MICROARCHITECTURE CHOICES AND TRADEOFFS FOR MAXIMIZING PROCESSING EFFICIENCY

by

Deborah T. Marr

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

For my husband

Jim Brayton

and our lovely children

Matthew, Mason, Mariella,

and one more to arrive this winter.

ACKNOWLEDGEMENTS

I am enormously grateful to the many brilliant, caring, and truly remarkable people who have encouraged and supported me in my pursuit of a Ph.D. Without their help, I would never have finished this dissertation.

I am grateful to my advisor, Trevor Mudge, for supporting and guiding me through the process and taking so much time for me out of his busy schedule. He supported and even encouraged me in my unusual approach and assured me that it could be done. I followed his guidance and came out the other end of the tunnel!

I would also like to thank my committee: Farnam Jahanian, Dennis Sylvester, and Bill Martin, who took the time to read and discuss my work.

A couple of people who I would like to call out especially are my mother-in-law and father-in-law, the esteemed Professor Robert Brayton of UC Berkeley and his kind-hearted wife Ruth Brayton who took such interest in the story about why I didn't finish my degree in my first attempt. Professor Brayton discussed my case with his friend and colleague, Professor Ralph Otten, of Eindhoven University. Professor Otten sent me information and offered to take me on as his student! I was humbled by his kind offer and seriously considered following through at Eindhoven, though ultimately decided to continue my degree at the University of Michigan where I had left off. I am grateful for all of their thoughtfulness and encouragement. This degree wouldn't have happened without them.

My colleagues at Intel have been so good to me. How can I ever thank all of them? I am honored to count these remarkable people as both co-workers and friends. Intel supported me throughout the process of this Ph.D. How I am worthy of this, I will never know. I will start with Joe Schutz. When I mentioned that someday I would like to finish my degree, Joe jumped on the suggestion and said that he would fully support me in doing just that in CTG (the research group at Intel)! I was truly astonished, and even more so when Steve Pawlowski said why go to CTG when I could stay in DEG (Digital Enterprise Group) and do the same? John Shen took an interest in what I was doing, and introduced me to Professor Trevor Mudge. I shall always be grateful to John for his time and his advice on how to get started. Such support has touched me and I am truly grateful to all of them. Intel has indeed been very good to me.

My good friends and colleagues in CTG's microprocessor research group then adopted me into their fold and treated me as one of their own. Konrad Lai, Haitham Akkary, Ravi Rajwar, Srikanth Srinivasan, and Chris Wilkerson were a fun team who made me come to lunch with them every day and made sure I participated in their group discussions.

My long-time colleagues in the Oregon CPU Architecture team are like family. We have worked and persevered through so many ground-breaking projects – the P6, the Pentium 4, Nehalem, and all their proliferations, and now the 2012/2013 "Tock" project. A special thank-you goes to Frank Binns, Kris Konigsfeld, Atiq Bajwa, Glenn Hinton, John Holm, Tom Huff, Mike Upton, and many others for encouraging me and allowing me to work on my little side project while they took the brunt of the hard work on the Nehalem project. They covered for me and showed remarkable restraint in not asking me to jump in to help with many fire-fights.

With the completion of my degree, my colleagues welcomed me back with open arms and asked me to lead the 2012/2013 Core Tock as lead architect! I was honored to accept, and I thank them for their confidence in placing such an important responsibility on my shoulders, in spite of my long "absence". As I came back into the fold, I was reminded again of the amazing the talents, enthusiasm, and spirit in this team! Together we will make the next-generation CPU core something to behold.

I'd like to thank Jim Hunt for sharing his power measurement setup with me and showing me how it worked and answering all of my many questions! He was patient and generous with his time and expertise. Thanks also go to Paul Zagacki who answered my questions about SPEC, attaching network drives, and Ryan who answered my questions about how to get the system booted with the latest drivers, cards, CPUs, etc.

I would like to give a very special "thank you" to my parents who I love dearly. My mom, Alice Ho, has always been there for me. She comes running at the drop of the hat! I don't know how I could have made it through life's challenges without her hugs, love, support, and sheer hard work! She is the best mother and grandmother in the world, and a great cook to boot! She has an adventurous spirit, often surprising us. She raised me, my brother, and my sister to be caring, responsible citizens of the world. Thanks Mom for all that you do! My dad, George Marr, was my idol and mentor growing up. My career and various academic achievements were inspired by him. I fondly remember hours of discussions about the meaning of life. He is a philosopher, an explorer adventurer, an achiever, and very generous to all around him. He sure enjoys a good party, too. Thanks Dad, for everything!

I have a wonderful family network that have encouraged, supported, and celebrated with me through all of life's ups and downs: My dear brother and sister, David and Dianah Marr; my half-sister Jennifer Marr; my step siblings James and Ann Liu; my brothers and sisters by marriage, May Marr, Matt Mow, Jane and Dan Burchard, and Mike Brayton. And I have some wonderful nephews and nieces who I am so proud of: Ryan and Ethan Marr, Eric and Evan Mow, Tommy and Lia Burchard. Thank you all for being there for me, I love you all!

Finally, I would like to thank my dear husband, Jim Brayton, who works so hard to support our family and is funny, caring, and ever-patient at the same time! Jim and I are truly a team; he helps me find a way through when I'm certain that I've taken on too much, he is the rock that is the foundation of my dreams, he celebrates my successes more than I do. A more perfect life partner cannot be had. Thank you Jim, I love you! We are the proud parents of some great kids. I just can't get over how we were blessed with such treasures: Matthew, Mason, and Mariella cheer me up and melt my heart just

by showing their sweet little faces, their unique personalities, and their calls of "Mommy!" We are expecting another bundle of joy in February 2009, an event we look forward to with great anticipation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABREVIATIONS

APIC – Advanced programmable interrupt controller.
BAC – Branch address cache
BHR – Branch history register
BIOS – Basic Input/Output System (the firmware code run by a computer when first powered on)
CD – Compact disk
CMP – Chip-level multiprocessing
CPU – Central processing unit
DDR – Double data rate synchronous dynamic random access memory (a type of DRAM).
DDR2 – Double data rate two synchronous dynamic random access memory (a type of DRAM twice as fast as DDR).
DRAM – Dynamic random access memory
DTLB – Data transition lookaside buffer
DVD – Digital video disk
ED2 – (Energy*delay)^2 metric
GCT – Global Completion Table (of IBM's Power 5 microarchitecture)
HT – Hyperthread Technology, Intel's brand name for SMT (simultaneous multithreading)
I/O – Input/output
IA-32 – Intel Architecture for the 32-bit address-space
ICH7 – I/O Controller Hub, part of a high-end Intel chipset that started shipping in 2005.
ID – Identifier
IPC – Instructions per cycle executed
IPC_f – Instructions per cycle fetched
ITLB – Instruction transition lookaside buffer
KB – Abbreviation for kilobyte
L1 – First-level cache
L2 – Second-level cache
L3 – Third-level cache
MB – Abbreviation for megabyte
MGAg – Multiple Branch Global Two-level Adaptive Branch Prediction using Global Pattern History Table
MGAp - Multiple Branch Global Two-level Adaptive Branch Prediction using Per-address Pattern History Tables
MGAs - Multiple Branch Global Two-Level Adaptive Branch Prediction using Per-set Pattern History Tables

MOSFET – Metal-oxide-semiconductor field-effect transistor
MPEG – Moving picture experts group – this group standardized compression formats for video.
MPEG-1 – First video compression format standardized by MPEG, currently used on video CDs.
MPEG-2 – Video compression format standardized by MPEG, used for over-the-air digital television.
MPEG-4 – Video compression format standardized by MPEG which expands MPEG-1 features and support.
MT – Multi-thread mode
OS – Operating system
PHT – Pattern history table
RAT – Register alias table
SMP – Shared memory multiprocessing
IMT – Interleaved multithreading
SMT – Simultaneous multithreading
SOEMT – Switch-on-event multithreading
SPEC – Standard Performance Evaluation Corporation (generally refers to the benchmarks released by this corporation for comparing computer performance).
SPEC2000 – Benchmark suite released by SPEC in 2000.
SPEC89 – Benchmark suite released by SPEC in 1989.
SPECfp – Portion of the SPEC benchmark suite that measures floating point application performance.
SPECint – Portion of the SPEC benchmark suite that measures integer application performance.
SPECjbb – Portion of the SPEC benchmark suite that measures the performance of server-side Java applications.
SPECweb – Portion of the SPEC benchmark suite that measures internet application performance.
SRAM – Static random access memory
ST – Single-thread mode
ST0 – Single-thread mode where logical processor 0 is active
ST1 – Single-thread mode where logical processor 1 is active
TC – Trace cache
TLB – Transition lookaside buffer
uROM – Read only memory structure that stores the microcode of the CPU.
VLIW – Very long instruction word

ABSTRACT

This thesis is concerned with hardware approaches for maximizing the number of independent instructions in the execution core and thereby maximizing the processing efficiency for a given amount of compute bandwidth. Compute bandwidth is the number of parallel execution units multiplied by the pipelining of those units in the processor. Keeping those computing elements busy is key to maximize processing efficiency and therefore power efficiency.

While some applications have many independent instructions that can be issued in parallel without inefficiencies due to branch behavior, cache behavior, or instruction dependencies, most applications have limited parallelism and plenty of stalling conditions.

This thesis presents two approaches to this problem, which in combination greatly increases the efficiency of the processor utilization of resources. The first approach addresses the problem of small basic blocks that arise when code has frequent branches. We introduce algorithms and mechanisms to predict multiple branches simultaneously and to fetch multiple non-continuous basic blocks every cycle along a predicted branch path. This makes what was previously an inherently serial process into a parallelized instruction fetch approach. For integer applications, the result is an increase in useful instruction fetch capacity of 40% when two basic blocks are fetched per cycle and 63% for three blocks per cycle. For floating point benchmarks, the associated improvement is 27% and 59%.

The second approach addresses increasing the number of independent instructions to the execution core through simultaneous multi-threading (SMT). We compare to another multithreading approach, *Switch-on-Event* multithreading, and show that SMT is far superior. Intel Pentium 4 SMT microarchitecture algorithms are analyzed, and we look at the impact of SMT on power efficiency of the Pentium 4 Processor. A new metric, the SMT Energy Benefit is defined. Not only do we show that the SMT Energy Benefit for a given workload with SMT can be quite significant, we also generalize the results and build a model for what other future processors' SMT Energy Benefit would be. We conclude that while SMT will continue to be an energy-efficient feature, as processors get more energy-efficient in general the relative SMT Energy Benefit may be reduced.

CHAPTER 1
INTRODUCTION

This thesis is concerned with the optimization of processor resource utilization. At its most basic level, a processor consists of arithmetic, logic, and branch processing capability, i.e., execution units. The maximum rate at which execution units are capable of executing instructions is the maximum execution throughput that the processor can achieve. The role of everything else - instruction fetch capacity, branch predictors, caches, memory, I/O - is to feed the execution units as efficiently as possible. Let us call "everything else" the support hardware. If the execution units are 100% busy doing useful work, then the support hardware is doing its job and no further improvement is possible. At that point, the way to increase performance is to add raw execution capability, a task which, on balance, is a rather simple thing to do. Indeed, every generation of processor generally adds some execution capacity, some of it general execution capacity and some of it specialized capacity (e.g., Intel's MMX, SSE, SSE2, etc.).

The ideal workload consists of an easily predictable instruction stream, easily predictable data, and long flows of instructions that exactly match the capacity of the execution units. However, this type of workload is unusual outside of some application-specific domains. In such cases, it is often possible to design special purpose processors where the execution units are chosen to fit the workload, while support hardware is designed with sufficient capability to keep those execution units as close to100% operating efficiency as possible.

However, the general-purpose processor must be designed to execute well on a wide range of different types of workloads. A sufficient array of execution units is assembled and circuit designers are employed to make them as fast as possible. Then the architect focuses most of his/her time on the support hardware. Although it is a challenge, a well-known list of general techniques is available: out-of-order execution, complex branch predictors, caches, TLBs, memory address predictors, etc. If the execution units are to be efficiently fed, the appropriate combination must be selected based on the technology and trends in application profiles. The key points are: how can we provide a continuous stream of ready instructions to feed the voracious appetites of those execution units, and how can we provide data to the instructions? It is rather like feeding heat, fuel, and oxygen to a fire. All three are needed to keep the fire going. Similarly, instructions and data must both be there to keep the execution units busy.

This thesis focuses on the processor resource utilization efficiency problem: how to keep all those execution units busy. Better resource utilization leads to better power utilization and less waste. In particular, this thesis will discuss the delivery problem: how to get more instructions per cycle to the execution units so that they are not idle, waiting for instructions. Then we will discuss how to improve the mix of instructions such that there is more parallelism. Finally, we will measure and show the energy efficiency of using resources more effectively.

The first basic problem of instruction delivery is overcome the basic block limit for fetching instructions. A basic block is the sequence of instructions following a branch instruction up to and including the next branch instruction in the sequence. The number of instructions that can be accurately and efficiently fetched is a key metric for keeping the execution units busy. One approach is to enlarge the basic blocks into traces [1] or into superblocks [2]. Forming instructions into traces and storing those traces assumes that the trace will be used enough to justify the area of the trace cache, and that the branch predictions made while making the traces will be accurate in future executions of the trace. Also, in the first execution of the instructions, the fetch delivery rate will be limited by the branch prediction bandwidth. Another approach is to exploit predicated execution to schedule instruction execution along multiple execution paths [3]. The disadvantage of predicated execution is that execution bandwidth is wasted on instructions whose results are discarded and instruction fetch bandwidth is wasted on instructions which will not be executed.

In this thesis we analyze a different approach which allows us to fully utilize the fetch and execution bandwidth with useful instructions from a dynamically predicted path. This approach also takes advantage of the latest branch predictor information on every instruction fetch. Our highly accurate branch prediction algorithm is capable of making predictions for multiple branches in a single cycle. We also include a branch address cache to provide the addresses of the basic blocks to which the branches direct the instruction flow, and an instruction cache configuration with a suitably high bandwidth. If we can correctly predict two to three branch paths every cycle and if the average basic block size is five instructions, then potentially the average fetch size can be doubled or tripled. Our analysis shows that this capability increases the useful instruction fetch capacity by 40% and 63% when 2 or 3 basic blocks can be fetched each cycle, respectively, for integer benchmarks. For floating point benchmarks the improvement is 27% and 59%.

The Intel Pentium 4 Processor uses a trace cache to provide similar instruction fetch capability. A trace is a concatenation of multiple basic blocks that together represents a likely path of execution. Since the trace cache is located after the branch predictor and instruction decoder, the branch predictions are effectively recorded in the trace. Also, since each trace starts at the beginning of a trace cache line and the instructions are already decoded, instruction fetch from the trace cache is very efficient. The trace cache in the Pentium 4 implementation can deliver 8 useful instructions each cycle. Those instructions originally may have spanned up to 3 non-consecutive basic blocks and may have been split across several cache lines. A trace cache is an alternative way to deliver performance similar to multiple branch prediction.

Once instruction delivery to the processor core is sufficient, we turn our attention to how the instruction mix can be efficiently executed. A major bottleneck to this is the number of independent instructions that can be executed simultaneously in the mix. A way of delivering more independent instructions into the processor is to implement SMT. It potentially doubles the number of independent instructions in the processor by fetching instructions from two (or more) software threads. This thesis discusses the implementation in the Pentium 4 processor and presents analysis of SMT on modern high-performance out-of-order processors. The analysis uses a wide variety of commercial software as well as common benchmarks to evaluate microarchitecture choices and tradeoffs. The workloads include applications from desktop, workstation, high-performance computing, server, and the SPEC suite of benchmarks. Detailed execution-driven simulators that were used in the development of the Pentium 4 microarchitecture, and real system measurements will be the tools for evaluating the microarchitecture optimizations.

More instructions in the processor that are independent means better processor resource utilization. Improvements in processor resource utilization also benefit energy efficiency. Energy efficiency is becoming an increasingly important aspect of design due to mobility requirements, sustainability, and cost. This thesis analyzes the energy efficiency of a processor with both a trace cache and simultaneous multithreading. We find that indeed the energy efficiency is improved. We show that although SMT power is typically ~5-15% higher than single-thread (ST) power, the energy efficiency can be quite substantial when the SMT speedup is > 1.1. A new metric, the SMT Energy Benefit, is defined and used to show that for a given increment of SMT speedup, approximately 80% of that directly lowers energy usage, while 20% is spent to obtain that speedup on the Intel Pentium 4 Processor. We then generalize the results and build a model for what future processors' SMT Energy Benefit might be. We conclude that SMT will continue to be an energy-efficient feature, however as processors get more energy efficient, the relative SMT Energy Benefit will be reduced.

1.1 Contributions

This thesis makes the following contributions:

- Introduce novel algorithms to predict and fetch multiple basic blocks, a key to improving execution efficiency. We introduce the multiple branch predictor and its companion instruction fetch mechanism. Combined, these new algorithms get around the basic block limitation, which previously forced instruction fetch to be an inherently serial process. We also use the best and most up-to-date branch prediction history to make those predictions.
- Develop an in-depth understanding of how simultaneous multithreading (SMT) and switch-on-event multithreading (SOEMT) can provide more varied instructions to the execution units and improve execution efficiency. We show that SMT gives better utilization and therefore better performance on a wider variety of applications than SOEMT.
- Examine the SMT implementation of the Intel Pentium 4 and Xeon family, and analyze the resource utilization choices made. We show why resources have different utilization patterns, and how those utilization patterns necessitate different approaches to sharing the resource. We consider how SMT changes the fundamentals of key algorithms and tradeoffs. We also discuss how SMT introduces new deadlock conditions and fairness requirements which impact the choice of implementation algorithms.
- Provide first measurements and analysis of a real SMT system, on microbenchmarks and a wide variety of real applications.
- Define and a new metric, the SMT energy benefit. We show that on the Pentium 4 Processor, for a given increment of SMT speedup, approximately 80% of that directly lowers energy usage, while 20% is spent on additional power to enable the speedup. We show that leakage power is proportional to SMT area growth, while dynamic power depends on SMT speedup.
- Create a simple power model to estimate the expected SMT Energy Benefit of future, possibly more power-efficient, processors.

1.2 Organization

Chapter 2 introduces the multiple branch predictor and instruction fetch mechanism that vastly improves instruction fetch rate. Increasing the number of instructions sent to the execution resources improves processor resource utilization. Chapter 3 discusses a variety of threading methods, including simultaneous multithreading, and shows that simultaneous multithreading can provide more and better resource utilization and performance than the alternative switch on event multithreading. Chapter 4 discusses the Intel Pentium 4 Processor microarchitecture and the microarchitecture decisions, tradeoffs, and challenges. Chapter 5 is an in-depth study that shows why three different resource sharing protocols were critically important and were the right choices for different parts of the microarchitecture. Chapter 6 looks at the power and energy efficiency of a processor with simultaneous multithreading. Finally Chapter 7 presents some concluding remarks.

CHAPTER 2
INCREASING INSTRUCTION FETCH RATE VIA MULTIPLE BRANCH PREDICTION

As architectures become increasingly parallel, it is important to fetch more and more instructions each cycle. This can be done either by increasing basic block size and fetching the entire block in a single cycle, or by fetching multiple basic blocks per cycle. The optimal solution may be to combine both. The first approach is being researched and implemented in today's advanced compilers. One approach is to enlarge the basic block into traces [1] or into superblocks [2]. Another approach is to exploit accurate predicated execution to schedule instruction execution along multiple execution paths [3]. The disadvantage of predicated execution is that execution bandwidth is wasted on instructions whose results are discarded, and instruction fetch bandwidth is wasted on instructions which will not be executed.

Here we propose a scheme which allows us to more fully utilize the fetch and execution bandwidth with useful instructions from a dynamically predicted path. We published this work in [4].

There are three essential components to providing the ability to fetch multiple basic blocks each cycle:

- Predicting the branch paths of multiple branches each cycle.
- Generating fetch addresses for multiple and possibly non-consecutive basic blocks each cycle.

- Designing an instruction cache with enough bandwidth to supply a large number of instructions from multiple, possibly non-consecutive basic blocks.

This chapter discusses an integrated solution for these problems. We introduce a highly accurate branch prediction algorithm capable of making predictions for multiple branches in a single cycle, a branch address cache to provide the addresses of the basic blocks to which the branches direct the instruction flow, and an instruction cache configuration with a suitably high bandwidth. Although hardware intensive, these solutions are not excessively so for today's modern processor implementations.

If we can correctly predict two to three branch paths every cycle and if the average basic block size is five instructions, then the average fetch size will be 10 to 15 instructions. Many non-numeric applications today have an average basic block size of 5 instructions, and floating point applications tend to be much larger. The ability to fetch multiple basic blocks per cycle coupled with compiler technology to increase basic block size can result in significant performance gains. This chapter shows that simply providing the ability to fetch multiple instructions without specific compiler optimizations already increases the useful instruction fetch capacity of a machine by 40% when 2 basic blocks can be fetched each cycle, or 63% for 3 basic blocks, in the case of integer benchmarks. For floating point benchmarks, the improvement is 27% and 59%, respectively.

In Section 2.1 we provide an overview of the branch prediction work that this work builds on. Our multiple branch prediction algorithm is based on the Two-level Adaptive Branch Predictor [5-7]. The Two-level Adaptive Branch Predictor achieves an average of 97% accuracy. An instruction supply mechanism [8] to do back-to-back branch predictions and supply up to one basic block per instruction cache fetch is also briefly reviewed.

Section 2.2 describes the multiple basic block supply mechanism, the multiple branch prediction algorithms, and the structure and operation of the branch address cache, and the instruction cache design issues. Section 2.3 describes the simulation model and the benchmarks used, and Section 2.4 our simulation results.

Section 2.5 discusses other related research on multiple branch prediction or alternative methods of increasing instruction delivery, and finally Section 2.6 summarizes the benefits and trade-offs of multiple branch prediction.

2.1 Branch Prediction Previous Work

2.1.1 Two-level Adaptive Branch Predictor

Yeh and Patt [5-7] introduced several implementations of the Two-level Adaptive Branch Predictor, each with somewhat different cost vs. prediction accuracies. The average prediction accuracy on the SPEC89 benchmarks was shown to be 97%. One important result was that each of the different Two-level Adaptive Branch Prediction schemes can achieve the same accuracy by varying its configuration. The following is a brief overview of the schemes. The interested reader is referred to the original papers for more details.

The Two-level Adaptive Branch Predictor uses two structures, a Branch History Register (BHR), and a Pattern History Table (PHT). The BHR is used to record the history of taken and not taken branches. For example, if the recent history of the branch behavior is: taken twice, not taken, and taken again, then the BHR would contain the pattern 1101, where 1 indicates taken, and 0 indicates not taken.

