*, 2007.

25. T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Sharend Memory Programming.* Wiley Series on Parallel and Distributed Computing. John Wiley and Sons, Inc., Hoboken, NJ, 2005.

26. Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electr. Notes Theor. Comput. Sci.*, 128(3):75–90, 2005.

27. J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, Ischia, Italy, 2005. ISBN 1-59593-019-1.

28. T. Fischer, F. Anderson, B. Patella, and S. Naffziger. A 90 nm variable frequency clock system for a power-managed itanium architecture processor. In *ISSCC*, pages 294–295, Feb. 2005.

29. M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of theCilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

30. V. George, S. Jahagirdar, C. Tong, K. Smits, S. Damaraju, S. Siers, V. Naydenov, T. Khondker, S. Sarkar, and P. Singh. Penryn: 45-nm next generation Intel core 2 processor. In *ASSCC'07 IEEE Asian Solid-State Circuits Conference*, 2007.

31. A. D. Gloria and M. Olivieri. An application specific multi-port RAM cell circuit for register renaming units in high speed microprocessors. In *ISCAS*, 2001.

32. M. K. Gowan, L. L. Biro, and D. B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *DAC*, 1998.

33. S. Gupta, S. Keckler, and D. Burger. Technology Independent Area and Delay Estimates for Microprocessor Building Blocks. Tech. rep., UT Austin, Department of Computer Science, 2000.

34. J. R. Haigh, M. W. Wilkerson, J. B. Miller, T. S. Beatty, S. J. Strazdus, and L. T. A Low-Power 2.5-GHz 90-nm Level 1 Cache and Memory Management Unit. *IEEE Journal of Solid-State Circuits*, 40(5), May 2005.

35. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach. 4th Edition.* 2007.

36. J. L. Henning. Performance Counters and Development of SPEC CPU2006. *Computer Architecture News*, 35(1), 2007.

37. G. Hinton, D. Sager, M. Upton, D. Boggs, D. P. Group, and I. Corp. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 1, 2001.

38. R. Ho. *On-chip Wires: Scaling and Efficiency*. PhD thesis, Stanford University, 2003.

39. M. Horowitz, R. Ho, and K. Mai. The future of wires. Technical report, In Invited Workshop Paper for SRC Conference., 1999. Available at http://velox.stanford.edu/.

40. M. A. Horowitz. TIMING MODELS FOR MOS CIRCUITS. Technical report, Stanford University, 1984.

41. Intel. P6 Family of Processors Hardware Developer's Manual. *Intel White Paper*, 1998.

42. Ir. Frank Vanden Berghen. XML Parser. http://www.applied-mathematics.net/tools/xmlParser.html.

43. A. Jain, W. Anderson, T. Benninghoff, D. Bertucci, M. Braganza, J. Burnette, T. Chang, J. Eble, R. Faber, D. Gowda, J. Grodstein, G. Hess, J. Kowaleski, A. Kumar, B. Miller, R. Mueller, P. Paul, J. Pickholtz, S. Russell, M. Shen, T. Truex, A. Vardharajan, D. Xanthopoulos, and T. Zou. A 1.2 GHz Alpha Microprocessor with 44.8 GB/s Chip Pin Bandwidth. In *ISSCC*, 2001.

44. N. Jennings and M. Wooldridge. *Agent technology: foundations, applications, and markets*. Springer, 1998.

45. T. Johnson and U. Nawathe. An 8-core, 64-thread, 64-bit Power Efficient Sparc SoC (Niagara2). In *ISPD*, 2007.

46. A. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. In *DATE*, 2009.

47. R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), 1999.

48. N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage Current: Moore's Law Meets Static Power. *Computer*, 36(12), 2003.

49. P. M. Kogge and L. Yerosheva. Towards non-copying, highly multi-threaded bcps. Cascade technical report, University of Notre Dame, 2006.