In addition, for each possible pattern in the BHR, a pattern history is recorded in the PHT. If the BHR contains k bits to record the history of the last k branches, then there are 2^k entries, each entry containing a 2-bit up-down saturating counter to record the execution history of the last several times the corresponding pattern occurred in the BHR. Yeh and Patt showed that the 2-bit up-down saturating counter was sufficient in keeping pattern history to give highly accurate branch predictions. Prediction decision logic interprets the two pattern history bits to make a branch prediction. If the 2-bit up-down saturating counter is used, the prediction is usually based on the high-order bit of the counter value.

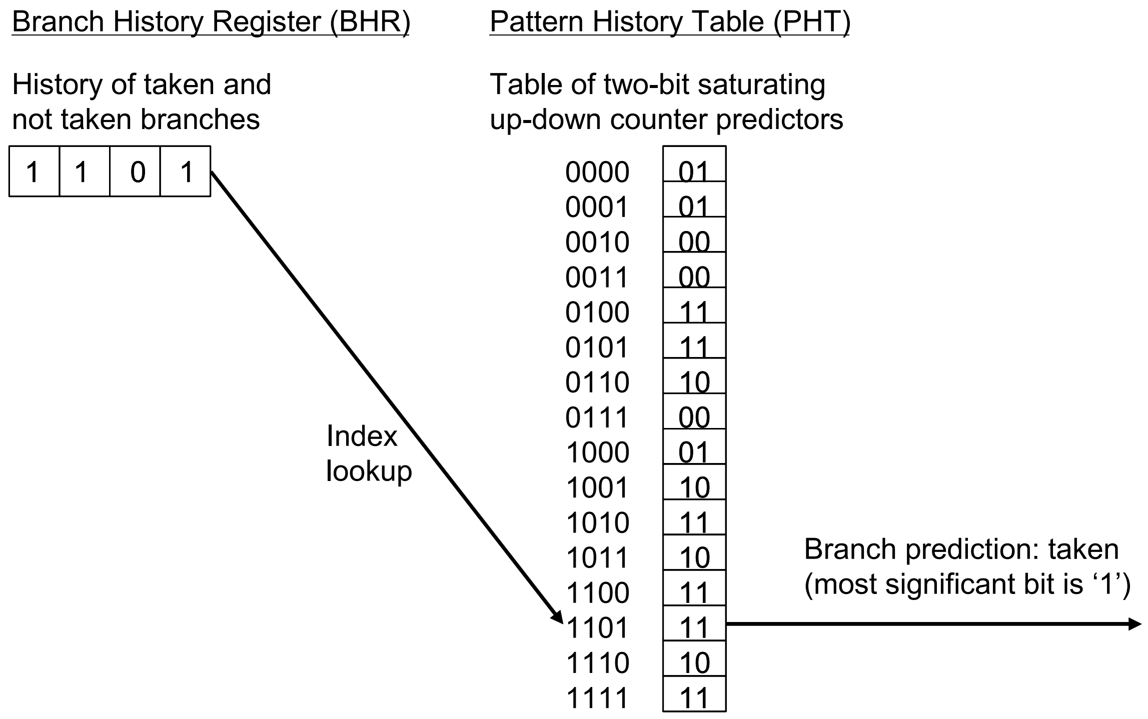
Branch History Register (BHR)
History of taken and not taken branches
1 1 0 1
Pattern History Table (PHT)
Table of two-bit saturating up-down counter predictors
0000 01
0001 01
0010 00
0011 00
0100 11
0101 11
0110 10
0111 00
1000 01
1001 10
1010 11
1011 10
1100 11
1101 11
1110 10
1111 11
Index lookup
Branch prediction: taken (most significant bit is '1')

Figure 1. Example of 4-bit Global Two-level Adaptive Branch Predictor making a branch prediction. In this example, since the branch history register (BHR) is 4 bits, the pattern history table (PHT) must have 2^4 = 16 entries.

For example, as shown in Figure 1, if the BHR were 4 bits wide, the PHT would have 2^4 = 16 entries. Suppose that each entry in the PHT contains 2 bits with initial value of 01, and that the last two times the pattern 1101 showed up in the BHR, the branch was taken. Then the 1101_2-th entry of the PHT will contain 11 and the next prediction when the BHR has the pattern 1101 will be predicted taken.

The BHR and PHT are updated with the predicted branch direction to make the next branch prediction, as shown in Figure 2. The 2-bit counter associated with the prediction is incremented (if the prediction was “not taken” then the counter would be decremented). The global branch history register is updated by shifting the entries to the left, and adding the most recent branch’s predicted behavior on the right in order to be ready to make the next branch prediction immediately. The high accuracy of the branch predictor means that it is far better to update the tables with the predicted branch behaviors. However, this policy means that on mispredictions the BHR and PHT entries need to be recovered.

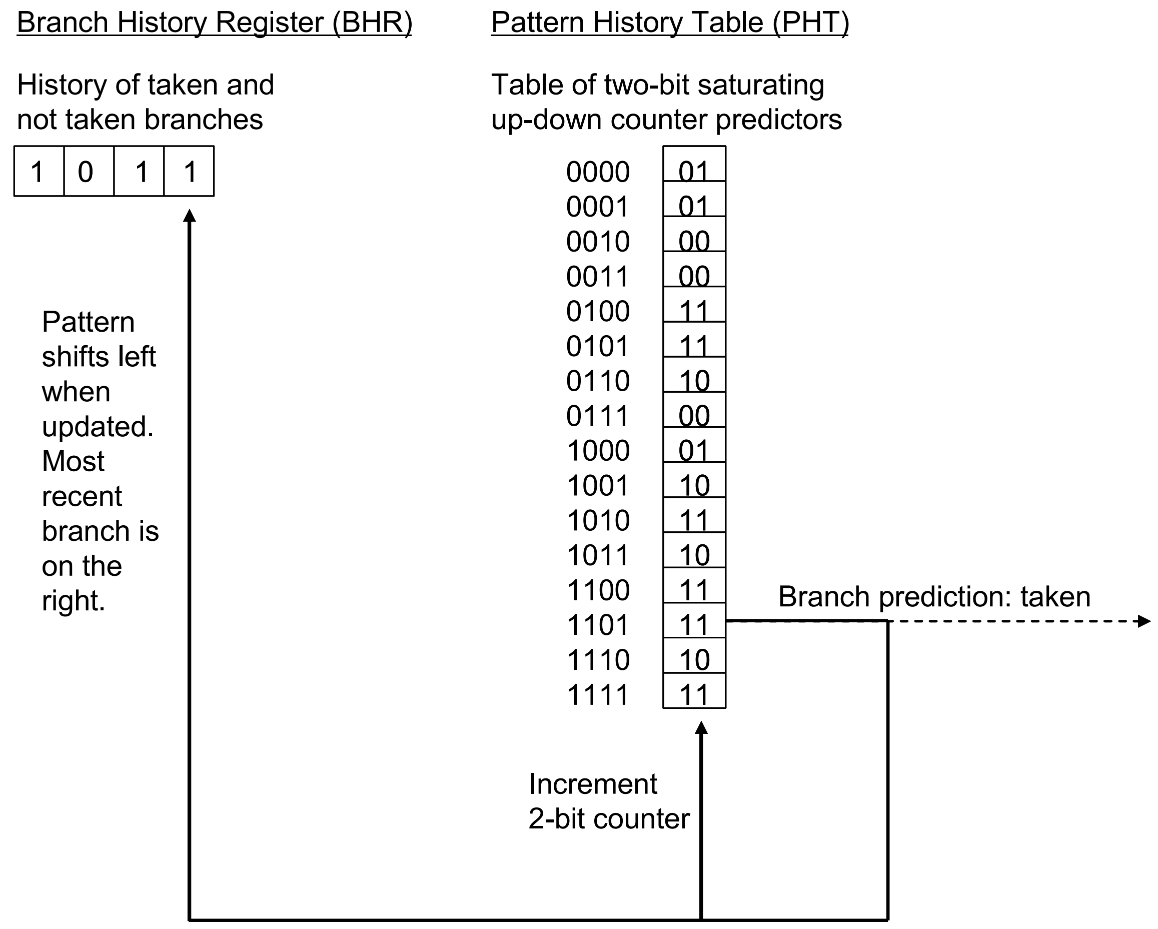
Branch History Register (BHR)
History of taken and not taken branches
1 0 1 1
Pattern shifts left when updated. Most recent branch is on the right.
Pattern History Table (PHT)
Table of two-bit saturating up-down counter predictors
0000 01
0001 01
0010 00
0011 00
0100 11
0101 11
0110 10
0111 00
1000 01
1001 10
1010 11
1011 10
1100 11
1101 11
1110 10
1111 11
Branch prediction: taken
Increment 2-bit counter

Figure 2. Example of a 4-bit Global Two-level Adaptive Branch Predictor updating the branch history register (BHR) and pattern history table (PHT).

Based on the source of the first-level branch history, Two-level Adaptive Branch Prediction has three classes of variations: global history schemes (as described in the previous example), per-address history schemes, and per-set history schemes.

1. Global history schemes (also called Correlation Branch Prediction [9]) use a single BHR, called the Global BHR, to record the history of all branches. The pattern in this Global BHR is used to look up the PHT prediction. The prediction of a conditional branch is influenced by the history of other branches.
2. Per-address history schemes use one BHR per static branch; therefore, multiple BHRs are used in the scheme. The prediction of a conditional branch is influenced by the history of the branch itself.
3. Per-set history schemes use one BHR to record the history of a set of adjacent static branches. The prediction of a conditional branch is influenced by the history of the branches in the same set, not just the branch itself.

2.1.2 Instruction Supply

In Yeh and Patt [8] an instruction supply mechanism was introduced where up to one basic block per cycle can be fetched by predicting branch targets in back-to-back cycles. We summarize a few details of the mechanism in this section, but the interested reader is referred to the original paper for more details. We will use the term "fetch address" to be the address used to fetch a sequence of instructions from the instruction cache. Three things are done at the same time: the instruction cache access, the branch address cache access, and the branch path prediction. The fetch address is used for both the instruction cache access and the branch address cache access from which a fall-through address, target address, and branch type are retrieved (conditional, unconditional, or call/return).

If the instructions fetched include a branch, those instructions up to and including the branch instruction comprise one basic block. Instructions after the branch are not issued to the processor until the next branch prediction is made.

If the fetch address misses in the branch address cache, then either there is no branch in the sequence of instructions fetched, or the sequence is being fetched for the first time. In either case, the fetch address is incremented by the fetch size, and the hardware continues fetching the next sequential block of instructions. In the event that a branch instruction is discovered after the instructions are decoded, the fall-through address, target address, basic block size, type of branch, and branch path are recorded in the branch address cache.

If the fetch address hits in the branch address cache, then we know that there is a branch somewhere in the sequence of instructions just fetched. Since the information from the branch address cache is available at the same time that the instructions are fetched from the instruction cache, a new fetch address (either the fall-through address or the taken address) can be determined immediately. The next instruction cache and branch address cache accesses begin on the next cycle.

2.2 Fetching Multiple Basic Blocks Each Cycle

The performance of the mechanism described Section 2.1.2 limited the fetch capacity to one basic block per cycle. Since only one branch path prediction and only one set of consecutive instructions could be fetched from the instruction cache per cycle, instruction fetch stopped when a branch was encountered. This was due to the limitation of a single prediction per cycle and limitations in the instruction cache configuration.

Fetching multiple basic blocks each cycle requires more than a multiple branch prediction algorithm. At the same time that multiple branch paths are being predicted, the addresses of the basic blocks following those branches must be determined. In addition, the instruction cache must be able to supply multiple non-consecutive blocks of instructions in a single cycle. Our solutions to these issues are:

- The Multiple Branch Two-level Adaptive Branch Predictor which provides highly accurate predictions for multiple branch paths.
- The Branch Address Cache (BAC) which is a hardware structure to provide multiple fetch addresses of the basic blocks following each branch.
- An instruction cache with enough bandwidth to supply a large number of instructions from non-consecutive basic blocks.

In this chapter we will describe the mechanisms for fetching two and three basic blocks each cycle. The mechanisms described can be easily extended to more than three branches, but the hardware cost increases exponentially with each additional basic block.

2.2.1 The Multiple Branch Two-Level Adaptive Branch Predictor

The prediction algorithm for a single branch per cycle described in Section 2.1.1 can be extended to two branch predictions per cycle. We will henceforth identify the first branch as the *primary branch*, the second branch as the *secondary branch*, and the third branch as the *tertiary branch*.

The *primary basic block* is the basic block dynamically following the primary branch, i.e., the basic block containing a secondary branch. There are two possibilities for the primary basic block: The target and the fall-through basic blocks of the primary branch. These will be denoted as T or N, depending on whether the primary branch was taken or not taken. The *secondary basic block* is the basic block following the secondary branch. The secondary basic block can be one of up to four different blocks depending on the direction of the primary and the secondary branches. These will be denoted as TT, TN, NT, or NN, depending on whether the primary and secondary branches were taken-taken, taken-not taken, not-taken-taken, or not taken-not taken, respectively. Finally, the *tertiary basic block* is the one following the tertiary branch. The tertiary basic block can be one of up to 8 different blocks depending on the outcome of the primary, secondary, and tertiary branch paths, and its denotations are TTT, TTN, TNT, etc.

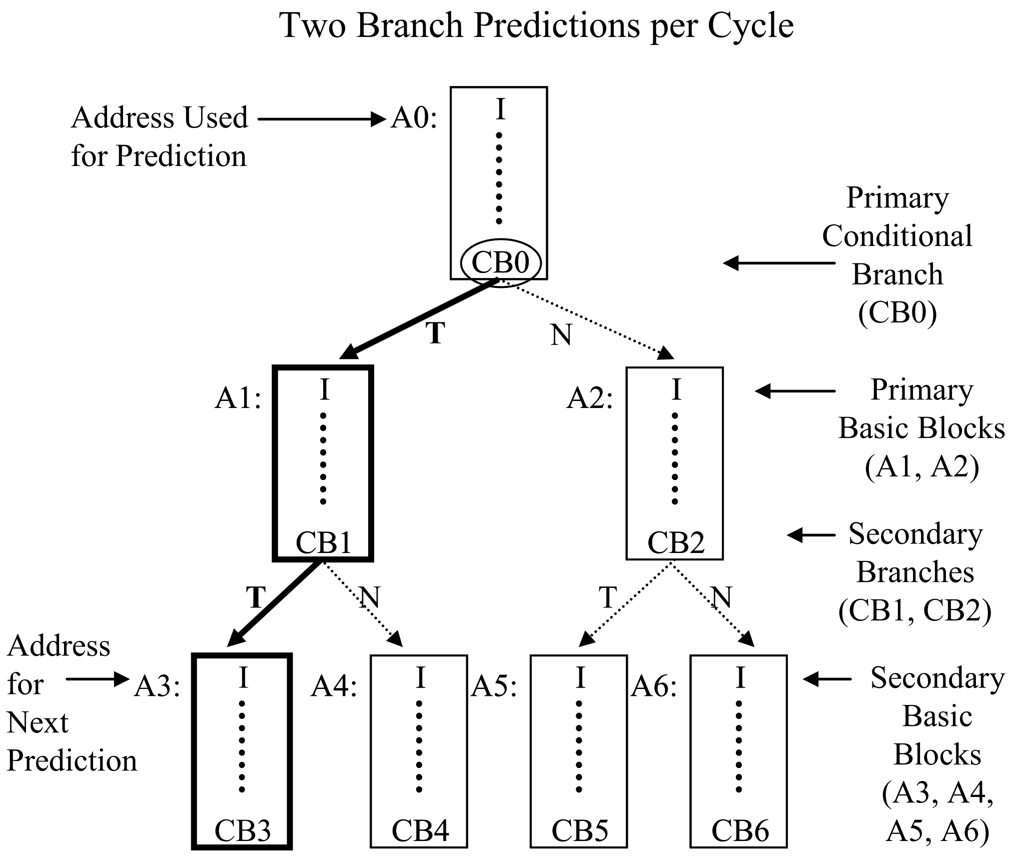
Two Branch Predictions per Cycle
Address Used for Prediction
A0:
I
CB0
Primary Conditional Branch (CB0)
T
N
A1:
CB1
A2:
CB2
Primary Basic Blocks (A1, A2)
Secondary Branches (CB1, CB2)
T
N
T
N
Address for Next Prediction
A3:
CB3
A4:
CB4
A5:
CB5
A6:
CB6
Secondary Basic Blocks (A3, A4, A5, A6)

Figure 3. Identification of the primary and secondary branches, and the primary and secondary basic blocks for a two-branch-per-cycle predictor.

Figure 3 shows the primary and secondary branches and the primary and secondary basic blocks for the case when two predictions are made per cycle. If the darker branch paths are predicted, the darker basic blocks are fetched.

Figure 4 adds the tertiary branches and the tertiary basic blocks. When three branch paths are predicted, then the address used for the next prediction would be in a tertiary basic block. The circled branches are the branches for which predictions were made in this example.

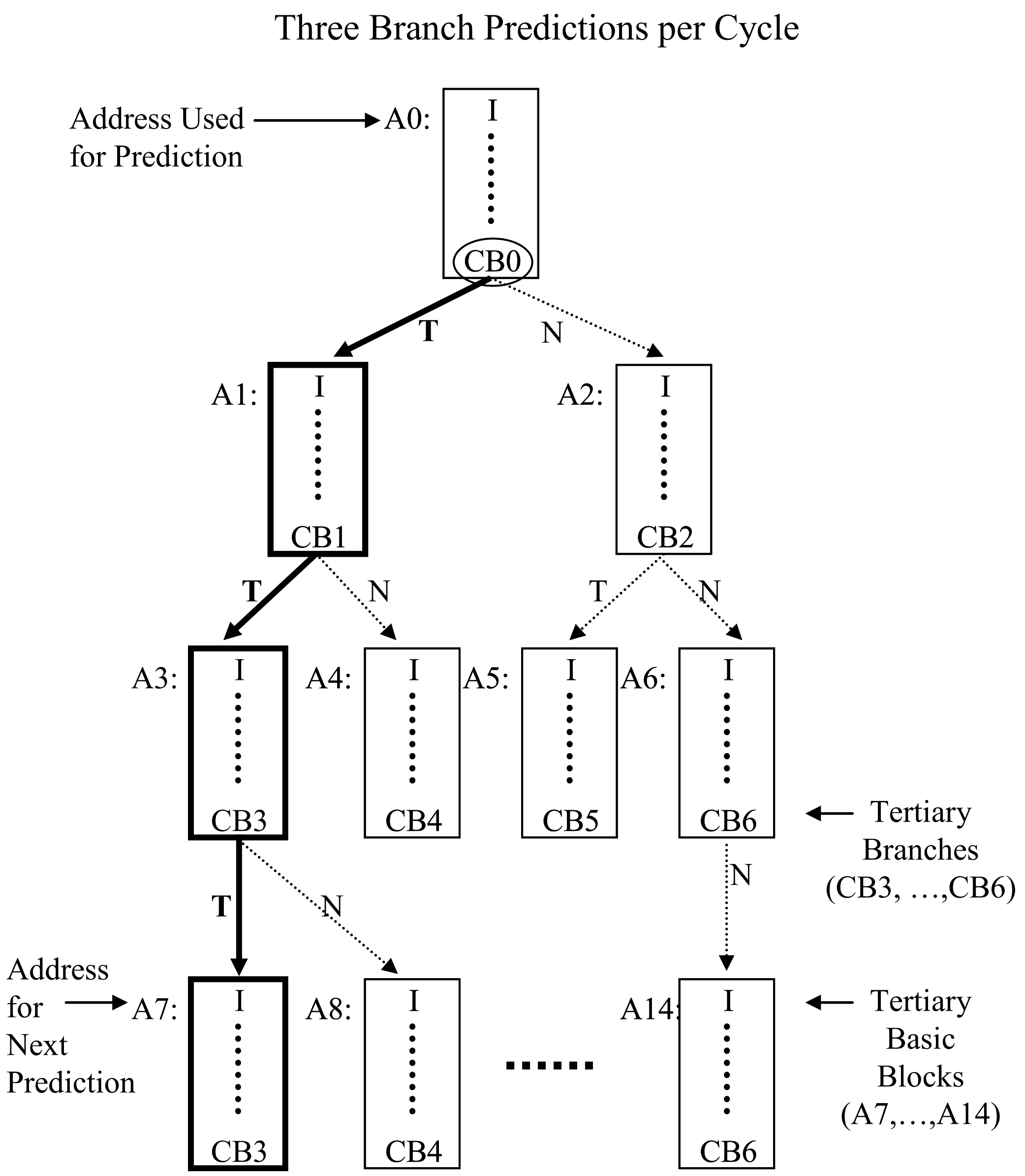
Three Branch Predictions per Cycle
Address Used for Prediction
A0:
I
CB0
T
N
A1:
A2:
CB1
CB2
T
N
T
N
A3:
A4:
A5:
A6:
CB3
CB4
CB5
CB6
Tertiary Branches (CB3, …,CB6)
T
N
N
Address for Next Prediction
A7:
A8:
A14:
CB3
CB4
CB6
Tertiary Basic Blocks (A7,…,A14)

Figure 4. Identification of the tertiary branches and tertiary basic blocks for a three-branch-per-cycle predictor.

The multiple branch prediction algorithm introduced in this chapter is modified from the global history schemes of the Two-level Adaptive Branch Prediction described in

[7] and summarized in Section 2.1.1. The modified global history schemes not only make the prediction of the immediately-following branch, but predict subsequent branches. The per-address history and per-set history schemes of Two-Level Adaptive Branch Prediction, on the other hand, require more complicated BHT access logic for making multiple branch predictions in each cycle, because they may require many different branch histories to make predictions for different branches. In order to simplify the BHT design, we consider only the global history schemes in this chapter.

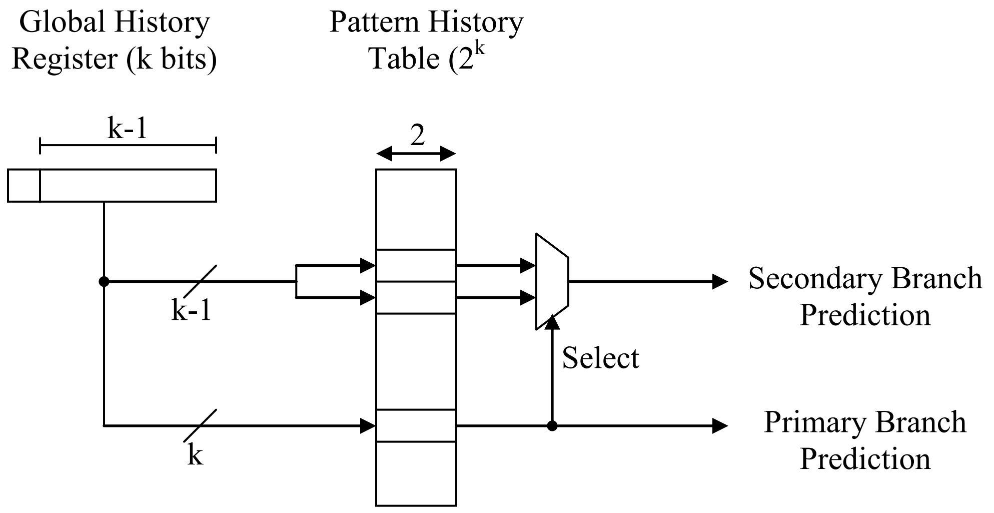
Global History Register (k bits)
Pattern History Table (2^k
k-1
2
k-1
k
Select
Secondary Branch Prediction
Primary Branch Prediction

Figure 5. Algorithm to make 2 branch predictions from a single branch history register.

The first multiple branch prediction variation is called *Multiple Branch Global Two-level Adaptive Branch Prediction using Global Pattern History Table* (MGAg). This scheme uses a global history register of k bits and a global pattern history table of 2^k entries, each entry containing 2 bits. The k bits in the history register record the outcome of the last k branches. The history register is updated speculatively with the predicted branch outcomes and corrected later in the event of an incorrect prediction, because the prediction accuracy is expected to be high. The right-most bit corresponds to the prediction of the most recent branch, and the leftmost bit corresponds to the prediction of the oldest branch.

As shown in Figure 5, all k bits in the history register are used to index into the pattern history table to make a primary branch prediction. The 2-bit counter value read from the pattern history table entry is used to make the prediction, just as in the single branch Adaptive Two-level Branch Predictor.

To predict the secondary branch, the right-most k-1 branch history bits are used to index into the pattern history table. Note that since we are missing one binary digit, k-1 bits would address two adjacent entries in the PHT, which are resolved by using the primary branch prediction. In other words, the primary branch prediction is used to select one of the two entries to make the secondary branch prediction.

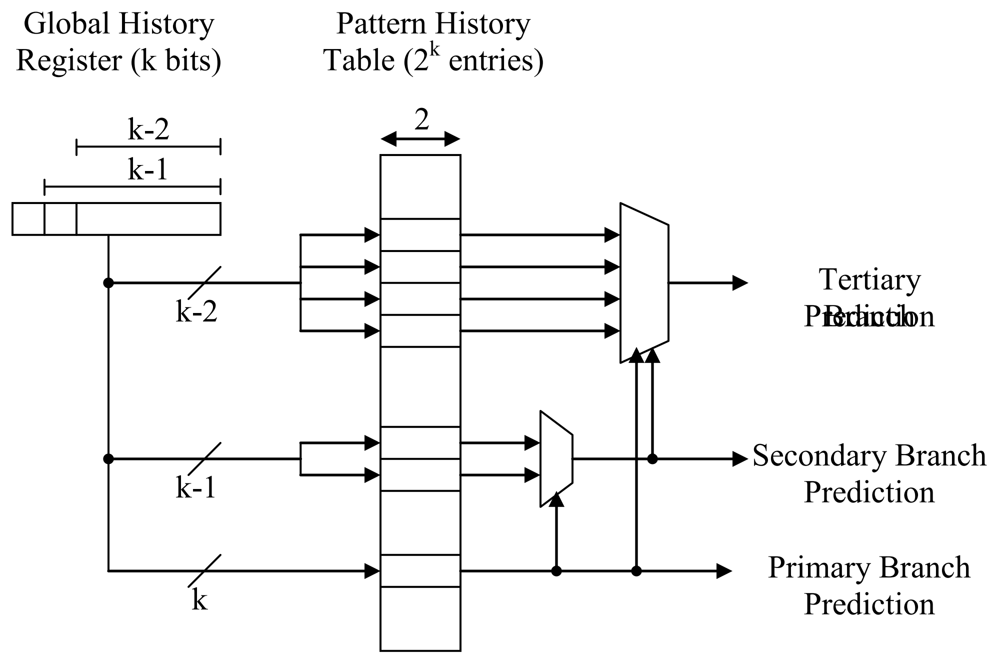
Global History Register (k bits)
Pattern History Table (2^k entries)
k-2
k-1
2
k-2
k-1
k
Tertiary Branch Prediction
Secondary Branch Prediction
Primary Branch Prediction

Figure 6. Algorithm to make 3 branch predictions from a single branch history register.

Finally, as shown in Figure 6 the tertiary prediction uses the right-most k-2 history register bits to address the pattern history table and access four adjacent entries. The primary and secondary predictions are used to select one of the four entries for the tertiary branch path prediction. This algorithm allows each of the multiple branch path predictions to take full advantage of the k bits of branch history. Longer history registers increase the prediction accuracy, and as multiple branches are predicted, the accuracy becomes increasingly important.

The second multiple branch prediction variation is called Multiple Branch Global Two-Level Adaptive Branch Prediction using Per-set Pattern History Tables (MGAs). It differs from the previous scheme in that there are multiple pattern history tables. The pattern history tables are associated with the primary branches. Similar to MGAg, all k bits are used to index into a pattern history table to make a prediction for the primary branch. The pattern history table is selected based on the fetch address corresponding to the primary branch. The second prediction is made from the same pattern table since the address of the secondary branch is not known at the time of the prediction. This scheme attempts to limit the amount of pollution in the pattern history tables by different branches, but may result in less accurate secondary and tertiary branch predictions.

The extreme case of the MGAs scheme is when there is a separate pattern history table associated with each branch. This scheme is called Multiple Branch Global Two-level Adaptive Branch Prediction using Per-address Pattern History Tables (MGAp).

The pattern table entries are updated after the branch instructions are resolved, which could take several cycles. Therefore the pattern table entries are always somewhat out-of-date. This is likely to degrade the accuracy of the multiple branch prediction algorithm more than the accuracy of a single branch prediction algorithm. The reason the branch may take several cycles to resolve is that it may have to wait for a condition to be evaluated or an address to be computed which may take several cycles due to data dependencies.

Since the branch predictions are done at the same time the instructions are fetched, the determination of whether there is a branch in a fetch sequence is done through the

Branch Address Cache which is described in detail in the next section. If the fetch address hits in the Branch Address Cache, then there is a branch in the sequence being fetched. Otherwise no branch is assumed and the instruction fetch mechanism fetches down the sequential stream.

The branch path predictions made with the Multiple Branch Two-level Adaptive Branch Predictor are done at the same time the Branch Address Cache and instruction cache are accessed. These branch path prediction bits are used to select the fetch addresses that are needed for the next cycle from the possible fetch addresses provided by the Branch Address Cache. For now, we will merely state that if two predictions are made, then two fetch addresses are selected. If three predictions are made, then three fetch addresses are selected.

Multiple predictions might not be made every cycle for several reasons. The first case is when a basic block is very large, so the entire instruction cache bandwidth may be devoted to fetching the basic block. Fetching the primary basic block has higher priority than fetching secondary or tertiary basic blocks. Therefore if we cannot fetch one basic block in its entirety with its instruction cache bandwidth quota, then we allow it to usurp the bandwidth quota from a subsequent block.

If a secondary or tertiary basic block's bandwidth is usurped, the prediction of the branch in that basic block is delayed until the cycle when it is actually being fetched. At that point it becomes the primary branch and a (different) secondary and tertiary branch may be predicted along with it.

The other case when multiple branch path predictions are not made is when the branch is a return instruction. The return instruction's predicted target address is obtained from the return address stack. The next branch is difficult to predict because the return may direct the instruction stream to any number of locations.

2.2.2 The Branch Address Cache (BAC) Design

With each of the MGAg, MGAs, and MGAp algorithms, we use a Branch Address Cache (BAC) to store the addresses to which the branches may direct the instruction flow. Recall that with the single basic block instruction supply algorithm summarized in Section 2.2 the branch address cache is indexed by the fetch address, from which two potential fetch addresses are obtained (one for the target block and one for the fall-through block). The branch prediction chooses between the two addresses.

The multiple basic block supply algorithms use a similar BAC. The fetch address is used to access the BAC. This is done in parallel to the instruction cache access. Although there may be two or three fetch addresses accessing the instruction cache simultaneously, *only a single fetch address is used to access the BAC*. If only one basic block is being fetched, that fetch address is used. If two basic blocks are being fetched simultaneously, the second fetch address is used to access the BAC. If three basic blocks are fetched, the third fetch address is used.

If the fetch address hits in the BAC, there is a branch in the sequence of instructions just fetched. The BAC entry records the branch type (conditional, unconditional, or return) and the target and fall-through basic block starting addresses of the primary branch. The same entry also contains the branch type and fetch addresses of basic blocks for each of the expected number of branches for which we will make predictions, and all the known potential fetch addresses of their targets. If the number of basic blocks predicted and fetched per cycle is limited to 2, we get 6 fetch addresses: 2 for the two primary basic block addresses, and 4 for the four possible secondary basic blocks. If the basic block prediction and fetch limit is 3, we get 14 possible fetch addresses: 2 for the primary basic blocks, 4 for the secondary, and 8 for the tertiary basic blocks.

Each entry in a 512-entry, 4-way set associative Branch Address Cache which supports two branch predictions per cycle has the following fields: TAG, P_valid, P_type, Taddr, Naddr, ST_valid, ST_type, TTaddr, TNaddr, SN_valid, SN_type, NTaddr, NNaddr, where each field contains:

- TAG field – The 23 high-order bits of the primary fetch address. A “BAC hit” occurs if the tag matches with the upper address bits of the current fetch address and the primary branch is valid.

- Valid bits – The valid bits for the corresponding branch entries. P refers to the primary branch, ST refers to the secondary branch if the primary branch is taken, and SN is the secondary branch if the primary branch is not taken.

- Type fields – The branch type of the corresponding branch. The type can be conditional, unconditional, or return. Each type field consists of 2 bits.

- Addr fields – The address of the corresponding basic block. Each address field consists of 30 bits.

A BAC supporting two branch predictions per cycle would have a total of 212 bits per entry. A BAC supporting three branches would have an additional eight address fields and four additional valid bits for the four possible tertiary branches, making each entry 464 bits wide.

When a fetch address misses in the BAC, a large basic block is assumed and the entire instruction cache bandwidth is devoted to fetching sequential instructions. If a branch is discovered once the instructions are decoded and the branch is predicted taken or is an unconditional branch, the prefetched instructions after the branch are discarded. The address of the fall-through and target addresses are calculated in the cycle after decode. The branch is then allocated a primary branch entry in the BAC. The higher order bits of the fetch address are entered in the tag field, the primary branch valid bit is set, the secondary (and tertiary) branch valid bits are cleared, and primary fall-through and target addresses are entered. If the branch is an indirect branch, however, the target address is not calculated until the operands are ready, and the valid bit is not set until that time.

The branch will also be entered as a secondary branch in the BAC entry of the previous branch if:

- the previous fetch address had a valid primary branch entry in the BAC but did not predict a secondary branch *and*
- the basic block of the previous fetch address was not oversized (i.e., there was enough instruction cache bandwidth for another basic block fetch) *and*
- the previous branch was not a return.

2.2.3 The Instruction Cache

The ability of the instruction cache to provide enough instructions becomes critical when multiple possibly non-consecutive basic blocks are fetched each cycle. The instruction cache must have high bandwidth, low miss rate, and the ability to fetch from multiple addresses in parallel.

To satisfy the high bandwidth requirement, the cache must either have a large number of banks, or have wide banks. Also, due to off-chip bandwidth and pin limitations, the instruction cache should be on-chip.

The ability to fetch from multiple addresses in parallel implies a cache with either interleaved or multi-ported banks, or both. With interleaved banks, each independently addressable, multiple fetch addresses can access the instruction cache simultaneously provided that their accesses are not to the same bank. If there is a bank conflict, priority is given to the earlier (relative to the dynamic instruction stream) fetch address. Therefore it is important to have enough banks to make the probability of bank conflicts low.

A multi-ported cache eliminates the bank conflict problem. For example, a dual-ported cache allows the simultaneous access of two fetch addresses, and a tri-ported cache allows the simultaneous access of three fetch addresses. Unfortunately, multi-ported memories are expensive in silicon chip area.

It is critical for the instruction cache miss rate to be low. Each instruction cache miss stalls the fetch sequence. Since multiple basic blocks can be fetched each cycle, the opportunity cost can be (up to) the number of cycles it takes to service the miss multiplied by the number of instructions that could have been fetched during those idle fetch cycles. Also, since more instructions are fetched each cycle, there are fewer cycles between instruction cache misses. Therefore more time is spent waiting for instruction cache misses to be satisfied. Commonly used ways to minimize instruction cache miss rates are to increase the associativity, to increase the size of the cache, and to prefetch instructions.

We chose several cache configurations which gave us reasonably high bandwidth, the ability to fetch multiple addresses in parallel, and a relatively low miss rate. Most of our simulations were done with a 32K cache which was 2-way set associative with 8 interleaved single-ported banks, each bank having a line size of 16 bytes. Each fetch address can access two banks so that we guarantee between 5 and 8 instructions per fetch address (due to basic block alignment). This configuration and several others are compared in Section 2.4.

2.3 Simulation Methodology

2.3.1 Simulation Environment

We used a trace-driven simulator to evaluate the performance of a machine front-end which implements the Multiple Branch Two-level Adaptive Branch Predictor, a 512-entry 4-way set associative Branch Address Cache (BAC), and a high-bandwidth instruction cache. Unless otherwise specified, the instruction cache configuration used was 32K bytes, 2-way set associative, 8-way interleaved, single-ported, and with a line size of 16 bytes (4 instructions).

For the multiple basic block mechanisms, we can fetch two cache lines (a maximum of 8 instructions) per basic block fetch address because most basic blocks contain 4 to 8 instructions. In order to do a fair comparison, we allow the single basic block prediction and fetch algorithm to fetch up to 4 cache lines. The maximum number of instructions issued, passed to the back-end of the machine, is limited to 16 instructions per cycle.

The benchmarks written in C were compiled with the Motorola Apogee C compiler for the Motorola 88100 instruction set and the ones written in Fortran were compiled with the Green Hill Fortran compiler. A Motorola 88100 instruction level simulator generated the instruction traces. The first 50 million instructions from each trace were used rather than the entire trace due to simulation time constraints.

Nine benchmarks were selected from the SPEC89 benchmark suite. These included 4 integer and 5 floating point benchmarks. The integer benchmarks were li, gcc, eqntott, and espresso. The floating point benchmarks were doduc, fpppp, matrix300, spice2g6, and tomcatv. The figures included in the result section have the abbreviations listed in Table 1 for the various benchmarks. Table 1 also shows the average basic block size of the first 50 million instructions of each benchmark.

	Benchmark	Abbreviation	Average Basic Block Size (instructions)
Integer	eqntott	Eq	4.76
	espresso	Es	3.41
	gcc	Gc	4.94
	li	Li	4.14
Floating Point	doduc	Dd	10.46
	fpppp	Fp	57.01
	matrix300	Mt	28.20
	spice2g6	Sp	5.36
	tomcatv	Tc	26.33

Table 1. Benchmark list and average basic block size.

The MGAg, MGAs, and MGAp are parameterized according to the history register length and the number of Pattern History Tables. These parameters will be given as: HhPp, where h is the number of bits in the Global History Register, and p is the number of pattern history tables.

2.3.2 Performance Metric

Since the simulator only models the front end of a machine, we use a new metric, IPC_f (instructions per cycle fetched) to evaluate the performance of an instruction fetch mechanism. IPC_f measures the effective number of instructions fetched per cycle by an instruction fetch mechanism. To derive IPC_f, we assume the machine stalls or wastes cycles for various reasons from the instruction fetch mechanism but not from the rest of the machine, so the instructions issued can be executed without stalling the machine front end. Moreover, only effective instructions are counted;

instructions fetched down the incorrectly predicted paths are not counted. The machine front end could waste cycles due to the following reasons:

- Instruction cache misses
- Incorrect branch predictions which include incorrect branch path predictions and incorrect fetch address predictions
- Branch Address Cache misses on taken branches

Since we do not simulate the rest of the machine, the exact mispredicted branch penalty is approximated. A 6 cycle mispredicted branch penalty is assumed; therefore, the instructions following an incorrectly predicted branch will not be fetched until 6 cycles after the branch is fetched. The I-cache miss penalty is assumed to be 10 cycles. We also show how the machine performance changes as the branch misprediction penalty and I-cache miss penalty are varied.

2.4 Simulation Results

2.4.1 Effect on Prediction Accuracy and IPC_f of History Register Length

Figure 7 shows how the prediction accuracy changes as we increase the number of bits in the global history register of the MGAg scheme for two branch predictions per cycle. The prediction accuracy is the number of correctly predicted branches over the total number of branches in the dynamic instruction stream. Longer branch histories give better prediction accuracy which is reflected in the rising curves. The hardware cost goes up exponentially with the number of history bits due to the number of pattern history table (PHT) entries required.

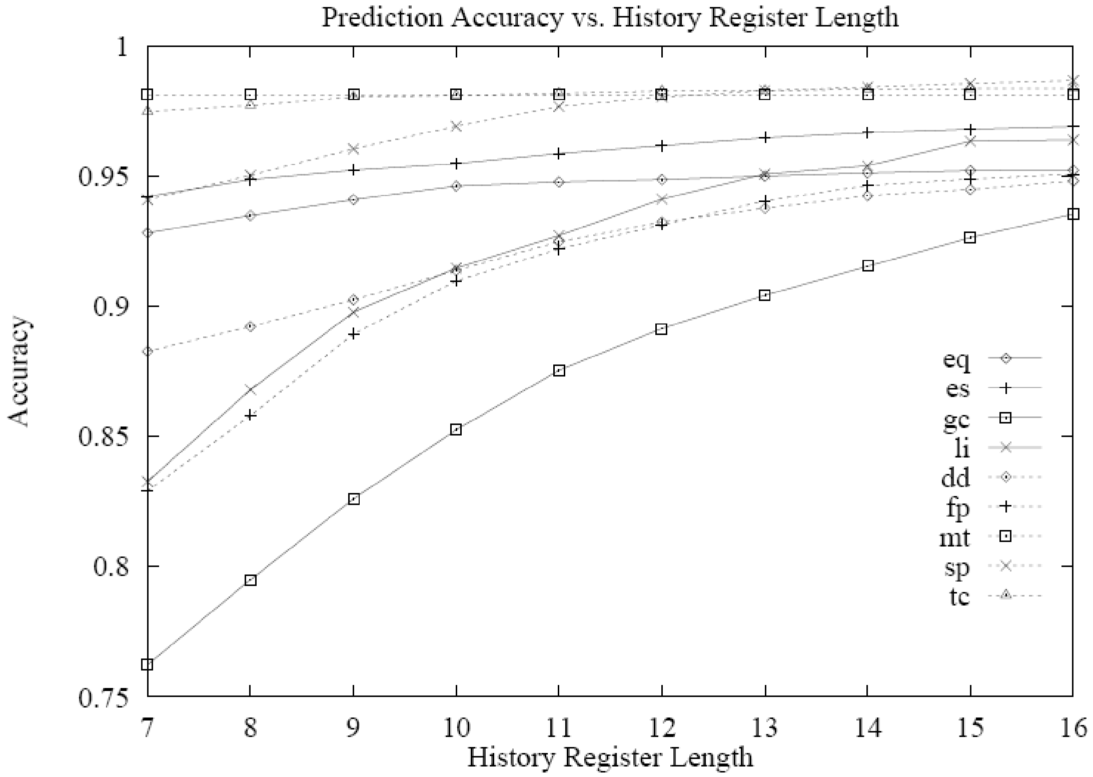

Figure 7. Variation of the size of the global branch history register.

The prediction accuracies varied between 91.5 and 98.4% for a branch history register (BHR) length of 14 bits, and between 93.5 and 98.7% for a history register length of 16 bits. The knees of the curves for most benchmarks are reached at a BHR length of 14 bits. We used a 14-bit BHR length for the other experiments reported in this chapter. A 14-bit BHR length means that a PHT has 2^{14} X 2 bits, or 32K bits.

2.4.2 Tradeoff Between the Number of Pattern History Tables and History Register Length

We simulated several MGAg, MGAs, and MGAp configurations to determine how the performance accuracy changes with the number of PHTs for two branch predictions per cycle. Figure 8 for integer benchmarks and Figure 9 for floating point benchmarks show the branch prediction accuracy for 1 to 512 PHTs. Each configuration shown has the same hardware cost, which was achieved by decreasing the number of entries in each PHT as the number of PHTs is increased. Since the entries in the PHTs are addressed by the BHR, the BHR length is reduced when we decrease the number of entries in each PHT.

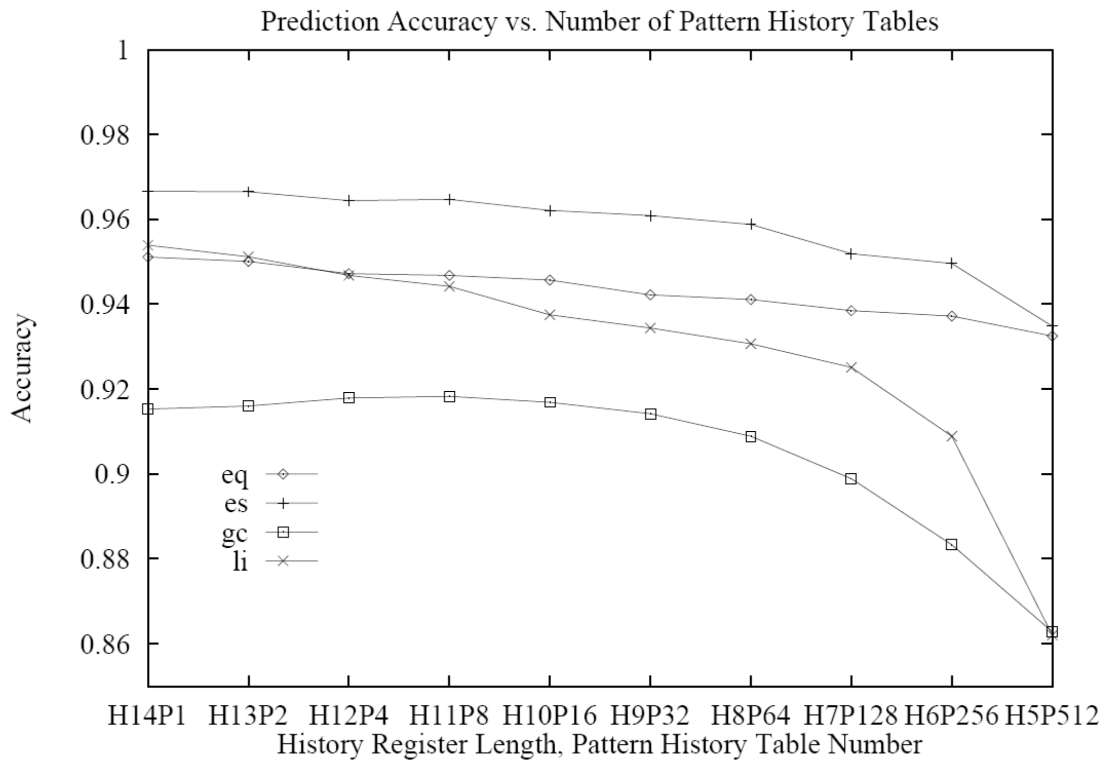

Figure 8. Variation of the number of the PHTs with the hardware cost held constant, for integer benchmarks.