50. Kogge, Peter M., et al. Computer Architectures with Lightweight Multithreaded Architectures. Patent pending.

51. P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2), 2005.

52. D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2), 2003.

53. D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA*, 1981.

54. R. Kumar and G. Hinton. A family of 45nm IA processors. *ISSCC*, pages 58–59, 2009.

55. R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *ISCA*, 2005.

56. H. Lee, K.-Y. K. Chang, J.-H. Chun, T. Wu, Y. Frans, B. Leibowitz, N. Nguyen, T. J. Chin, K. Kaviani, J. Shen, X. Shi, W. T. Beyene, S. Li, R. Navid, M. Aleksic, F. S. Lee, F. Quan, J. Zerbe, R. Perego, and F. Assaderaghi. A 16Gb/s/link, 64GB/s Bidirectional Asymmetric Memory Interface. *JSSC*, 44(4), 2009.

57. C. E. Leiserson. Multithreaded programming in cilk. In *Supercomputing '07*, 2007.

58. A. S. Leon, K. W. Tam, J. L. Shin, D. Weisner, and F. Schumacher. A Power-Efficient High-Throughput 32-Thread SPARC Processor. *JSSC*, 42, 2007.

59. S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Micro*, 2009.

60. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, Jun 2005.

61. P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The hpc challenge (HPCC) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, 2006.

62. P. Mahoney, E. Fetzer, B. Doyle, and S. Naffziger. Clock Distribution on a Dual-Core Multi-Threaded Itanium Family Processor. In *ISSCC*, 2005.

63. M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

64. S. Mathew, M. Anders, B. Bloechel, T. Nguyen, R. Krishnamurthy, and S. Borkar. A 4-GHz 300-mW 64-bit Integer Execution ALU with Dual Supply Voltages in 90-nm CMOS. *JSSC*, 40(1), 2005.

65. K.-S. Min, K. Kanda, and T. Sakurai. Row-by-row dynamic source-line voltage control (RRDSV) scheme for two orders of magnitude leakage current reduction of sub-1-V-VDD SRAM's. In *ISLPED '03*, 2003.

66. N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical report, School of Computing, University of Utah, 2007.

67. S. Narendra, V. De, D. Antoniadis, A. Chandrakasan, and S. Borkar. Scaling of stack effect and its application for leakage reduction. In *ISLPED '01*, 2001.

68. G. J. Narlikar and G. E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Supercomputing '98:*, pages 1–16, 1998.

69. A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and Thermal Management in the Intel Core Duo Processor. *Intel Technology Journal*, 10:109–122, 2006.

70. U. Nawathe, M. Hassan, K. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-Core, 64-Thread, Power-Efficient SPARC Server on a Chip. *JSSC*, 43(1), 2008.

71. S. S. Newmawarkar and G. R. Gao. Measurement and modeling of EARTH-MANNA multithreaded architecture. In *Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 109–114, San Jose, CA, Feb. 1996.

72. J. Nieplocha, A. Márquez, J. Feo, D. Chavarría-Miranda, G. Chin, C. Scherrer, and N. Beagley. Evaluating the potential of multithreaded platforms for irregular scientific computations. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 47–58, 2007.

73. R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 262–272, June 1989.

74. K. Nose and T. Sakurai. Analysis and Future Trend of Short-circuit Power. *IEEE TCAD*, 19(9), 2000.

75. L. Oliker and R. Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *Proceedings of the ACM/IEEE 1999 Conference on Supercomputing*, pages 39–39, Nov. 1999. doi: 10.1109/SC.1999.10047.

76. S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *ISCA*, 1997.

77. H. Pan, K. Asanović, R. Cohn, and C.-K. Luk. Controlling Program Execution through Binary Instrumentation. *Computer Architecture News*, 33(5), 2005.

78. G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *17th International Symposium on Computer Architecture*, pages 82–91, 1990.