The PHT used to make the predictions is determined by the primary branch address. The experiments shown in Figure 8 and Figure 9 used the branch address starting at bit 10 to select a PHT. This allows branches within the same 256-instruction block in the static code to map to the same PHT.

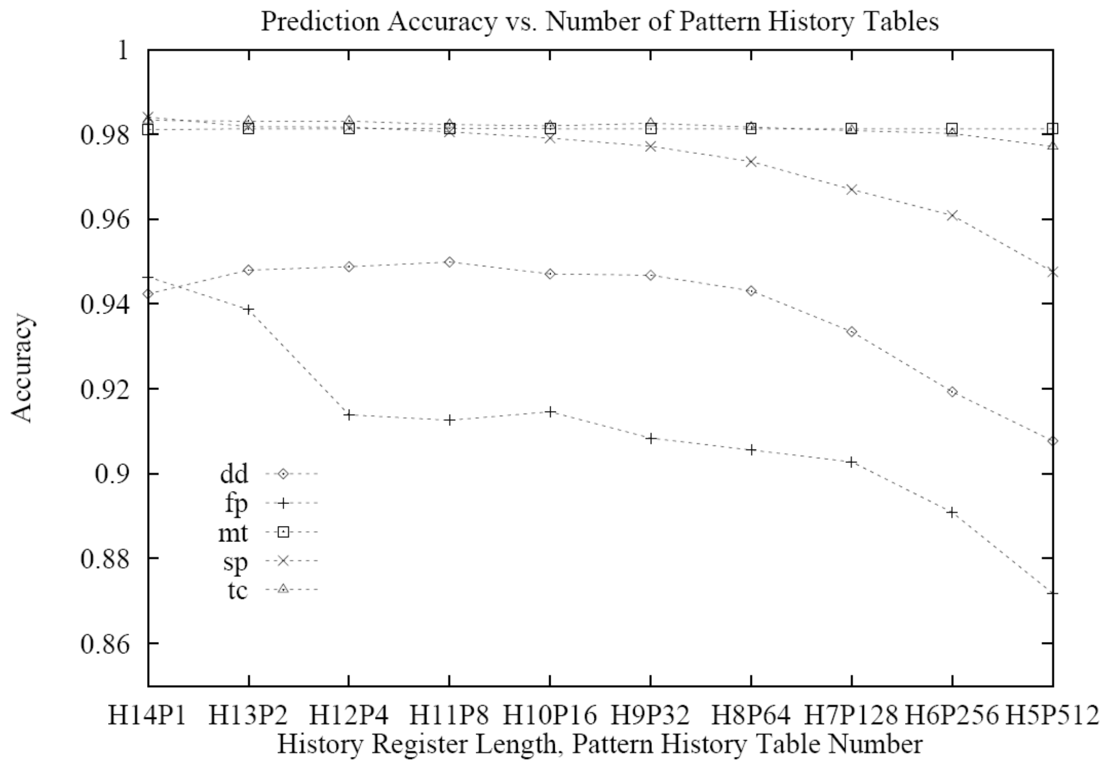

Figure 9. Variation of the number of the PHTs with the hardware cost held constant, for floating point benchmarks.

The prediction accuracies shown in Figure 8 and Figure 9 tend to be higher for configurations with one to eight pattern history tables, then decreases when the number of pattern history tables is increased beyond 8. Longer branch history helps to increase the prediction accuracy. Increasing the number of PHTs reduces the interference between branches, but since the second branch is predicted using the PHT of the first branch, the probability of mapping two branches predicted together into different PHTs is higher when more PHTs are used.

2.4.3 Number of Branch Predictions per Cycle

Figure 10 shows the IPC_f increase with the number of branch predictions per cycle. The number of opportunities for multiple branch prediction is quite high despite the greater likelihood of bank conflicts in the instruction cache when three basic blocks are fetched.

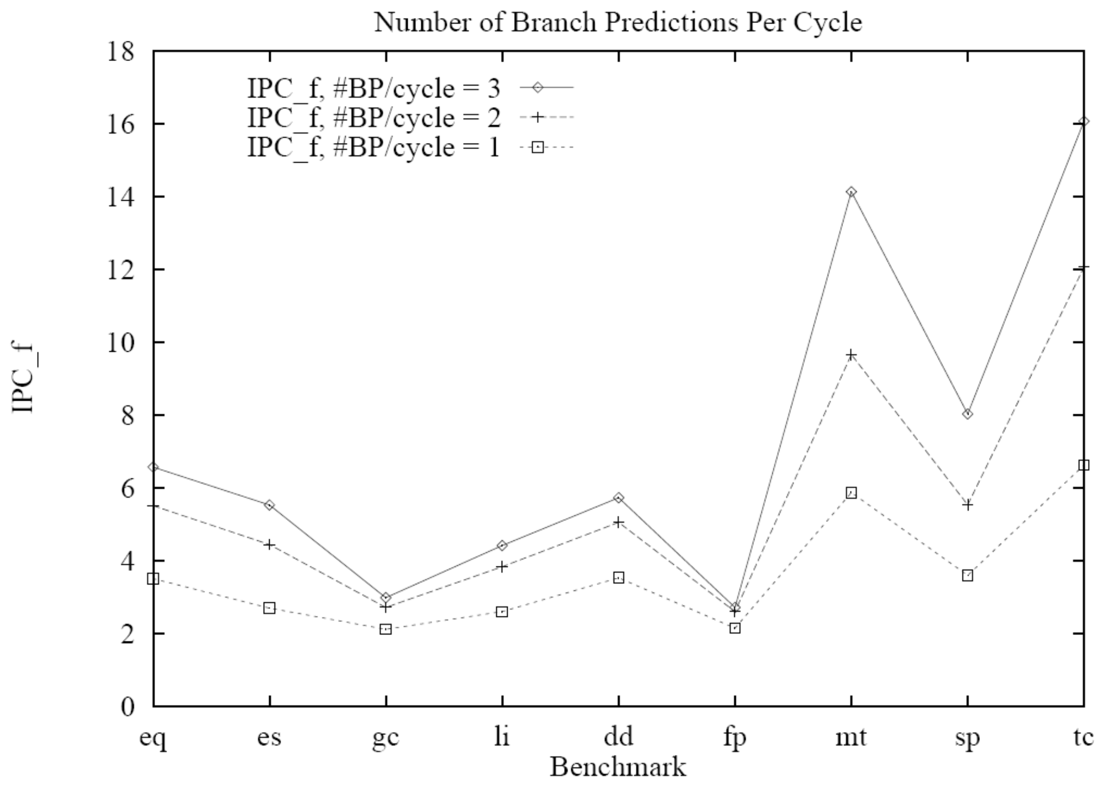

Figure 10. Instructions per cycle when 1, 2, and 3 branches are predicted each cycle.

The average IPC_f when one basic block is predicted per cycle is 3.0 and 5.6, for integer and floating point benchmarks, respectively. Two predictions per cycle increase this to 4.2 for integer and 7.1 for floating point. Three predictions per cycle increases IPC_f further to 4.9 for integer and 8.9 for floating point.

For the one and two predictions per cycle experiments we allowed a maximum of 16 instructions to be fetched from the instruction cache per cycle. For the three predictions per cycle experiments we increased the instruction cache bandwidth to 24 instructions in order to accommodate the 3 fetch addresses. To cap the number of instructions issued, we constrained the issue width to 16 instructions for all three cases. The larger instruction cache bandwidth allows more instructions to be fetched per cycle, which affects the performance of floating point programs more than integer programs because of the high branch prediction accuracy and large basic block size of floating point benchmarks. This effect results in the significant floating point performance increase when going from two to three predictions per cycle.

The application fpppp (abbreviated fp in the graph) does not show significant performance increase when going from one to two to three predictions per cycle due to the repeated execution of an extremely long sequential code segment which causes the instruction cache to thrash. The instruction cache miss penalty dominates its performance.

Integer programs show noticeable performance increase except for gcc which is dominated by incorrect branch predictions.

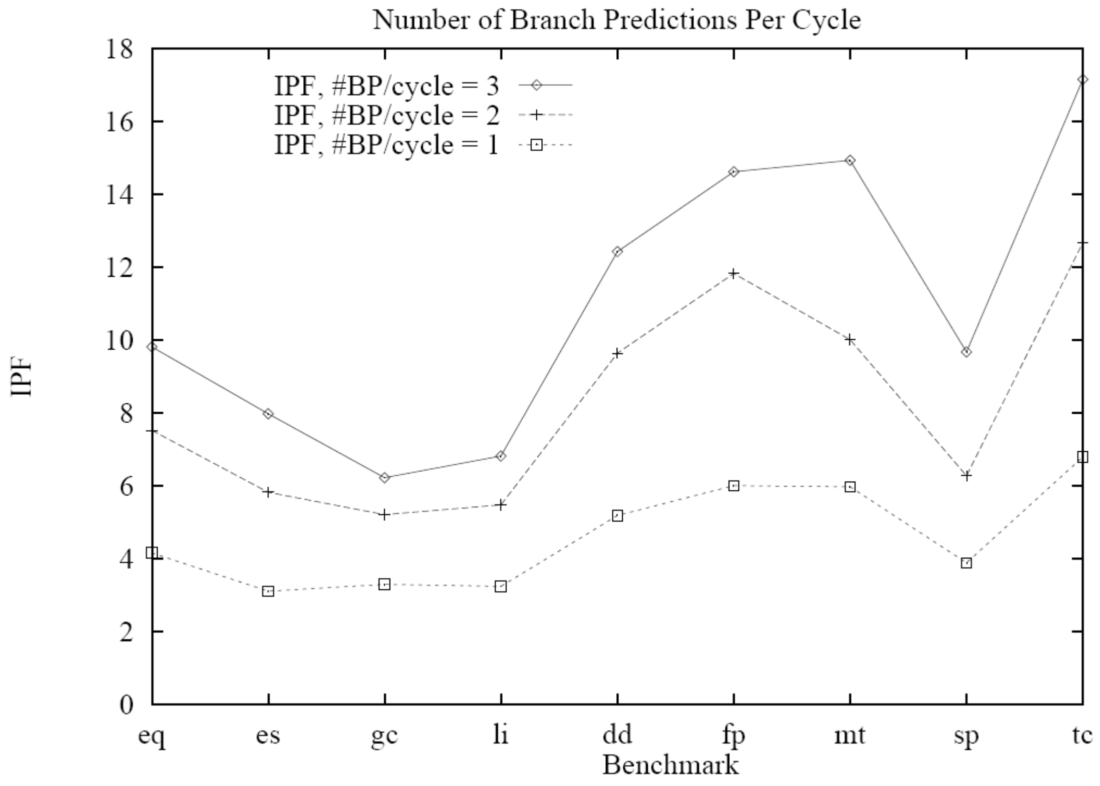

Figure 11. Instructions per fetch when 1, 2, and 3 branches are predicted each cycle.

Figure 11 shows the IPF, instructions per fetch, for the benchmarks as the number of branch predictions and basic block fetches of 1, 2, and 3 per cycle. An efficient instruction fetch mechanism should attain an IPC_f as close to the IPF as possible. The discrepancy between IPF and IPC_f is due to the branch misprediction penalty, BAC misses, and instruction cache miss penalty.

2.4.4 Branch Prediction Efficiency

Now we look at how often we use the unique ability of our multiple branch predictor to actually predict multiple branches per cycle. We call this the multiple branch prediction utilization. Table 2 shows the data for the case where 2 basic blocks can be predicted and fetched each cycle. We count the percentage of cycles when zero, one, and two branches were predicted. Zero branches are predicted if we are fetching a long sequential segment of code, or if the fetch address misses in the Branch Address Cache, and a branch is found in the sequence of instructions after the instructions are decoded. The application fpppp has a high percentage of cycles with no predictions due to the extremely long sequential code segment which is repeatedly executed. The percentage of cycles when zero predictions were done per cycle is 10% per cycle for integer and 44% for floating point.

Only a single branch is predicted when the primary branch is a return, or the primary basic block is large (oversized) in which case the instruction fetch bandwidth of the secondary basic block is usurped. About 24% of the single basic block fetches are due to oversized basic blocks, and about 5% are due to the primary branch being a return. Two branch predictions are made and two basic blocks are fetched 62% of the time for integer and 24% of the time for floating point benchmarks.

Table 2. Branch prediction utilization of an instruction fetch mechanism which is able to provide fetch addresses of two basic blocks in each cycle.

Benchmark	No Prediction	One Prediction		Two Predictions
		Oversized	Return	
Eq	0.0839	0.1231	0.0272	0.7528
Es	0.0364	0.1125	0.0145	0.8317
Gc	0.1843	0.3634	0.0522	0.3844
Li	0.0939	0.2518	0.1213	0.5244
Dd	0.3335	0.2580	0.0602	0.3364
Fp	0.7415	0.1909	0.0120	0.0550
Mt	0.3386	0.3335	0.0042	0.3236
Sp	0.2145	0.2142	0.1613	0.4081
Tc	0.5893	0.3337	0.0006	0.0751

Table 3. Percentage of fetches causing the instruction fetch mechanism to stall.

Benchmark	No Delay	Decode Delay	Incorrect Branch Prediction	I-cache Miss	Bank Conflict
Eq	0.8924	0.0004	0.0679	0.0001	0.0392
Es	0.9207	0.0063	0.0565	0.0001	0.0164
Gc	0.7674	0.0708	0.0980	0.0288	0.0351
Li	0.8753	0.0202	0.0645	0.0060	0.0340
Dd	0.8678	0.0110	0.0452	0.0632	0.0128
Fp	0.6357	0.0003	0.0091	0.3508	0.0041
Mt	0.6805	0.0000	0.0065	0.0001	0.3129
Sp	0.9706	0.0068	0.0150	0.0034	0.0042
Tc	0.9905	0.0006	0.0085	0.0001	0.0003

Table 3 shows the percentage of fetches that cause the machine front-end to stall. The machine front-end stalls only due to instruction cache misses, mispredicted branches, and branch decode penalties.

No_Delay cause no stalls in instruction fetching. Bank_Conflicts to the same cache line do not stall instruction fetch, but conflicts to different cache lines within the same bank do stall instruction fetch. Therefore 84 to 90% of the fetches do not cause any instruction fetch stall. If a taken branch is not detected in a fetched instruction sequence (via a Branch Address Cache miss), a branch decode penalty is taken. Branch Decode penalties occur in approximately 2.4% and 0.4% of the fetch cycles for integer and floating point benchmarks, respectively. An incorrect branch path prediction requires a full branch penalty to be incurred. This happens about 7.2% and 1.7% of the time for integer and floating point.

2.4.5 Instruction Cache Configuration

Table 4. Instruction Cache Configurations

Configuration Number	Number of Interleaved Banks	Number of Read Ports	Set Associativity	Line Size	Fetch Size
0	8	1	2	16	2
1	8	1	1	16	2
2	4	1	2	16	2
3	8	1	4	16	2
4	8	1	2	32	1
5	8	2	2	16	2

We simulated six instruction cache configurations with various numbers of read ports, degrees of interleaving, set associativity, and line sizes. These configurations are listed in Table 4. Configuration 0 was used for most of our experiments. Fetch size refers to the number of cache lines each fetch address can access.

Figure 12 and Figure 13 show the performance with the various instruction cache configurations. The applications gcc and fpppp were chosen because they have more significant instruction cache miss rates. Each curve represents a different cache size. More read ports and more banks reduce bank conflicts but result in only a minimal performance increase. Higher set associativity significantly improves performance. However, fpppp actually has better performance with either direct-mapped or 4-way set associative caches due to the large sequential code segment. 32-byte line size degrades the performance a little because some bandwidth is wasted due to basic block alignment.

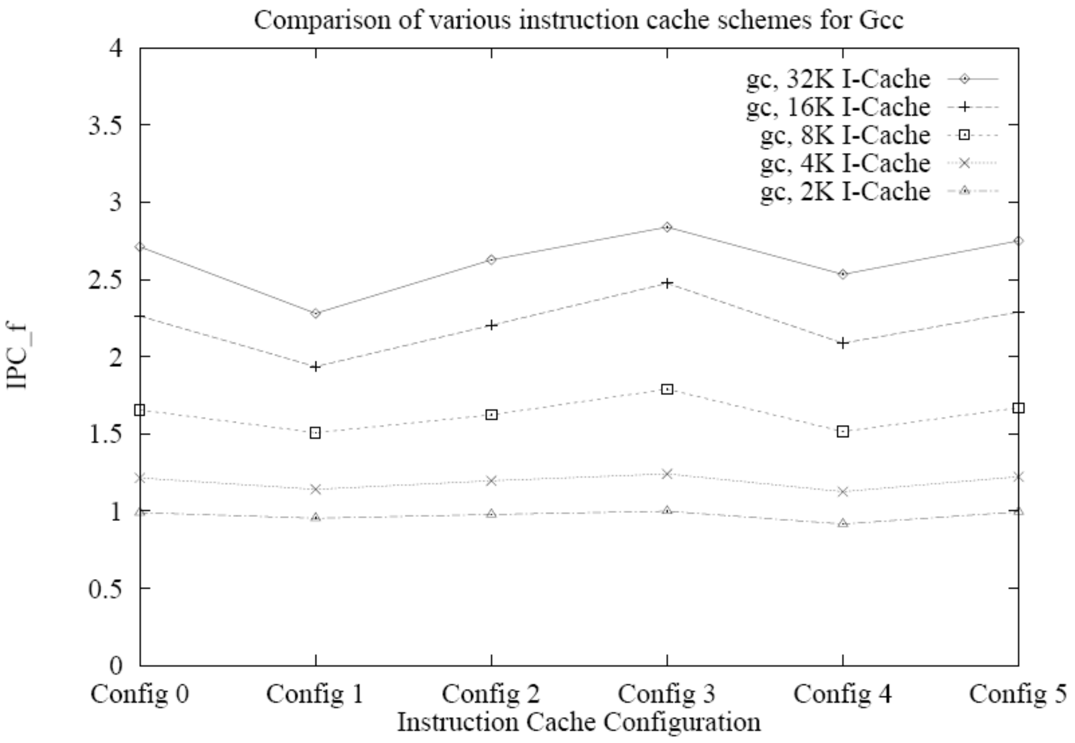

Figure 12. Machine performance of various instruction cache configurations on gcc.

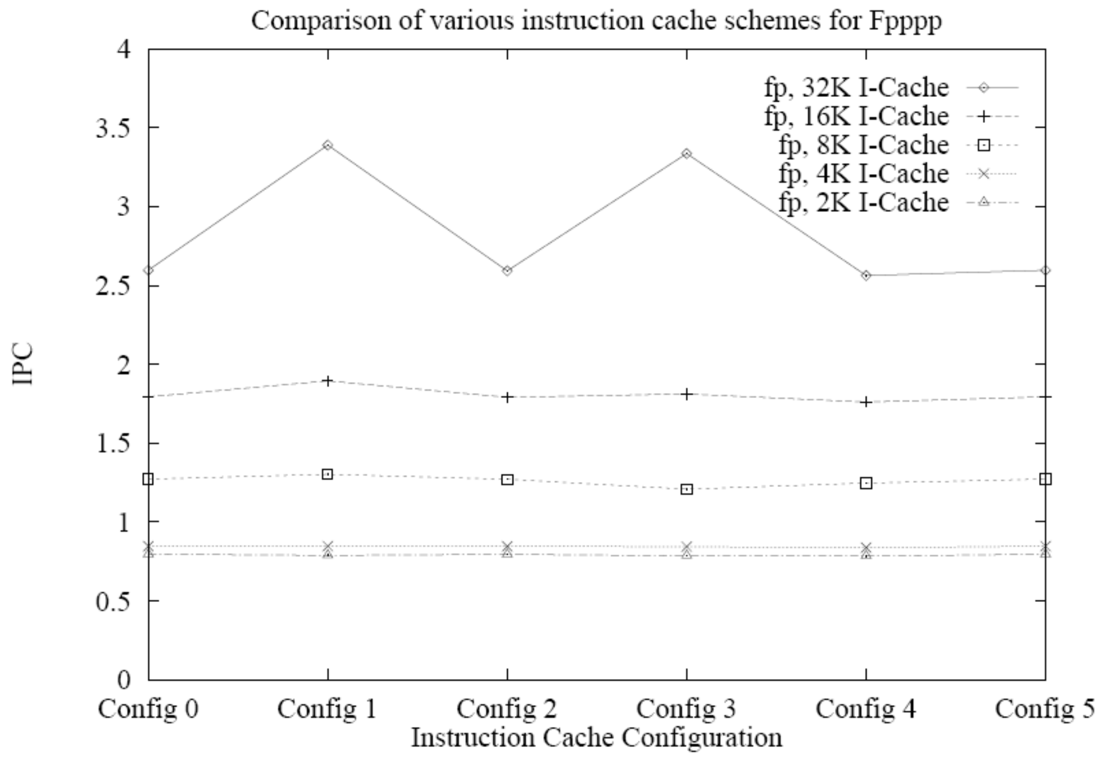

Figure 13. Machine performance of various instruction cache configurations on fpppp.

2.4.6 Effect of Branch Misprediction Penalty

To investigate the effect of branch misprediction penalty on machine performance, we varied the time to resolve a branch from 4 cycles to 12 cycles, as shown in Figure 14. Floating point programs have flatter curves because they contain fewer branches and the prediction accuracy of those branches is higher. The performance degradation when the branch resolution time is increased from 4 cycles to 12 cycles is less than 10%. Integer programs have about 20% to 30% performance degradation.

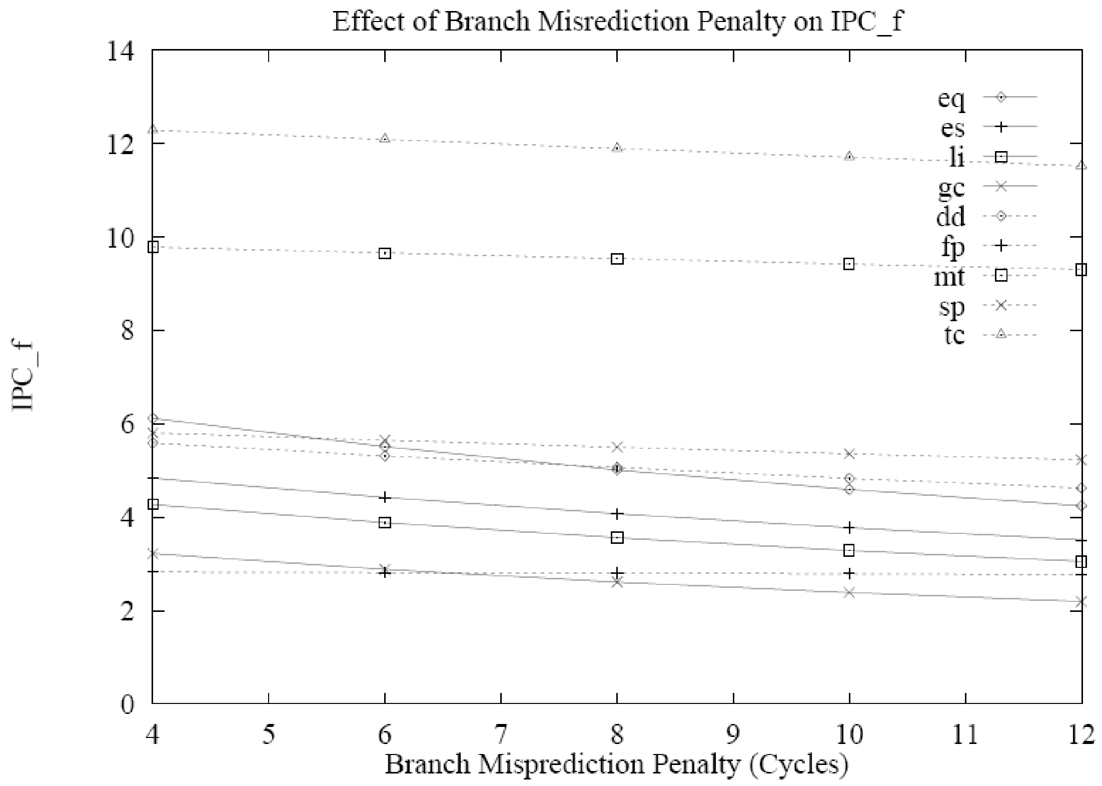

Figure 14. Effect of branch misprediction penalty on machine performance.

2.4.7 Effect of Instruction Cache Miss Penalty

We varied the instruction cache miss penalty from 4 cycles to 12 cycles. Configuration 0 of Table 4 is used. Among the nine benchmarks, fpppp, doduc, and gcc have lower cache hit rates, as listed in the legend of Figure 15. When the instruction cache miss penalty is increased from 4 cycles to 12 cycles, doduc's performance degrades by about 20%. The application fpppp's performance degrades by about 50%. The other benchmarks showed minimal performance degradation due to their low instruction cache miss rates.

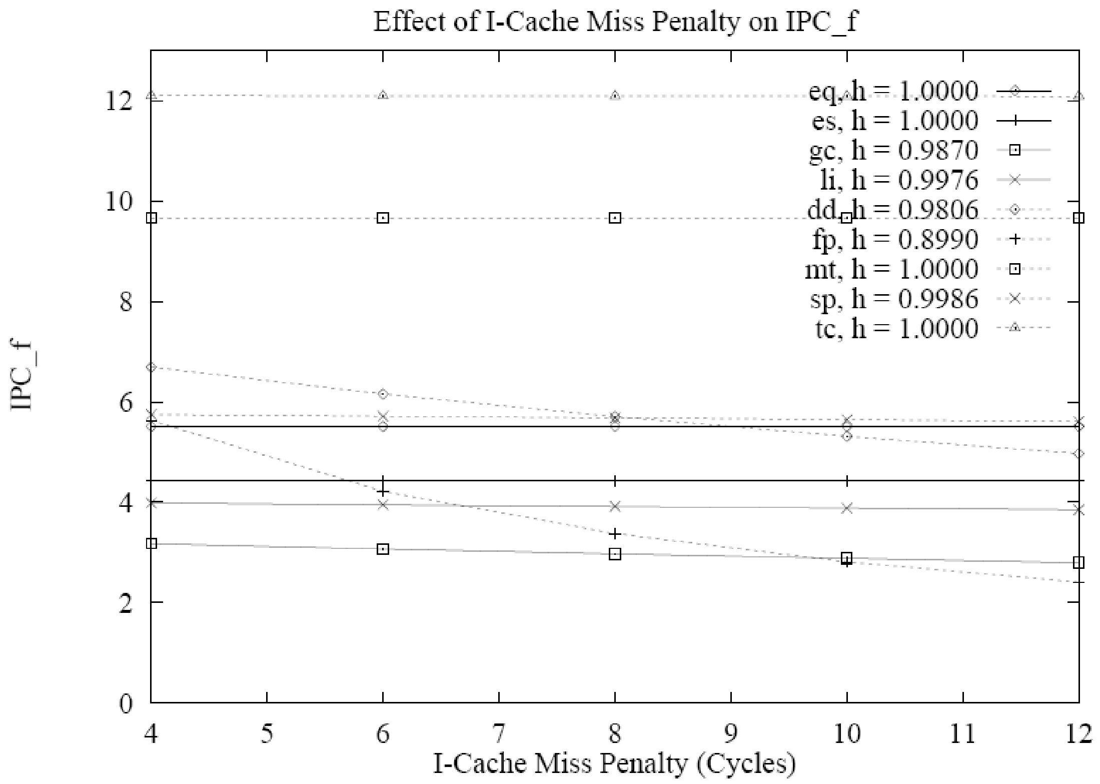

Figure 15. Effect of instruction cache miss penalty on machine performance.

2.5 Related Work

The research described in this chapter was the first in the area of multiple branch prediction, and is considered to be the seminal piece of work in this field and which kicked off a large body of other research which builds on this work. Therefore there is an extensive body of related work. We categorize the related work into three categories. The first is research that builds directly on the ideas and algorithms we presented in this chapter. The second category is other multiple branch prediction research which are not modifications of our algorithms. The third category is the area of trace caches, an alternative way of increasing fetch bandwidth by creating traces of execution paths and storing multiple, possibly non-contiguous, basic blocks into contiguous storage, or special caches, called trace caches.

While there are many papers which build on our algorithms, we will call out a few of the more notable improvements to our algorithms here. Calder et al. [10] proposed a modification of our multiple branch prediction algorithms to use cache indices instead of the full instruction fetch addresses. The advantage of this idea is lower storage area and costs, and the possible expense of accuracy. Conte et al. [11] improved on our multiple branch prediction algorithms by introducing the collapsing buffer for grouping non-contiguous basic blocks. Wallace et al. [12] improved on our multiple branch prediction algorithm by using an array of two-bit predictors for each instruction in the fetch block to predict multiple branches per fetch block. They also used a BTB with targets for each instruction in the fetch block which was used for near-block encoding to reduce the space needed to store target addresses. The limitation to their approach is that they can only handle one taken branch per cycle. In Lee et al. [13], instead of using the global history register they used the per-address history register to reduce interference. Both predictions come from the same history register of the primary address. Finally, Koppelman [14] published some interesting system simulations comparing our multiple branch prediction algorithms to a superblock predictor and showed that our multiple branch predictor gave superior performance of 10% over the superblock predictor which gave 8%.

The next category of related work is other multiple branch prediction algorithms, published after our work, inspired by our work, but not building directly on our algorithms. We list some of the more interesting research here. Seznec et al. [15] proposed an algorithm where the current instruction fetch address is not used for predicting the address of the next instruction block, but rather for predicting the block following the next instruction. This effectively pipelines the branch prediction loop to achieve higher clock rate, as opposed to wider instruction fetch (similar to the difference between superpipelining vs. superscalar). Rakvic et al. [16] proposed a tree-based multiple branch predictor which employs a three-level design with two levels of Pattern History Tables. It maintains a tree structure of individual single branch predictors, and based on their predictions, a path in the tree is identified as the candidate trace. Pnevmatikatos et al. [17] relied on compilers to partition the control flow graph (CFG) into tree-like subgraphs of depth 3. All parameters required to describe a subgraph are stored in a Subgraph History Table. Finally, in Reinman et al. [18] they propose decoupling the branch predictor and instruction fetch. The branch predictor is allowed to run far ahead of instruction fetch by storing many fetch addresses in a Fetch Target Queue. This allows optimizations such as a multi-level branch predictor design, and fetch-directed prefetching.

The last category of related work described here will be that of the Trace Cache. The instructions stored in the trace cache represent a dynamic trace of execution, and therefore takes multiple possibly non-contiguous blocks of instructions and stores them together to be easily fetched. The earliest publication was in the form of a U.S. Patent filed by Peleg and Weiser of Intel [19]. The earliest academic publication of the trace cache was by Rotenberg et al. [20], this publication gave the idea its current commonly known name, the Trace Cache. Patel et al. [21] published several more practical implementation options for the trace cache. And Intel's Pentium 4 Processor was the first commercial processor to implement a trace cache to support its aggressive out-of-order and simultaneous multithreading microarchitecture [22].

2.6 Conclusion

The trend towards increasingly complex and parallel hardware mechanisms to extract instruction level parallelism from sequential code is advancing at an accelerated rate. Much research has gone into compiler technology to increase basic block size in order to fetch more and more instructions at a time. Increasing basic block size is not enough, however. We propose in this chapter a hardware mechanism to fetch multiple basic blocks simultaneously.

We demonstrate in this chapter the viability of such schemes by identifying the three essential problems and presenting solutions to each of these. The Multiple Branch Two-level Adaptive Branch Predictor provides the capability of predicting multiple branches each cycle, the Branch Address Cache supplies the starting addresses of basic blocks following the multiple predicted branches, and an instruction cache with interleaved banks provides sufficient bandwidth for fetching multiple non-consecutive basic blocks without the hardware cost of multiple read ports.

In addition, we have presented simulation results indicating that significant performance improvements can be achieved even without specific compiler optimizations. When going from one to two to three branch predictions and basic block fetches per cycle, we saw the IPC_f (effective instructions fetched per cycle for a machine front-end) improve from 3.0 to 4.2, and 4.9, respectively for integer benchmarks. For floating point benchmarks, the IPC_f went from 5.6 to 7.1 and 8.9. These improvements were achieved by providing the hardware mechanisms to predict and fetch multiple basic blocks without specific compiler optimizations.

CHAPTER 3
SIMULTANEOUS MULTITHREADING

3.1 Motivation

Processor speeds have increased dramatically over the past decades, much faster than the speed of memory, so memory latency tolerance has become a major focus of attention. Even more importantly, power consumption has been increasing at a greater rate than processing speeds [23-25]. In the past, shrinking process technology producing smaller and smaller devices has alleviated the power requirements to some extent. However, process devices are now so small that current leakage is becoming a major concern.

Power-efficient computing for memory latency tolerance has led Intel, IBM, and others to embrace simultaneous multithreading (SMT) as a solution. Intel and IBM have introduced the first commercially available processors with SMT in recent years.

SMT builds on the dynamic-issue superscalar processing technology of modern high-performance processors to increase the pool of instructions available to the processing units. This helps keep the processor busy in the face of long memory latencies due to cache misses, as well as for code with low to medium instruction-level parallelism. Since leakage power is typically about 40% of total power for today's processors [23], and is expected to grow as a percentage of total power [24, 25], reusing the same processor's resources more efficiently can result in much more power-efficient computing. It is better to use existing physical processing resources to their full extent in order to amortize the leakage current.

An SMT processor from the hardware perspective looks like one physical processor. However from a software (operating system or application) perspective it looks like multiple processors. The operating system can schedule a runable software thread for every "logical processor", and there may be many logical processors for each physical processor. The SMT processor will fetch the instructions for all the software threads scheduled on it, to increase the pool of independent instructions available to the execution units. Since there are no data dependencies between instructions from different software threads, the number of independent instructions can be dramatically greater than in a non-SMT processor. In addition, the variety of instructions could potentially be greater and thus make more efficient use of hardware resources that otherwise might remain idle.

Each logical processor has its own set of registers, instruction pointer, and a few other resources, such as interrupt handling mechanisms. In general, SMT processor execution units are unaware of what instructions belong to which software thread. Completed instructions are sorted out to the independent software threads to which they belong so that each architectural state can be updated correctly.

SMT provides higher overall system throughput and therefore performance. However, there are important considerations that must be carefully addressed when designing an SMT processor. Since SMT logical processors share the physical processor's resources, the overall throughput depends on the mix of software threads and their characteristics that happened to be scheduled simultaneously. Predicting performance and throughput can be challenging. Ensuring fairness can be even more difficult. Some threads may thrash the cache, some threads may have aliased addresses with other threads, some threads may compete for the same resources, and some threads may be so highly optimized to run on the implementation alone, that any interference can result in poor performance. Some real-time applications may require a minimum bandwidth from physical resources. These and other considerations are only beginning to be understood.

If the SMT processor were targeted at a specific market, such as the network processor market where every threads is handling packet processing, the choices and tradeoffs for microarchitecting the processor may be easier. Or if the processor is targeted at a specific database processing market where the application and operating system are aware and optimized for the SMT implementation, the design choices and tradeoffs could also be made in a relatively straight-forward manner.

It can be an entirely different matter for a processor that must contend with any operating system, and a wide variety of applications. This is the environment for which the Pentium 4 processor was targeted, and the focus of this work.

3.2 Terms

For clarity, we will avoid using the generic terms “thread” or “threads” due to confusion between hardware contexts and software threads. Accordingly, we define the following terms for use in the rest of this thesis:

- **Software threads**: Software may be written with one or more threads that can be schedulable separately by the operating system onto logical processors for execution.
- **Logical processor**: What the software/operating system views as a schedulable processor entity. Distinct logical processors may or may not share a physical processor’s resources.
- **Physical processor** or **Package**: The physical implementation of a processor entity, typically consisting of a full pipeline from instruction fetch to execution and then to retirement.
- **IA-32**: The Intel 32-bit instruction set architecture.

3.3 Background

In this section I will describe some of the history of SMT research as well as other hardware multithreading approaches. A concise survey of a variety of other hardware multithreaded processors, including their general characteristics, is presented in a paper by Ungerer et al. [26]. In general, there are three categories of hardware multithreading techniques: Simultaneous multithreading (SMT), interleaved multithreading (IMT), and switch-on-event multithreading (SOEMT).

3.3.1 Simultaneous Multithreading

Early work on SMT was done at both Intel, and at universities. At Intel, early research resulted in a proposal to add SMT (internally it was called shared resource multiprocessing) to the P6 microarchitecture was done in 1991. At the time, it was decided not to add this to the P6 for market reasons. After adding SMT to their Pentium 4 and Xeon products, Intel announced their SMT work in the Fall of 2001 at the Intel Developers' Forum where an entire track was devoted to introducing and optimizing for their SMT design [27].

Mario Nemirovsky did early research work published in his Ph.D. dissertation [28] on SMT in the late 1980s, which he called multistreaming. He later proposed to implement SMT in the Clearwater network processor [29, 30], but this project was never completed.

The most commonly cited research work was done at the University of Washington by Dean Tullsen [31], where he evaluated an SMT design and compared it with a superscalar non-SMT design and an interleaved multithreaded design (IMT). He found that SMT was a clear performance win. His later work showed that adding SMT to an out-of-order superscalar design would involve only a relatively small cost [32-34].

Another team, Loikkanen and Bagherzadeh, was also working on a fine-grain multithreading processor which had many similar characteristics to SMT, including a shared but partitioned register file and dynamically shared execution units [35].

3.3.2 Interleaved Multithreading

Interleaved multithreading refers to processors that hold the context of a number of software threads while alternating execution of the instructions in the threads on a cycle-by-cycle basis. Examples of supercomputing products which use interleaved hardware multithreaded architectures include the Heterogeneous Element Processor (HEP)[36], the Horizon[37, 38], the Tera (based on the Horizon)[39], and the Cray Multi-Threaded Architecture (MTA-2)[40]. Other proposals included the Multilisp Architecture for Symbolic Applications (MASA)[41], MIT's M-Machine[42], MicroUnity's Media Processor[43], and the SB-PRAM/HPP[44, 45]. An example of a network processor is the Lextra LX4580, which could have up to five threads. In principle, the interleaved multithreading technique can be combined with superscalar out-of-order execution techniques, but Eggers et al., show that SMT is more efficient [33].

3.3.3 Switch-on-Event Multithreading

SOEMT refers to processors that context switch to a different software thread on certain long-latency events, such as a cache miss, to a different software thread. This type of multithreading is coarser grained than SMT or interleaved multithreading. Examples of SOEMT include the MIT Sparcle [46] and the Msparc [47, 48] processors which switch on cache miss. The Columbia Homogeneous Parallel Processor (ChoPP) 1 [49] uses switch-on-cache-miss and switch-on-use. The Decoupled Multithreaded Processor Rhama [50, 51] uses several static and dynamic events. The EVENTS scheduler [48, 52] uses an external scheduler to trigger context switches. The Komodo microcontroller [53-55] detects real-time events requiring fast response and uses those events to trigger context switches. Commercial processors include the IBM RS64 IV [56] and the Sun MAJC [57]. Network processors include the Intel IXP [58], IBM Power NP [59], Vitesse IQ2x00 [60], and AMCC nP [60]. Finally, the MIT Jellybean Machine (J-Machine) [61] falls into this category.

3.4 Performance of SMT vs. SOEMT

Much has been said about the benefits of SOEMT compared to SMT. In the following sections, we will quantify the performance benefits.

Since we do not have any real systems that can be used to readily compare switch-on-multi-threading vs. simultaneous multi-threading, we use our Pentium 4 simulator. The simulations were done for the Pentium 4 "Prescott" configuration (31-stage pipeline, 16KB L1 cache, 1MB L2 cache, SSE3).

The switch-on-event configuration was as follows:
1. Only one software thread's uops are active in the pipeline at a time.
2. There are two instruction pointers in the trace cache, microcode ROM, and pre-decoder. There are instruction prefetch buffers for both threads so that both logical processors will have prefetched instruction bytes ready-to-decode.
3. The switching to executing uops from the other thread happens in a single cycle. We clear the pipeline and start reading bytes from the prefetch buffers for the other thread simultaneously.
4. The "events" that we switch on are cache and TLB misses that must go to DRAM (i.e., misses the last-level on-chip cache), and IN and OUT instructions that stall the pipeline. An out-of-order pipeline can often mitigate short-latency delays, but has trouble with long delays, such as those required for DRAM accesses and IN/OUT instructions.

5. Data loads that miss the cache are allowed to become the oldest uop in the pipeline before we clear the pipeline and switch to the other thread. This allows as many as possible simultaneous load misses to get started to DRAM before we switch to the other thread.

For workloads, we used a wide variety of traces grouped into "classes" to compare the performance. The classes of traces are: SPEC2000, Internet, Multimedia, Productivity, Workstation, and Server. Table 5 lists the applications in each trace class.

Table 5. Table of trace categories, the applications in those trace categories, and the number of traces of each application that were used for simulations.

Trace Category	Applications	# of Traces
SPEC2000:	Gzip	3
	Wupwise	3
	Swim	3
	Mgrid	3
	applu	3
	Vpr	3
	Gcc	3
	Mesa	3
	Mcf	3
	Equake	3
	Crafty	3
	Facerec	3
	Ammp	3
	Lucas	3
	Fma3d	3
	Parser	3
	Sixtrack	3
	Perlbmk	3
	Gap	3
	Vortex	3
	Bzip2	3
	Twolf	3
	Apsi	3
Internet	Webmark2001 b-autoconcepts	3
	Webmark2001 b2b-ecommodity	3

	Webmark2001 b2b-ehousebuilder	3
	Webmark2001 b2b-electronics	3
	Webmark2001 b2b-emedinsure	3
	Webmark2001 b2b-myfoyer	3
	Webmark2001 b2b-superetailer	3
	I-bench Quicktime	1
	I-bench Shockwave	1
	I-bench VRML	1
Multimedia	3dwb2k	1
	Dragon-Naturally-Speaking	3
	EjayMP3Encoder	2
	Flask-MPEG4-Encoder	4
	Photoshop	1
	QuakeIII Arena	2
	Virtual-ray Scene5	1
Productivity	Winstone2001_Business	10
	Winstone2001_ContentCreation	10
	Wintune_Test2	1
	CPUmark99	1
	Hammerhead	2
	Officebench11	2
	Sysmark2K	23
Workstation	Ansys55	5
	Nastran	4
	Oasis	3
	FPUmark99	1
	Verilog	3
	Catlym	1
	Viewperf	1
Server	SQLServer	15

We ran the simulations in two ways. The first we referred to as “domain decomposition”, where we follow the software threading practice of taking a single computation loop and multi-threading each iteration of the loop. Since we don’t have any traces of true multi-threaded applications, we approximated the behavior by running the same trace on both threads, but offsetting the physical addresses of the memory accesses. The second we referred to as “functional decomposition”, where we follow the software threading practice of using different threads to implement different functions (e.g. grammar checking, printing, rendering different objects) or multi-tasking. Again, we don’t have any traces of a true multi-threaded applications or multi-tasking workloads, so we approximated the behavior by running different combinations of traces. The trace combinations were established randomly.

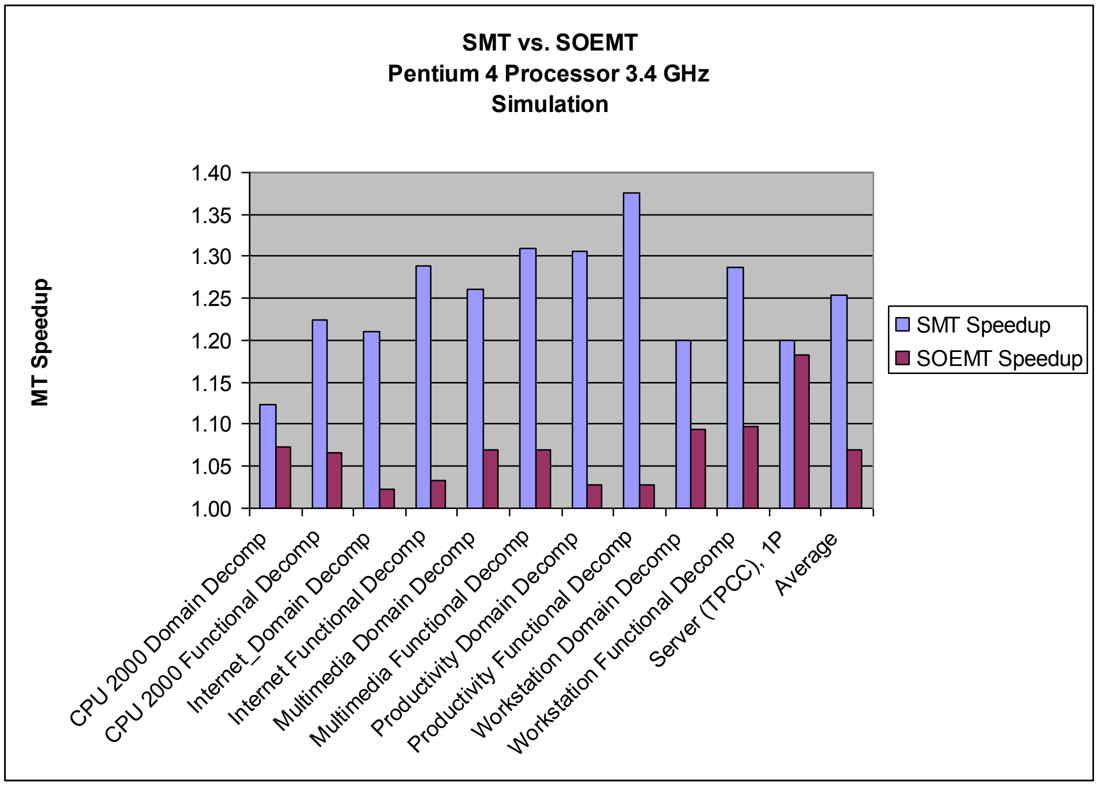

Figure 16. Simultaneous Multithreading vs. Switch-on-Event Multithreading performance comparison on different application classes.