79. J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits: A Design Perspective; 2nd ed.* 2003.

80. J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Inc, Sebastopol, 2007. ISBN 0596514808.

81. A. Rodrigues. Structural simulation toolkit (sst). http://www.cs.sandia.gov/sst/.

82. A. F. Rodrigues. Parametric Sizing for Processors. Tech. rep., Sandia National Laboratories, 2007.

83. S. Rusu, S. Tam, H. Muljono, D. Ayers, and J. Chang. A Dual-Core Multi-Threaded Xeon Processor with 16MB L3 Cache. In *ISSCC*, 2006.

84. E. Safi, P. Akl, A. Moshovos, A. Veneris, and A. Arapoyianni. On the Latency, Energy and Area of Checkpointed, Superscalar Register Alias Tables. In *ISLPED*, 2007.

85. N. Sakran, M. Yuffe, M. Mehalel, J. Doweck, E. Knoll, and A. Kovacs. The Implementation of the 65nm Dual-Core 64b Merom Processor. In *ISSCC'07*, pages 106–590, 2007.

86. J. Schutz and C. Webb. A scalable x86 cpu design for 90 nm process. In *ISSCC'04*, page 62, 2004.

87. Selantek. Technical report, 2007.

88. Semiconductor Industries Association. Model for Assessment of CMOS Technologies and Roadmaps (MASTAR), 2007. http://www.itrs.net/models.html.

89. F. Shafai, K. J. Schultz, G. F. R. Gibson, A. G. Bluschke, and D. E. Somppi. Fully Parallel 30-MHz, 2.5-Mb CAM. *IEEE Journal of Solid-State Circuits*, 33(11), Nov 1998.

90. S. Shastry, A. Bhatia, and S. Reddy. A Single-Cycle-Access 128-Entry Fully Associative TLB for Multi-Core Multi-Threaded Server-on-a-Chip. In *ISSCC*, 2007.

91. J. P. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. 2004.

92. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*, Oct 2002.

93. J. Silc, B. Robic, and T. Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond*. 1999.

94. D. Sima. The design space of register renaming techniques. *IEEE Micro*, 20 (5):70–83, 2000.

95. B. Sinharoy. Power7 multi-core processor design. In *MICRO 42*, 2009.

96. B. J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

97. Sun Microsystems. OpenSPARC. http://www.opensparc.net.

98. Supercomputing. *Cilk 5.3.6 Reference Manual*. Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, 2007.

99. I. E. Sutherland, R. F. Sproull, and D. Harris. *Logical Effort:Designing Fast CMOS Circuits*. Morgan Kaufmann, San Mateo, CA, 1st edition, 1999.

100. S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs.

101. S. Thoziyoor, J. Ahn, M. Monchiero, J. Brockman, and N. Jouppi. A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. In *ISCA*, 2008.

102. D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *ISCA*, 1996.

103. S. Vangal, N. Borkar, and A. Alvandpour. A Six-port 57GB/s Double-pumped Nonblocking Router Core. In *VLSI*, June 2005.

104. M. N. Velev. Vliw-unsat-4.0. http://www.ece.cmu.edu/mvelev.

105. H. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *MICRO*, Nov 2002.

106. N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective. 3rd Edition.* 2004.

107. S. Wilton and N. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

108. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

109. K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2), 1996.

110. V. Zaccaria, D. Sciuto, and C. Silvano. *Power Estimation and Optimization Methodologies for VLIW-Based Embedded Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 2003. ISBN 1402073771.

111. K. Zhang, U. Bhattacharya, Z. Chen, F. Hamzaoglu, D. Murray, N. Vallepalli, Y. Wang, B. Zheng, and M. Bohr. SRAM Design on 65-nm CMOS Technology with Dynamic Sleep Transistor for Leakage Reduction. *JSSC*, 40(4):895–901, 2005.

112. Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Tech. rep., University of Virgina, Department of Computer Science, 2003.

113. W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ISCA '07*, 2007.