Figure 16 shows the results of the simulation. With the exception of the database class of workloads, much of the performance benefit of SMT comes from taking advantage of both inefficient pipeline use and cache latency tolerance, while SOEMT only takes advantage of cache latency tolerance . The database class of workload is dominated by cache misses and I/O, so it is only in that area that SOEMT benefits are close to that of SMT.

Specifically, applications where SOEMT exceeded 20% included: database server application running a warehouse transaction application, SPECjbb, SPECweb, Verilog, trade2, facerec, mcf, msvc, equake, art, and vpr. Applications where

SOEMT gave a 10-20% benefit included Media Encoder, Ansys, Nastran, Dragon Naturally Speaking, and an MPEG1 encoder.

In conclusion, SOEMT might provide significant benefit in some application spaces, especially server application spaces which run a lot of applications similar to transaction processing, SPECweb, and SPECjbb.   It can also significantly boost a few other applications. However, SMT benefits a much wider classes of applications.

CHAPTER 4
SIMULTANEOUS MULTITHREADING IMPLEMENTATION

This chapter presents our work in designing the Intel Pentium 4 Processor's SMT implementation, and compares our implementation with those of other commercial implementations. Section 4.1 describes the Pentium 4 microarchitecture. Sections 4.2 through 4.4 describe the implementations of commercial processors such as the IBM Power5, Alpha EV8, and the Clearwater CNP810SP, as well as comparing their features with the Pentium 4 processor SMT implementation. Only the Intel and IBM Power5 implementations were completed; the Alpha and Clearwater processor designs were cancelled before they could be completed.

Several goals were at the heart of the microarchitecture choices and tradeoffs of the Pentium 4 implementation of simultaneous multithreading (SMT). One goal was to minimize the implementation cost in terms of die area. Since the logical processors share the vast majority of microarchitecture resources, only a few small structures were replicated. The cost was less than 5% of the total die area. Figure 17 shows some of the larger structures contributing to this die area increase.

Since the die area allotted to the Pentium 4 processor's SMT implementation was limited, there were some interesting trade-offs that were made to maximize the usefulness of the limited resources for best performance. There were also interesting opportunities that were left on the table since they were either too expensive or too complex for a first implementation of a risky technology.

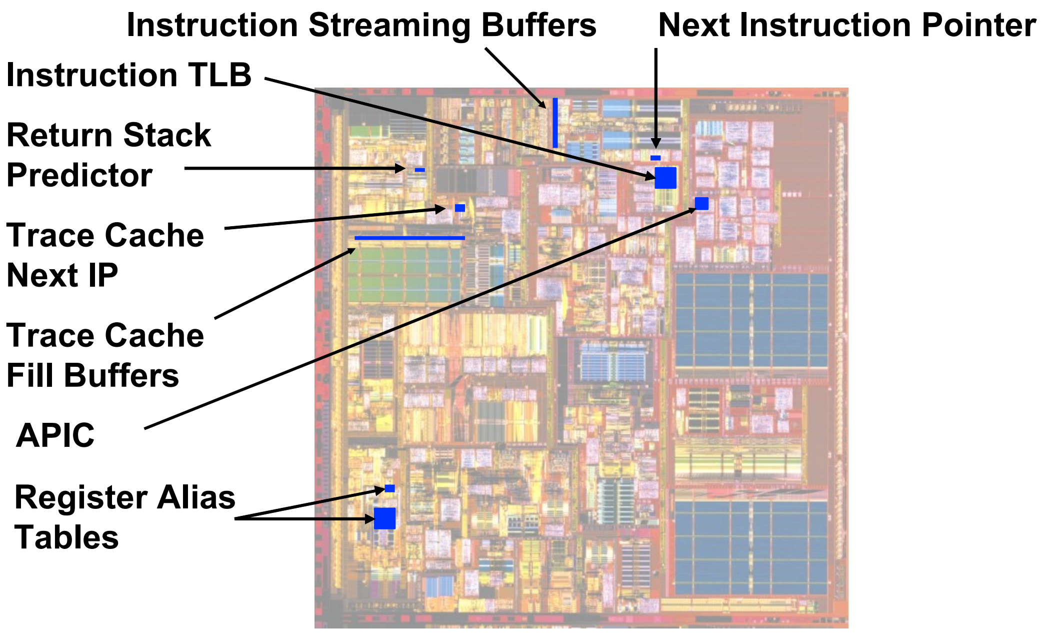

Figure 17. The Intel Pentium 4 processor and the visible processor resources duplicated or added to support SMT. There are also scattered miscellaneous pointers and control logic that were too small to highlight in this figure. The APIC is the advanced programmable interrupt controller.

A second goal was to ensure that when one logical processor was stalled the other logical processor could continue to make progress. A logical processor can be stalled for a variety of reasons, including servicing cache misses, handling branch mispredictions, or waiting for the results of previous instructions. Independent progress was ensured by managing buffering queues such that no logical processor could use all the entries when two active software threads were executing. Raasch and Reinhardt reinforced the Pentium resource partitioning decisions by showing that simple partitioning schemes were the fairest, while more complex dynamic partitioning schemes did not significantly improve overall throughput [62].

A third goal was to ensure that an SMT processor running a single software thread could run at the same speed as a processor without SMT capability. This implies that partitioned resources should be able to be recombined when only one software thread is active to give the single software thread the full resources of the physical processor.

4.1 Intel Pentium 4 and Xeon Processor Family SMT Microarchitecture

The Pentium 4 processor [22] and Xeon processor family share a common microarchitecture. The microarchitecture was designed as a general-purpose microprocessor aimed at running a wide variety of computing applications well. The Pentium 4 and Xeon family can vary in the maximum number of processors supported and in cache size and hierarchy. The SMT features were implemented at minimum cost to allow two logical processors [63] to share the resources of a single physical processor. The general flow of the pipeline is shown in Figure 18. Buffering queues separate major pipeline logic blocks. These buffering queues are either partitioned or duplicated to ensure independent forward progress through each logic block.

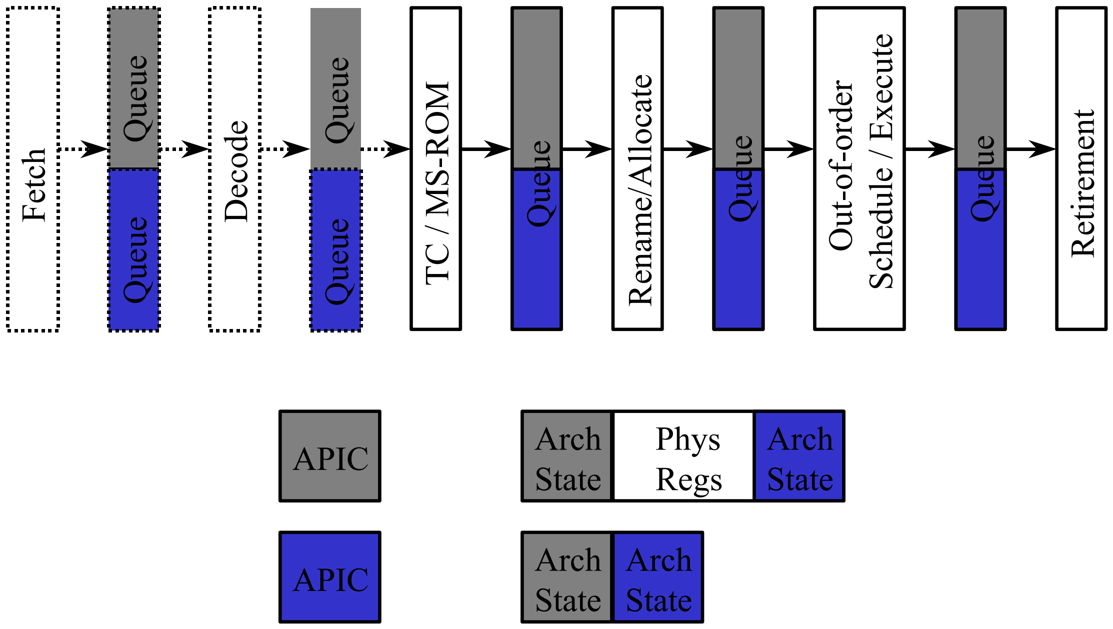

Figure 18. Intel Pentium 4 Processor Pipeline. APIC is the advanced programmable interrupt controller. TC is the trace cache. MS-ROM is the micro-sequencer read-only-memory which stores and sequences microcode.

In the following sections we will walk through the pipeline and discuss the implementation of the major function blocks.

4.1.1 Front End

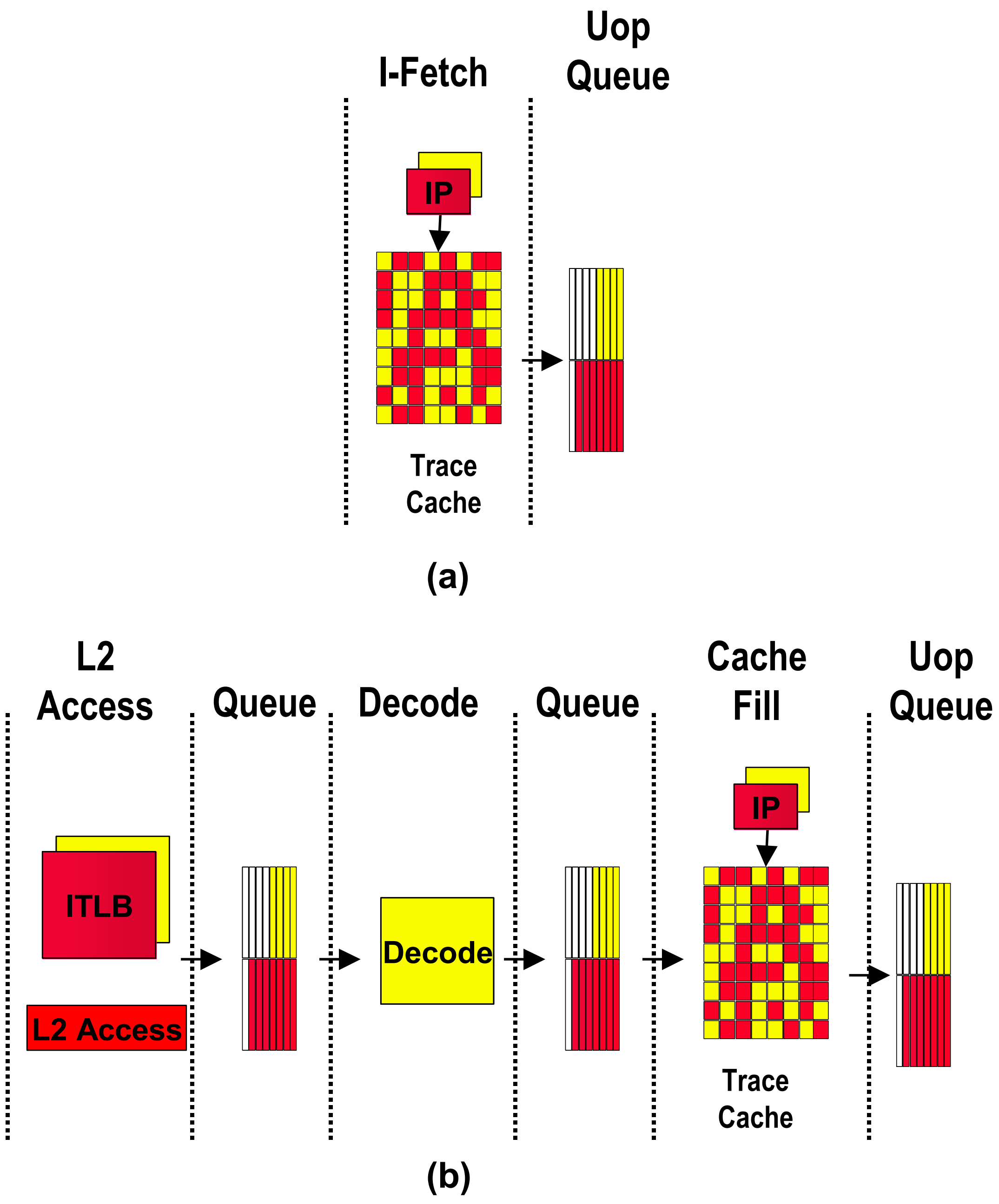

Figure 19. Details of the Front-end Pipeline for (a) Trace Cache Hit and (b) Trace Cache Miss.

The front-end of the pipeline is responsible for fetching, decoding, and delivering micro-operations ("uops") to the later stages of the pipeline.

Instructions generally come from the Execution Trace Cache (TC), which is the primary instruction cache, as shown in Figure 19(a). Figure 19(b) shows that only when there is a TC miss does the machine fetch and decode instructions from the unified Level 2 (L2) cache (unified because it is a shared cache for both instructions and data). Not shown, but in the same functional block as the TC is the Microcode ROM (MS-ROM), which stores decoded instructions for the longer and more complex IA-32 instructions.

Execution Trace Cache (TC). The TC stores decoded instructions, or uops as described above. Most instructions are fetched and executed from the TC. Two sets of next-instruction-pointers (one for each logical processor) track the progress of the two software threads running on two logical processors. The two logical processors arbitrate access to the TC every clock cycle. All TC entries are tagged with logical processor ID such that a single entry cannot be used by both logical processors. The TC is 8-way set-associative; entries are replaced based on a least-recently-used (LRU) algorithm.

Microcode ROM (uROM). The uROM stores uops for either less-commonly used or more complex Intel Architecture instructions. When one of these instructions are encountered, the TC sends a pointer to the uROM which then fetches the sequences of uops needed and returns control to the TC. Two microcode instruction pointers are used to control the flows independently if both logical processors are executing uops from the uROM; however, both logical processors share the uROM entries.

ITLB. When there is a TC miss, the ITLB receives the request from the TC to deliver new instructions, and translates the next-instruction-pointer address to a physical address. The request is sent to the L2 unified cache, and instruction bytes are returned. These bytes are placed into streaming buffers which hold the bytes until they can be decoded. Each logical processor has its own ITLB and its own set of instruction pointers to track progress of instruction fetch and its own set of streaming buffers to enable independent progress for instruction fetch.

Branch prediction. The return stack buffer is duplicated for better call/return prediction. The large global branch history array is a shared structure with entries tagged with a logical processor ID. The branch history register used to look up the global history array is duplicated to track the branch history of the two logical processors independently.

Instruction Decode. The decode logic takes instruction bytes from the streaming buffers and decodes them into uops. In general, if both logical processors need access to the decoder, the decode logic will decode several instructions for one before switching to decode several for the other logical processor. This allows the implementation to share all of the complex logic and buffering required to decode IA-32 instructions. These decode uops are then placed into the TC.

Uop Queue. After uops are fetched from the TC or the uROM, or forwarded from the decode logic, they are placed in the uop queue. This queue decouples the Front End from the Out-of-order Execution Engine in the pipeline flow. The uop queue is partitioned such that each logical processor has half the entries.

4.1.2 Out-of-order Execution Engine

The out-of-order execution engine consists of the allocation, register renaming, scheduling, and execution functions, as shown in Figure 20. This part of the machine re-orders instructions and executes them as soon as their inputs are ready, without regard to the original program order.

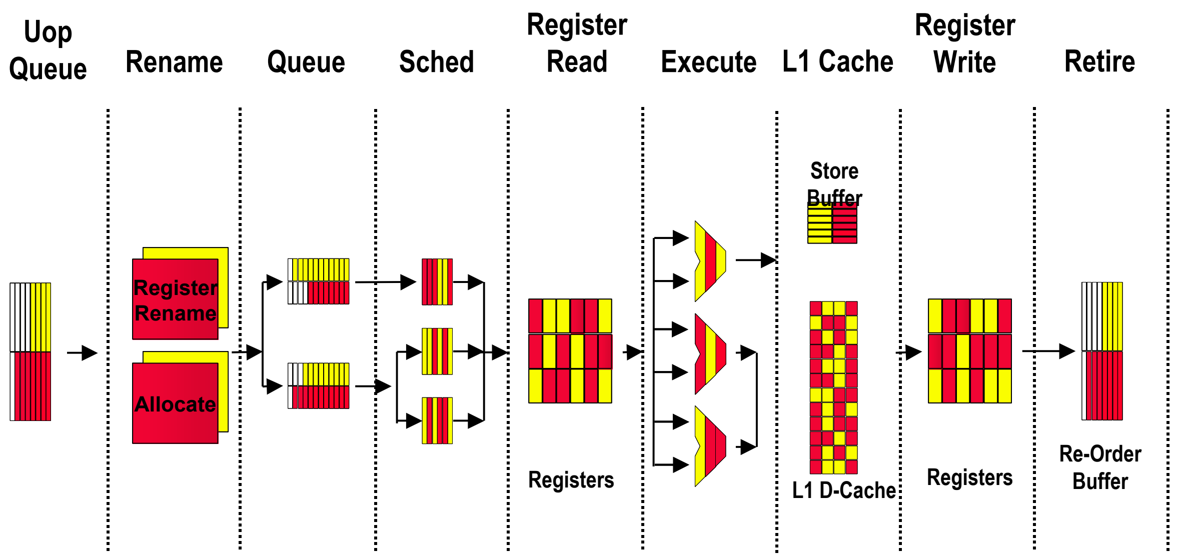

Figure 20. Details of the Out of order Execution Engine Pipeline.

Allocator. The allocator allocates many of the key machine buffers for each uop, including the 126 re-order buffer entries, 128 integer and 128 floating point physical registers, 48 load and 24 store buffer entries. If there are uops for both logical processors in the uop queue, the allocator will alternate selecting uops from the logical processors every other cycle to assign resources. Each logical processor can use at most half of the resources allocated at this stage.

Register rename. The register rename logic renames the architectural IA-32 registers onto the machine's physical registers. The renaming logic uses a Register Alias Table (RAT) to track the latest version of each architectural register to tell the next instruction(s) where to get its operands. There are two RATs, one for each logical processor. Register renaming is done in parallel with the allocator logic described above. Once uops have completed the allocate and register rename processes, they are placed into two sets of queues, one for memory operations and another for all other operations, called the memory instruction queue and the general instruction queue, respectively.

Instruction scheduling. Five uop schedulers are used to schedule different types of uops for the various execution units. Collectively, they can dispatch up to six uops each clock cycle. The memory instruction queue and the general instruction queue send uops to the five scheduler queues as fast as they can, alternating between uops for the two logical processors every clock cycle, as needed. The schedulers are oblivious to logical processor distinctions; they simply evaluate whether to dispatch uops based on dependent inputs and availability of execution resources. To avoid deadlock and ensure fairness, there is a limit on the number of active entries that a logical processor can have in each scheduler's queue.

Execution units. The execution core and memory hierarchy are largely oblivious to the logical processors. Uops merely access the physical register file to get their destinations and then write results back to the physical register file. By simply comparing physical register numbers, the forwarding logic sends results to other executing uops without having to understand logical processors. After execution, uops are placed in the reorder buffer.

Retirement. The retirement logic tracks when uops from the two logical processors are ready to be retired, then retires the uops in program order. Retirement will retire uops for one logical processor, then the other, alternating back and forth. For stores, once retired, the store uop needs to write its data into the data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

4.1.3 Memory Subsystem

The memory subsystem includes the DTLB, the level 1 data cache, the level 2 unified cache, and an optional level 3 unified cache. The memory subsystem is largely oblivious to logical processors. The schedulers send load or store uops to the memory subsystem without regard to logical processors, and the memory subsystem handles them as they come.

DTLB. Although the DTLB is a shared structure, each entry includes a logical processor ID tag. Each logical processor also has a reservation register to ensure fairness and forward progress in processing DTLB misses.

L1 data cache, L2 unified cache, (and optional L3 unified cache). The L1 data cache is virtually addressed and physically tagged. Each entry includes a context identifier [64] which is dynamically set or reset based on whether the page-directory base addresses (stored in a control register) are the same or different for the two logical processors. If the page-directory base addresses are the same, then the two logical processors are likely to be sharing the same data and therefore can read/write each others' cache entries in an optimal way. If different then the two logical processors are unlikely to be sharing the same data and we can prevent partial-address aliasing conflicts by giving them two different context identifiers.

Bus. From a service perspective, cache miss requests and other bus requests from the logical processors are processed on a first-come-first-served basis, with queue and buffering space shared. Priority is not given to one logical processor over another at any time. For debug purposes, the logical processor ID of the request that generated the transaction is visibly sent onto the bus in the request phase.

Interrupts. Requests to the local APIC (advanced programmable interrupt controller) and interrupt delivery resources are unique and separate per logical processors.

4.1.4 Single-task and Multi-task Modes

To optimize performance when there is one software thread to execute, there are two modes of operation, called single-task (ST) and multi-task (MT). In MT-mode, there are two active logical processors and some of the resources are partitioned as described earlier. In ST-mode, only one logical processor is active and resources that were partitioned in MT-mode are re-combined to give the single active logical processor full use of all the resources. The two flavors of ST-mode are ST0-mode and ST1-mode, depending on which logical processor is active.

4.2 IBM Power5

The IBM Power5 is a dual-core microarchitecture. It has two identical processor cores, each supporting two logical processors (or threads in IBM terminology). The two cores share a 1.875-Mbyte L2 cache. There is also an integrated directory for an off-chip 36MB L3 cache, and an integrated memory controller.

The key differences between the IBM Power5 [65] and the Intel Pentium 4 processor are due to different goals for the SMT implementations. Intel intended for their processors to run with shrink-wrapped off-the-shelf operating systems and any variety and combination of standard off-the-shelf software applications. Intel therefore went out of the way to balance fairness and throughput aspects throughout the microarchitecture.

The IBM Power5 assumes a special operating system to dynamically detect whether software threads run well together and only schedule threads if they run well. The applications themselves also must be aware of the SMT implementation in order to run. Therefore, while IBM includes deadlock detection and resolution mechanisms, they did not go to the same effort to balance throughput and fairness in the microarchitecture, since only applications that are known to benefit and to run "fairly" on the SMT-enabled system would share the SMT feature. In fact, the major drawback of the IBM approach is lack of backward compatibility and the requirement that applications and operating systems be aware of the SMT feature. This is an onerous restriction, and puts huge responsibility on application writers who may not want to worry about coding to specific hardware resource sharing on specific implementations. This is likely to limit the number of applications that can take advantage of IBM's SMT implementation, and those applications that do take advantage of it may be limited in performance on future implementations due to different tradeoffs in terms of resource utilization and balance.

4.2.1 Front-end

The Power5 front-end operation is similar to the Pentium 4 processor. Instruction fetches alternate between the two logical processors, they share the instruction translation facilities, and they share the instruction cache.

For branch prediction, the IBM Power5 uses separate return stack buffers, but entirely shares the branch prediction state. The Pentium 4 processor shared most of the large structures, but had separate branch history buffers and return stack buffers. A drawback of the IBM implementation, where even the branch history buffers are shared, is increased likelihood of incompatible branch histories resulting in poorer branch performance.

The logical processors have separate instruction fetch queues to place instruction bytes after fetch, similar to the Pentium 4 processor.

Decode is done for one logical processor at a time, the logical processor selection is based on logical processor priority, which can be set by software or by hardware if unfairness is detected. Instructions are decoded in groups of up to 5 instructions per cycle, and each group is allocated an entry in the global completion table (GCT). Unlike the Pentium 4 Processor's reorder buffer which allows each logical processor to occupy only half the entries, the Power5's GCT entries can be entirely occupied by a single thread. This can be a potential source of deadlock or fairness issue.

4.2.2 Out-of-order Execution Engine

Like the Pentium 4 processor, the Power5's logical processors dynamically share the physical register files. After renaming, the instructions are placed into shared issue queues. Instructions are scheduled and issued to execution units with no regard to logical processors.

The GCT, which tracks instructions through the pipeline, groups instructions for tracking. Each group takes an entry in the global completion table. There are 20 entries in the GCT, which can each hold up to 5 instructions in a group. While each entry can only contain instructions from one logical processor and are allocated in program order, successive entries may belong to different logical processors. When all instructions in a group have executed and the group is the oldest for the given logical processor, it can commit (or retire in Intel terminology). Up to two groups can commit per cycle, one group from each logical processor.

4.2.3 Memory Subsystem

The L1 instruction and data caches are 64KB 2-way set-associative and 32KB 4-way set-associative, respectively. The first-level data translation table is 128 entries, which is the same as the Power 4, but it was made fully associative for better SMT performance.

The non-core levels of cache hierarchy (1.875 MB L2 cache, the optional L3 cache, and bus/memory access) is assumed to be competitively shared.

Overall, since the L1 cache sizes are larger on the Power5 than the Pentium 4 processor, the Power5 may have better average cache hit rates on many applications. However, the lower set associativity increases the likelihood of cache interference (thrashing) resulting in less predictable performance benefits from the SMT capability.

4.2.4 Single-task and Multi-task Modes

The IBM Power5 supports the same set of single-task and multi-task modes as Intel's Pentium 4 processor: two flavors of the ST-mode, and the MT-mode.

4.2.5 SMT Performance Enhancing Features

The Power5 depends on software, including operating system, middleware, and applications, to appropriately set the priority levels in order to run optimally. They call this feature "adjustable thread priority". Software is responsible for choosing the correct balance of priority. Some of the reasons listed by IBM [65] for changing thread priority include:

- Spin loops. Software would give the software thread lower priority because it's not doing useful work.
- Idle loops. If there is no immediate work for the OS to schedule to a logical processor, the OS would run an idle loop which is similar to a spin loop. The idle loop is not doing useful work and the OS should move the idle thread to a lower priority.
- One application is more important than another. For example, real-time tasks may be given higher priority. Or foreground tasks may be given higher priority than background tasks.

Ensuring that all software would use thread priority appropriately without abusing it is a challenge. IBM feels that this is reasonable since they own the entire software stack for their server systems.

In addition to software-controlled priority levels, Power5 also has a feature they call "dynamic resource balancing" which is needed to ensure that instructions from two logical processors flow smoothly through the processor. This is needed because a single L2 cache miss can cause dependent instructions to quickly backup the issue queues, slowing down groups of instructions from the other logical processor. Or one logical processor may be running a software thread that has higher CPI (and is therefore slower) than the other due to the mix of dependencies and instruction types. The slower thread would eventually use more and more of the GCT entries and slow down the faster thread. The Power5 microarchitecture monitors the number of L2 cache misses and the number of GCT entries that each thread is using and then takes one or more of the following actions:

- Reduce the logical processor priority. This is the primary mechanism for cases where a thread uses more than a predetermined number of GCT entries.
- Inhibit the logical processor's instruction decoding until the congestion clears. This is the primary mechanism for cases where a logical processor has greater than a prescribed number of L2 cache misses.
- Flush all instructions waiting for dispatch and halt decoding instructions for one logical processor. This is the primary mechanism for throttling in the case of a long-executing instruction such as a synch instruction.

The problem with the first two mechanisms is that application behavior tends to be very bursty. Some instruction segments will be slower for a while, due to dependencies and cache misses, after which the instructions may execute quickly for a while. Putting throttling mechanisms at the front-end of the machine is generally too late to react to most conditions, such as L2 cache misses and slow segments of code. As for the third mechanism, flushing instructions is expensive and wastes power, and therefore is not suitable for frequently encountered conditions.

4.3 Alpha EV8

The Alpha EV8 processor was cancelled before it was completed. However, its microarchitecture included a proposed 4-way SMT implementation [66] [67] [68]. The EV8 was intended to be an 8-wide out-of-order superscalar microarchitecture. The additional silicon area to implement 4-way SMT was estimated to be less than

10%. Few details are publicly available describing the SMT implementation in the microarchitecture.

The EV8 was much more aggressive in trying to achieve 4-way SMT performance than the Pentium 4's modest 2-way SMT. However, going to 4-way SMT adds a disproportionate amount of complexity to the microarchitecture. With 4 logical processors, and many structure sizes are limited by access latency, which means that complex sharing algorithms and even more complex fairness algorithms are required. This is in contrast to the simple approach of dividing a few key resources in half as done in the Pentium 4.

The overly complex nature of having to support 4 logical processors may have contributed to the design time and ultimate failure of the EV8 processor becoming a real product. While this does not mean that a 4-way SMT cannot be done, the performance predictability and complexities of efficiently sharing resources are extremely difficult problems to solve. Complex sharing mechanisms are required to prevent structures from severely limiting the overall frequency of the processor. The problem is very similar to that encountered in trying to design thread priority mechanisms that improve overall throughput without compromising fairness and predictability. Thus far no mechanisms have been shown to be effective and tractable in dynamically sharing resources.

4.3.1 Front-end

The instruction fetch logic attempts to assemble 8 valid instructions each cycle. To do this, they fetch 16 instructions from two separate cache blocks, each 8 instructions wide. A collapsing block removes instructions not on the predicted path. In each clock cycle instructions are fetched for only one logical processor. The instruction cache is a shared structure. There are four separate program counters to track fetch progress for the four logical processors independently. During register renaming, there are four independent register alias tables. With 32 integer and 32 floating point architectural registers for each logical processor a total of 256 total architectural registers are required. The implementation has 512 physical registers to accommodate a total of 256 in-flight instructions. The register file takes 3 cycles to access due to transit delays and its size, so smaller register caches for the integer and floating point units were added to reduce the penalty of this latency. The register caches store copies of 8 cycles' worth of results. The renamed instructions are placed in a shared instruction queue, possibly allocated based on an I-count algorithm as described in [33].

4.3.2 Out-of-order Execution Engine

Presumably the instructions are issued to execution units and access the memory pipeline based on instruction dependencies. Instructions are retired in blocks of instructions in program order [67].

4.3.3 Memory Subsystem

The first and second level caches, and the translation buffers are shared by all logical processors [67].

4.4 Clearwater Networks CNP810SP Processor

The Clearwater Networks CNP810SP processor was another processor that spent years in the design but never taped out. It was also intended to be an SMT processor, but targeted specifically at the network processing market. The intention was to have eight logical processors executing simultaneously on a superscalar core capable of a executing a maximum of ten instructions per cycle [69]. Very little detail is publicly known about the Clearwater processor's microarchitecture implementation.

Again, supporting a large number of logical processors involved complexity and huge wiring challenges that may have contributed to its demise. However, since the Clearwater processor was narrowly targeted at a specific application class and market, it did not need to have the same level of fairness vs. throughput guarantees that the Pentium 4 required.

4.4.1 Front-end

A shared dual-ported instruction cache can supply up to eight instructions for each of two logical processors per cycle, for a maximum of sixteen instructions per cycle. The instructions are placed in separate instruction queues for each logical processor, so there are eight instruction queues, each can hold up to 16 instructions. Two logical processors are selected each cycle to access the instruction cache. The selection is based on which logical processors have the fewest number of instructions in their instruction queue.

Each logical processor has its own 31-entry register file. There is no sharing of registers between the logical processors.

4.4.2 Out-of-order Execution Engine

The logical processors are divided into two groups of four for scheduling and dispatch. The dispatch logic is therefore split into two groups, or "clusters". Each cluster consists of the dispatch logic, and four functional units. There are also two ports to the data cache that are shared by both clusters. Each cluster dispatch logic can send up to six instructions from the four different logical processors, where zero to three instructions can be executed from each logical processor depending on instruction dependencies and availability of resources. A maximum of ten instructions can be dispatched per cycle. The functional units are fully bypassed so that dependent instructions can be dispatched in successive cycles.

CHAPTER 5
SIMULTANEOUS MULTITHREADING MICROARCHITECTURE CHOICES AND TRADEOFFS

This chapter presents the results of our research on the sharing policies of key structures in the Pentium 4 microarchitecture.

The Intel Pentium 4 processor implementation of SMT required microarchitecture choices and tradeoffs with respect to the resource sharing policy for each shared resource[70] [71]. This chapter analyzes how the choice of sharing policy can impact performance dramatically. The policies discussed in this chapter included:

- *Partition* - dedicating equal resources to each logical processor;
- *Threshold* - flexible resource sharing with a limit on the maximum resource usage; and
- *Full sharing* - flexible resource sharing with no limit on the maximum resource usage.

The analysis and discussion covers evaluation of performance, throughput vs. fairness, potential livelock scenarios, as well as die size and complexity.

5.1 Partition

In a partitioned resource, each logical processor can use only half the entries. Clearly, resource partitioning has the advantage of simplicity and low complexity. It is a good choice for resources when you expect the structure’s utilization to be generally high and somewhat unpredictable. For example, partitioning is a good choice for the major pipeline queues, which provide buffering to avoid pipeline stalls and, ideally, remain full most of the time. However, because software thread execution speeds can differ at any instant in time, the rate at which the queues fill and empty is unpredictable. By partitioning these queues, we can allow slippage between a fast and a slow thread, preventing a slow thread from blocking or slowing down the faster thread and thereby making the best use of each pipeline stage.

Figure 21 illustrates how this works. At the start, Figure 21(a), both the shared queue (on the left) and the partitioned queue (on the right) have two light-shaded and two dark-shaded micro-ops. The light-shaded micro-ops belong to Thread 0, and the dark-shaded micro-ops belong to Thread 1. Both the light micro-ops and the dark micro-ops are labeled 0 and 1, representing the per-thread micro-op ID. Every micro-op is given a unique micro-op ID assigned in sequential order to distinguish it from other dynamic micro-ops in the pipeline. In Cycle 1, Figure 21(b), both the shared and partitioned queues send light micro-op 0 down to the next pipeline stage. In the shared queue, the previous pipeline stage sends dark micro-op 2, but in the partitioned queue, because the dark thread is already occupying its maximum number of entries, the previous pipeline stage sends a light micro-op instead (light micro-op 2). At the end of Cycle 1, the shared queue has one light micro-op and three dark micro-ops. The partitioned queue has two micro-ops of each shade.

In Cycle 2, Figure 21(c), both the shared and the partitioned queues send a light micro-op to the next pipeline stage, and the previous pipeline stage delivers a light micro-op in both cases. The shared queue gets a light micro-op in this cycle because in the previous cycle it sent a dark micro-op. In general, in-order pipeline stages will alternate between light and dark micro-ops unless the staging queue after the pipeline stage is full or the previous staging queue has no micro-ops available to work on.

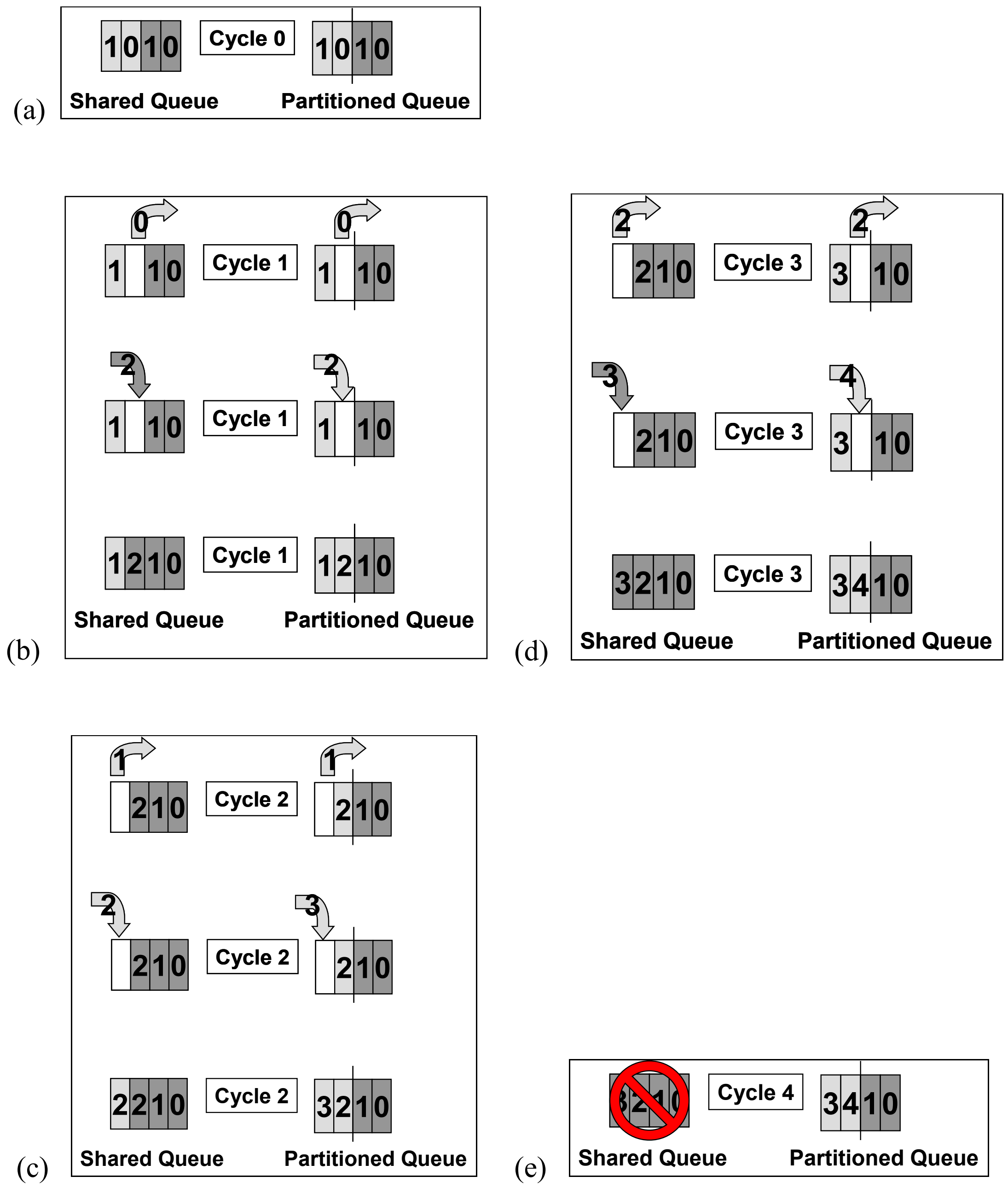

Figure 21. Comparison of a shared and a partitioned queue. The light-shaded micro-ops belong to Thread 0, and the dark-shaded micro-ops belong to Thread 1. The numbers in the boxes are the micro-op ID which are assigned sequentially to each thread's micro-ops. Thread 0 has a downstream stall, such as a data cache miss. In this situation, the queues will not send any slower micro-ops to the next pipeline stage. The figure shows how the queues will progress through cycles 0 (a), 1 (b), 2 (c), 3 (d), and 4 (e), where the shared queue lets the slower thread block the progress of the faster (light) thread.

In Cycle 3, Figure 21(d), both queues again send a light micro-op to the next pipeline stage. The previous pipeline stage sends a dark micro-op in the case of the shared queue and a light micro-op in the case of the partitioned queue. At the end of Cycle 3, the shared queue has four dark micro-ops and no light micro-ops, while the partitioned queue still has two of each.

In Figure 21(e), the shared queue is now blocked because it has no light micro-ops, and the dark thread has a downstream stall. The partitioned queue is thus a simple mechanism that can continue to issue light micro-ops. The partitioned queue prevents the pipeline from stalling.

In Section 4.1, Figure 18 showed a basic execution pipeline of the Pentium 4 microarchitecture. It is especially important to guarantee fairness and progress for the pipeline's in-order parts. Therefore, a partitioned scheme works best for the major pipeline queues in the in-order pipeline: the Instruction Fetch pipeline and Retirement. If there is a front-end stall (say, because of a trace-cache miss), the back-end can continue to take micro-ops from the micro-ops queue. If there is a back-end stall (say, because of a data cache miss), the front end can continue to fill the queue. Large queues can keep both the front end and the back end mostly busy when one end is temporarily stalled for one logical processor.

As illustrated in Figure 21, if the two logical processors fully shared these queues, a slow thread could gain an unfair share of the resources and prevent a fast thread from making progress. Because the slow thread is often stalled, its micro-ops start to pile up in the queues. In time, the slow thread will collect more and more entries, because it competitively shares entries with the fast thread. Eventually, the slow thread will get most, if not all, of the queue, thereby slowing the fast thread's progress. A partitioned queue, however, lets the fast thread always have half of the entries and advance at its own pace.

The use of partitioned resources is simple, entails little implementation complexity, and ensures fairness and progress for both logical processors.

5.2 Threshold

Another way of sharing resources is to limit the maximum resource usage that a logical processor can have. This approach is ideally suited for small structures where the resource utilization is bursty, and the length of time a micro-op stays in the structure is short, fairly uniform, and predictable. Processor schedulers provide an example of where threshold sharing is a good choice. Scheduler throughput on the Pentium 4 is high because they assume that load instructions will hit in the cache, so micro-ops don’t linger in the schedulers (a separate re-issue mechanism resubmits micro-ops to execution units in the event of a cache miss). Also, the schedulers are very small, to enable speed. They run at twice the clock frequency, so a 3 GHz processor has schedulers running at 6 GHz.

The allocation of micro-ops to these schedulers is round-robin until a logical processor reaches its threshold number of entries. At that point, it cannot allocate more micro-ops until it dispatches some of its current entries.

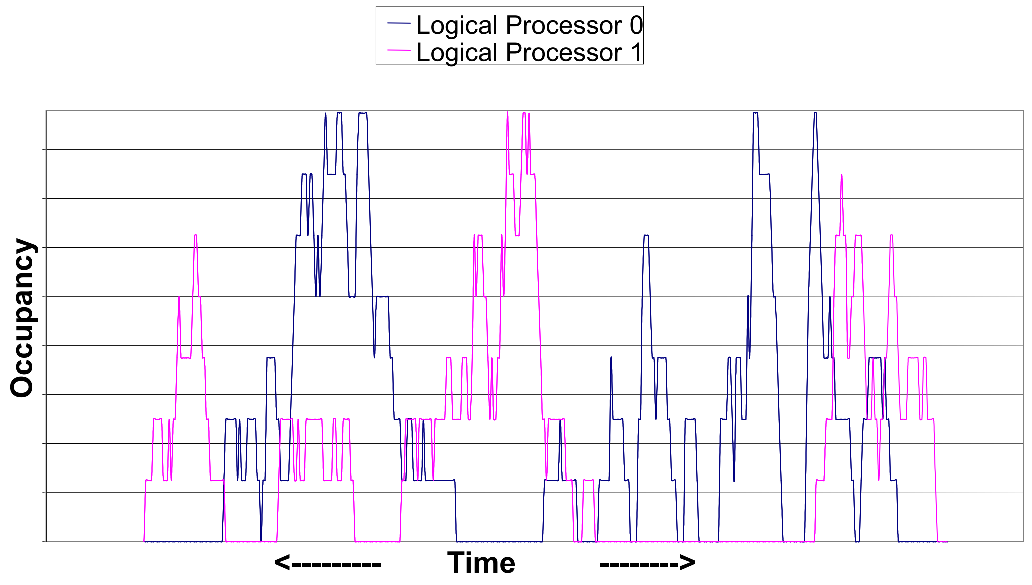

Figure 22. Snapshot of scheduler occupancy on a transaction processing workload over a short period of time. Each data point is the instantaneous scheduler occupancy for its respective logical processor, measured by the number of entries occupied by each thread.

Figure 22 shows scheduler occupancy over a number of processor clock cycles. Although average scheduler utilization is low, the activity can be bursty. A threshold limiting the maximum number of entries for each logical processor prevents one logical processor from blocking the other's access to the scheduler. The threshold lets the scheduler look for maximum parallelism among micro-ops across both threads, thereby improving execution resource utilization.

5.3 Full sharing

Fully shared resources, the most flexible mechanism for resource sharing, does not limit the maximum resource usage for a logical processor. In general, fully shared resources is a good mechanism for large structures in which working set sizes are variable, and one logical processor cannot starve the other.

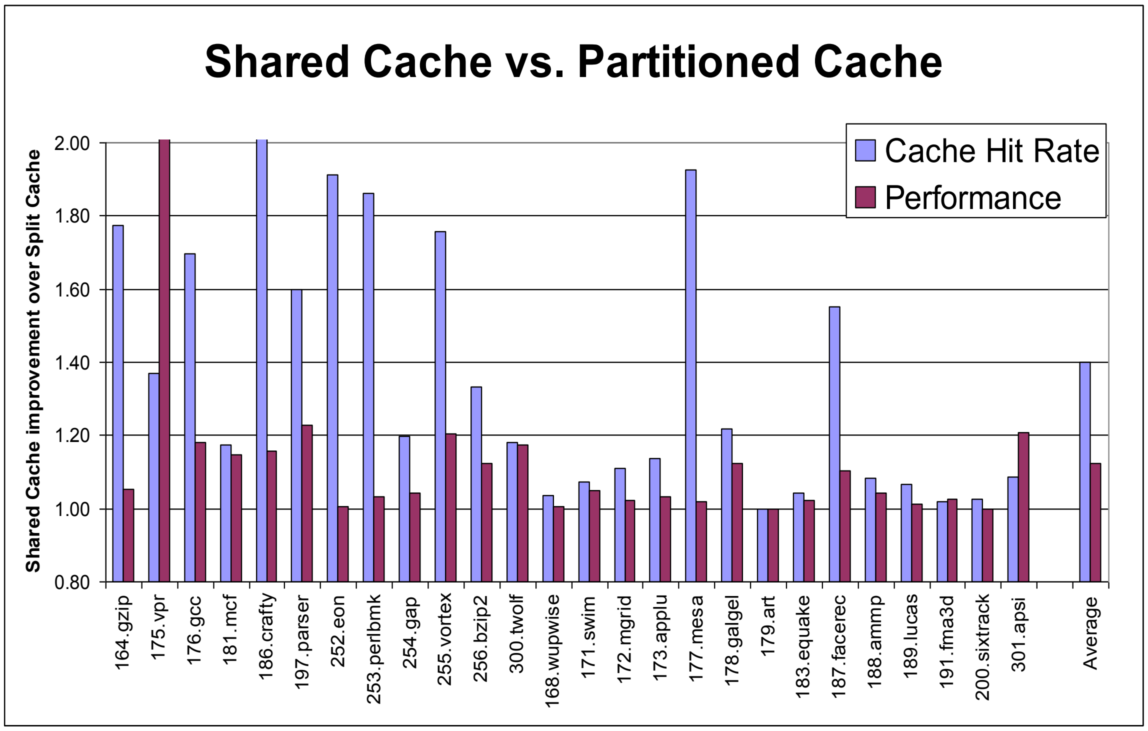

Figure 23. Cache hit rate and overall performance impact for a fully shared cache normalized against values for a partitioned cache. On average, the shared cache had a 40-percent better cache hit rate and 12-percent better performance. Notice that no single application workload lost performance because of the shared cache.

Processor caches are a good example of structures best suited to the full sharing policy. In the Pentium 4 microarchitecture, all processor caches are shared. This has several advantages. First, it allows for better overall performance than with a partitioned or threshold cache because cache interference is usually modest. Second, some applications benefit from a shared cache because they share code and data, minimizing redundant data in the caches. Finally, high set-associativity minimizes conflict misses between logical processors. The second- and third-level caches (if present) are eight-way set associative. Because SMT technology was a new architectural field, we implemented multiple resource management algorithms in some areas of the processor. This included the cache sharing policy. This feature lets us experiment with various cache management policies on real systems. Figure 23 shows results for some of those experiments and the advantage of using a shared cache. The figure compares the results of running multiple workloads on two cache configurations: fully shared and partitioned. For each workload, the figure shows the cache hit rate and performance impact of a fully shared cache normalized to those of a partitioned cache. We collected cache miss statistics using the Intel Pentium 4 event-monitoring counters [72], specifically the second-level cache's load-miss-retired event. The workload consisted of running two copies of the same application. This study highlights the modest cache interference in a shared cache.

5.4 Conclusions

With a resource sharing policy matched to the traffic and performance requirements of each resource, SMT can significantly increase resource utilization and improve performance. This research and development has also resulted in a number of related patents [73-82]. Continued research in academia [83] and industry [84, 85] has also continued to make progress in continuously improving the algorithms for sharing resources in a simultaneous multithreaded processor. Brayton [84] demonstrated Intel's latest SMT technology on the Nehalem processor, to start shipping in 2008, and Singhal [85] described it in more detail.

CHAPTER 6
POWER AND ENERGY ANALYSIS OF SIMULTANEOUS MULTI-THREADING

SMT is an energy-efficient method to get performance. In this chapter we analyze the additional power needed for a processor with SMT, and how much less energy SMT systems can use doing the same amount of work.

Power and energy [86] are frequently confused. Power is a measure of the rate of doing work or *using* energy. The common way of measuring electrical power is in watts. One watt is equal to one joule (J) of energy per second. Computer system and processor manufacturers and designers are often concerned about peak power and average power. In a computer system, peak power will determine the power supply and heat dissipation requirements for the processor. For a computer lab, the combined peak power for all components in the lab (computers, instruments, lights, fans, air conditioning, etc.) determines the peak power supply requirements for the room.

Energy is the capacity to do work. Energy can be stored, in a battery for example. A common electrical energy metric for batteries is the watt-hour. The standard MKS metric for energy is the joule (J), equal to one watt-second. One watt-hour is equal to 3600 joules. We can compute the energy requirement for a specific task. A battery that stores 200 watt-hours can keep a system that draws on average 50 watts of power, up and running for 4 hours. Energy requirements determine the battery life for battery-powered systems, or the energy cost to buy power from power companies. Laptop designers need to know the average power of a system to specify the battery requirements. This is of increasing importance as computers and handheld devices become more powerful and ever smaller.

We show that, although SMT power is typically ~5-15% higher than ST power, SMT can do the same amount of work using ***less*** energy. The higher the SMT speedup is, the more energy efficient it is. The SMT energy efficiency can be quite compelling when speedup is > 1.1.

6.1 Methodology

Our power and energy measurements were done on real systems. For the same amount of work, we compared the power and energy used with SMT on and SMT off. Every workload had at least two software threads/processes. When SMT is off, the operating system will context-switch the workload's threads/processes to share the single logical CPU. When SMT is on, the operating system will schedule the two threads to run simultaneously on the two logical CPUs.

The system used for these experiments was as follows:

- Hardware configuration
    - CPU: Intel® Pentium® 4 CPU 661 3.60 GHz (ES) 2MB L2 cache.
    - Bus speed: 800 MHz
    - Memory: 1.0 GB DDR2
    - Motherboard: Lakeport/ICH7
    - Chipset: Intel i975x Rev. c0
    - Southbridge: Intel 82801GB (ICH7)
    - BIOS: American Megatrends Inc. (AMI) version VVPLI763.86P

- Software configuration

  - Operating System: Windows XP Professional x64 Edition Version 5.2, Service Pack1

For SPEC2000, the libraries and compilers used are as follows:

- Intel C++ and Fortran Compiler 9.1 for 32-bit applications
- Microsoft Visual Studio 2005 for libraries
- SmartHeap Library version 8.0.

To measure power accurately, we made modifications to the motherboard to access the voltage and current drawn by the CPU.

A National Instruments Model SC-2345 with National Instruments Measurement and Automation Explorer Software was used to log the data at a rate of 1000 measurements/second. For our purposes, this is sufficient granularity.

A chiller (USTC Thermal Tool Quick Disconnect System Model # USTC-5502MX with USTC Thermal Tool PreChiller System Model # USTC-LC10MC050) was used to keep the CPU package temperature at a constant 25C because of the super-linear impact of temperature on leakage power. The actual leakage and its dependence on temperature varies widely depending on the size and type of circuit as well as many other design and process technology factors [87].

SMT was enabled/disabled through the BIOS. Enabling and disabling SMT through the BIOS ensures that the ITLB are usable by the single thread in single-thread mode.

6.1.1 Motherboard modifications to measure CPU power

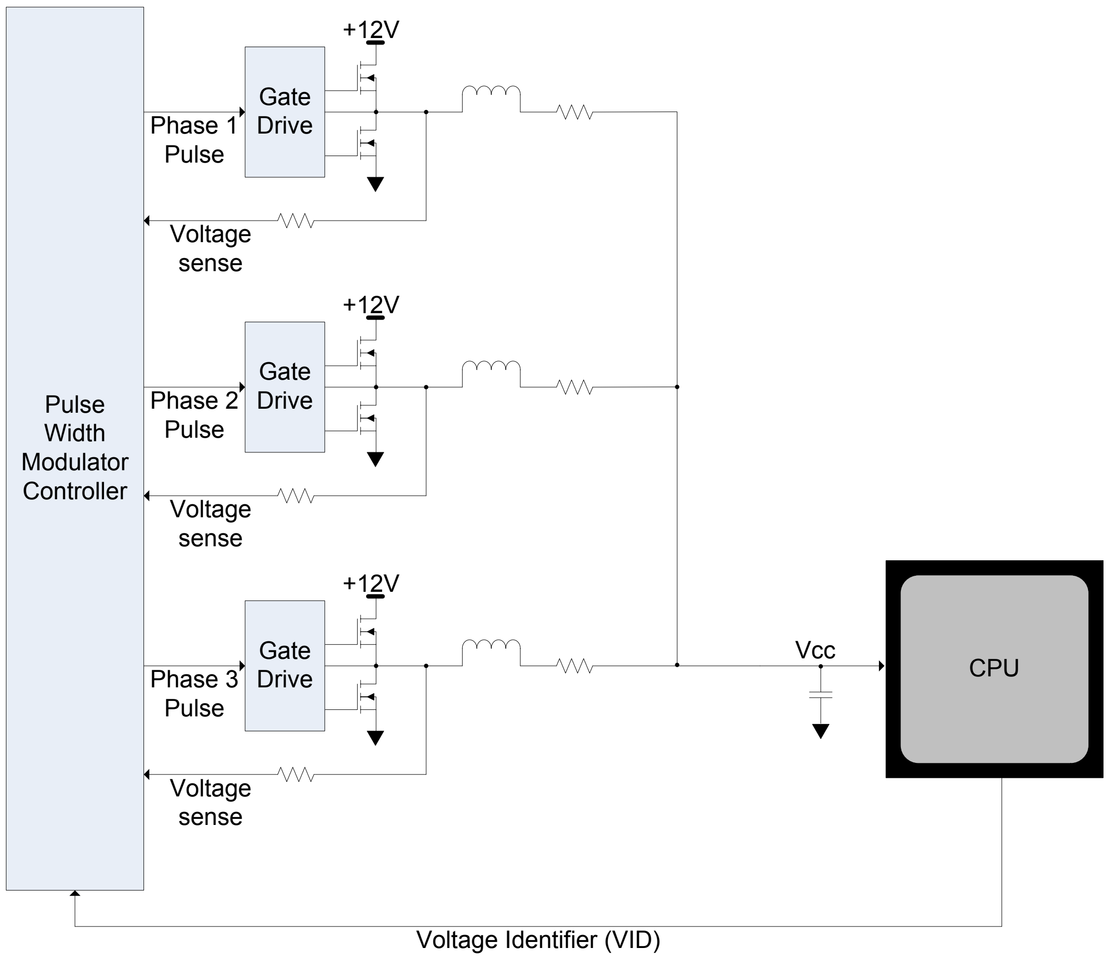

Figure 24. Three-phase voltage regulator on the Lakeport/ICH7 motherboard.

This section describes the modifications made to the motherboard voltage regulator for power measurements. To measure power at the CPU, we need the current and voltage delivered to the CPU silicon. We were fortunate to have available a system designed and built by Jim Hunt, of Intel Corporation [88].

Modifications were made to the voltage regulator to add a current sensor. Figure 24 shows the unmodified three-phase voltage regulator for the Pentium 4 processor. A three-phase voltage regulator has three identical interleaved cells connected such that their output is a summation of all individual voltage regulator cells. Each cell uses a synchronous rectifier consisting of two MOSFETs to convert the 12V input voltage to the Vcc voltage requested by the CPU via the encoded voltage identifier (VID) signal. The Pulse Width Modulator Controller sends pulses of a given width to each of the three phases in turn. The width of the pulses determines the output voltage. The "voltage sense" signal allows the Pulse Width Modulator Controller to fine-tune the pulse width for an accurate Vcc output.

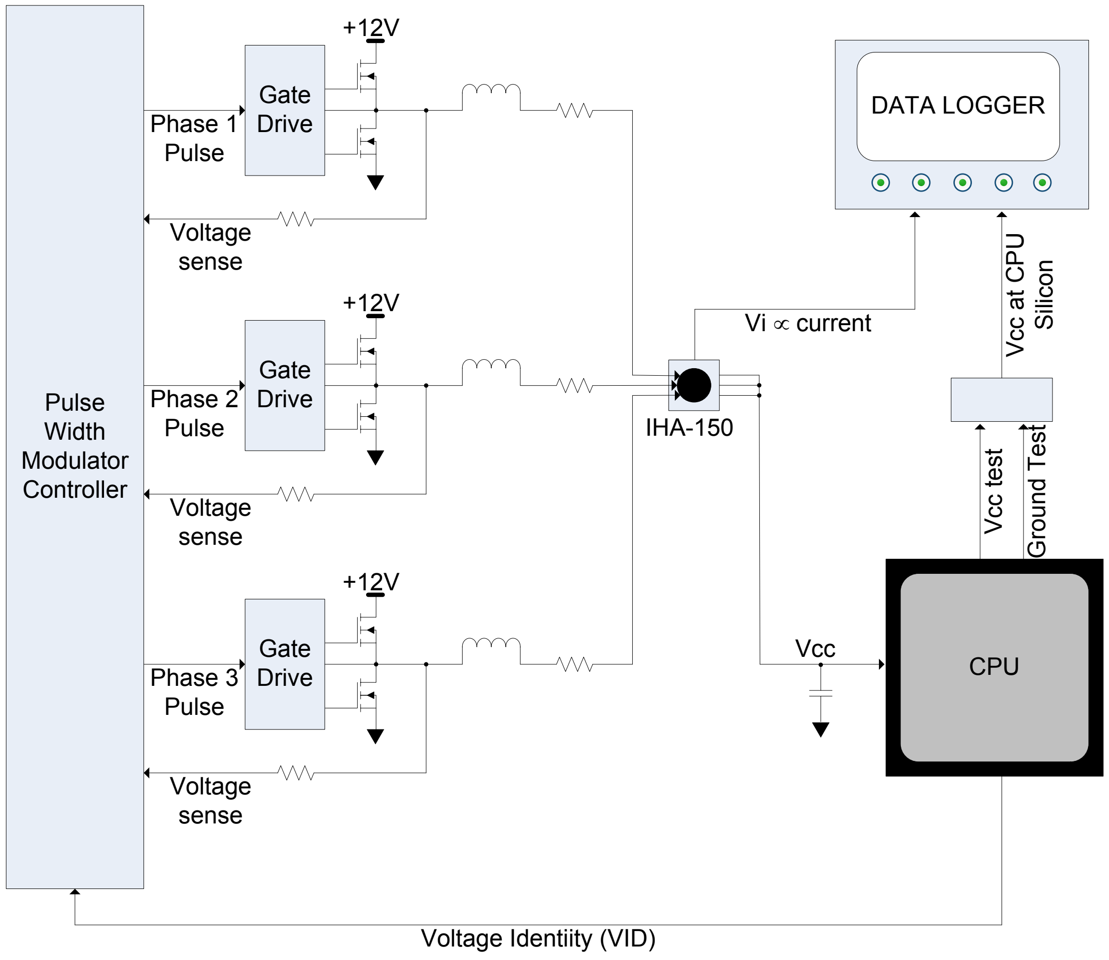

Figure 25. Voltage Regulator modifications to measure power at the CPU core.

Figure 25 shows the modifications to the voltage regulator that were done. The IHA-150 current sensor was chosen because of its accuracy and because it is a Hall effect [89] current sensor which provides electrical isolation of the current being sensed and the voltage output. The output wires of the three regulator cells are run through the hole in the IHA-150. The total electric current flowing through the wires creates a magnetic field which produces a voltage, the Hall voltage, which is proportional to the amount of current flowing through the wires. This voltage, Vi, is sent to our data logger system. The data logger logs the changes to Vi over time to tell us how much current is being drawn by the CPU at any point in time. The Vcc that the voltage regulator produces goes to the CPU package, then to the CPU silicon itself. By the time the Vcc gets to the actual CPU silicon, it has experienced some voltage droop so it is important to get an actual Vcc measurement from the CPU silicon itself. Fortunately, we have two test outputs to tell us effective Vcc and ground levels that the CPU is actually seeing. We route the difference between these two test signals to our data logger to get an accurate reading of effective Vcc.

Another important modification involved the load line. Because of the additional wires that were soldered to the voltage regulator cells, care had to be taken to make sure the lengths of those wires were all the same. The wire lengths were determined by the longest wire to get the signals through the IHA-150 and back. Because these wires put additional load on the Vcc lines, the resistors on the three voltage sense signals had to be swapped out and replaced with different size resistors to compensate for the additional load.

6.1.2 SMT Energy Benefit Metric

We need a metric to understand the energy benefit when SMT is enabled similar to the speedup metric for measuring performance benefit. Let us review the definition of speedup [90]. Speedup is defined as the execution time for the workload on a single processor divided by the execution time for the workload on multiple processors:

$$\text{Speedup} = T_1 / T_P$$

where T_1 is the execution time on a single processor, and T_P is the execution time on P processors.

In the case of SMT speedup, this means:

SMT Speedup = T_{ST} / T_{SMT}

Where T_{ST} is the execution time in single-thread mode, and T_{SMT} is the execution time in SMT-mode. If the execution time in SMT-mode is half the execution time in ST-mode, then the speedup would be 2.0.

Similarly, we define a metric called **SMT Energy Benefit** as follows:

SMT Energy Benefit = E_{ST} / E_{SMT}

Where E_{ST} is the total energy used to execute the workload in ST-mode, and E_{SMT} is the total energy used to execute the workload in SMT-mode. SMT Energy Benefit = 1 means that the same energy is required to do the work in ST mode as SMT mode. If SMT-mode takes half the energy compared to ST-mode, then the SMT Energy Benefit would be 2.0, i.e., we can do twice the amount of work on the same battery, or we doubled the battery life.

Since energy is equal to power multiplied by execution time, SMT Energy Benefit can also be calculated as follows:

SMT Energy Benefit = $(P_{ST} * T_{ST}) / (P_{SMT} * T_{SMT})$

Where P_{ST} and T_{ST} are the average power and total execution time in ST-mode, respectively, and P_{SMT} and T_{SMT} are the average power and total execution time in SMT-mode.

With the SMT Energy Benefit metric, a value >1 means that the total energy with SMT enabled improves battery life. A value of 2.0 means that we can expect to double the battery life needed for that workload.

6.2 Micros Study Results

SMT power efficiency is excellent on the Pentium 4 processor. To test the extent of this efficiency, we wrote and ran several micro-benchmarks, or micros. Micros are small simple programs, often written in assembly language, designed to exercise and/or isolate a specific feature of the CPU for better understanding of processor behavior.

We wrote and ran two types of micros: throughput and latency micros. Throughput micros use a particular resource at or near 100%. When both logical processors are attempting to saturate the resource, we expect SMT speedup to be around 1.0. On the other hand, latency micros measure the latency of a particular type of resource. Since some resources are pipelined, those resources are <=50% busy in latency micros. In these cases, we can expect an SMT speedup of about 2.0 when both logical processors are simultaneously testing the latency of that resource. Micros with SMT speedups around 1 (no performance benefit) and 2 (performance doubles with SMT) are particularly interesting for our analysis because they lead to an intuitive understanding of the relationship between performance, power, and energy.

The micros used in the following experiments have 2 threads. Both threads are doing the same thing. When run in ST-only mode, the operating system context-switches the threads. In SMT-enabled mode, the operating system schedules the two threads to run on the two logical processors simultaneously.

Table 6 shows the results of several throughput micros whose SMT speedup is ~1.0. The expectation is that the SMT power increase would be approximately the area growth for enabling SMT. Looking at the measured results, although there is no performance benefit for these micros, enabling SMT results in a small increase in power of 3-5%, or 1-2 Watts. This is precisely as expected! The increase in power is due to (1) enabling additional structures required for SMT such as architectural resources, a second set of rename tables, queues in the front-end, etc., and (2) logic that monitors each thread's forward progress and controls thread switching at the various pipestages.

Table 6. Throughput Micro SMT speedup measurements.

	fmul through-put	if-then loop	L0 cache load hit through-put	noop loop	padd through-put	fadd through-put
SMT Speedup	1.00	0.96	0.95	0.92	1.01	1.00
ST Power (Watts)	28.3	34.4	36.7	30.4	28.4	28.6
SMT Power (Watts)	29.4	35.8	38.3	32.2	29.9	30.2
Power Difference (Watts)	1.0	1.4	1.6	1.8	1.5	1.6
Power Difference %	3%	4%	4%	5%	5%	5%
SMT Energy Benefit (ST Energy/SMT Energy)	0.966	0.923	0.910	0.871	0.957	0.953
ST CPI	1.87	0.56	0.47	0.38	0.95	0.93
SMT CPI	1.87	0.55	0.50	0.41	0.94	0.94

Some of the very low-CPI throughput micros have a small performance loss due to too-small buffers between major pipestages. On the Pentium 4 processor the buffers were sized for ST performance and therefore when half the buffers are allocated to each logical processor they are too small for SMT performance. The inefficiencies are due to the latency of signaling "buffer full" and "buffer not full" to prior pipestages that are feeding the buffers. "Buffer full" must be signaled early enough to accommodate all potentially in-flight uops coming from previous pipestages. This results in times when the buffer is not actually full when we signal "buffer full". If later pipestages start to empty the buffer at maximum rate, in some cases the buffers can be empty before the "buffer not full" signal to the previous pipestage can restart and deliver new uops. While the size of the buffers are appropriate in ST-only mode to hide these inefficiencies, in MT-enabled mode where each logical processor may have only half the entries, the size of the buffers may be insufficient. Fortunately, while these inefficiencies tend to show up on micros with very low CPIs, they rarely impact real applications.

The SMT Energy Benefit row in Table 6 is as defined in Section 6.1.2, the ratio of energy expended in ST-mode divided by the energy expended in SMT-mode. A value >1 means that the total energy with SMT enabled improves battery life. For the throughput micros, the ratios are all <1 meaning that it is better not to enable SMT for these workloads. This is intuitively correct since we have no performance benefit.

Table 7 shows several micros whose SMT speedup is ~2.0. These are the latency micros. In these micros, the performance doubles with SMT enabled. The power increase ranges from 7-18%, or 2-6 Watts. The L1 cache latency micro has the largest increase in power because accessing the 2MB L1 cache is high power due to powering up and enabling the large cache banks.

Table 7. Latency Micro SMT speedup measurements.

	fmul latency	L1 cache load hit latency	padd latency	fadd latency	imul latency
SMT Speedup	2.00	1.99	2.00	2.00	2.00
ST Power (Watts)	26.7	31.7	28.7	27.1	26.6
SMT Power (Watts)	28.6	37.5	31.8	29.4	28.5
Power Difference (Watts)	1.9	5.8	3.1	2.3	1.9
Power Difference %	7%	18%	11%	8%	7%
SMT Energy Benefit (ST Energy/SMT Energy)	1.866	1.679	1.804	1.845	1.868
ST CPI	7.25	1.97	1.87	5.42	8.84
SMT CPI	3.65	0.99	0.94	2.74	4.47

The expectation would be that the dynamic power increase should be proportional to the increase in activity due to SMT. To see if this is true, we need to graph the slope of the increase in power difference vs. ST power increase, and see if the slope of the latency micros is approximately equal to one.

Figure 26 plots the data from the two tables. The graph shows that indeed the latency micros have a slope of one! It also shows visually that the throughput micros (where the SMT speedup ~ 1) are 3-5% higher as expected. Also note that the latency micros, where SMT speedup = 2, tend to be lower-power due to lower utilization of CPU resources, but there is more increase in power from SMT due to doubling the dynamic usage of resources.

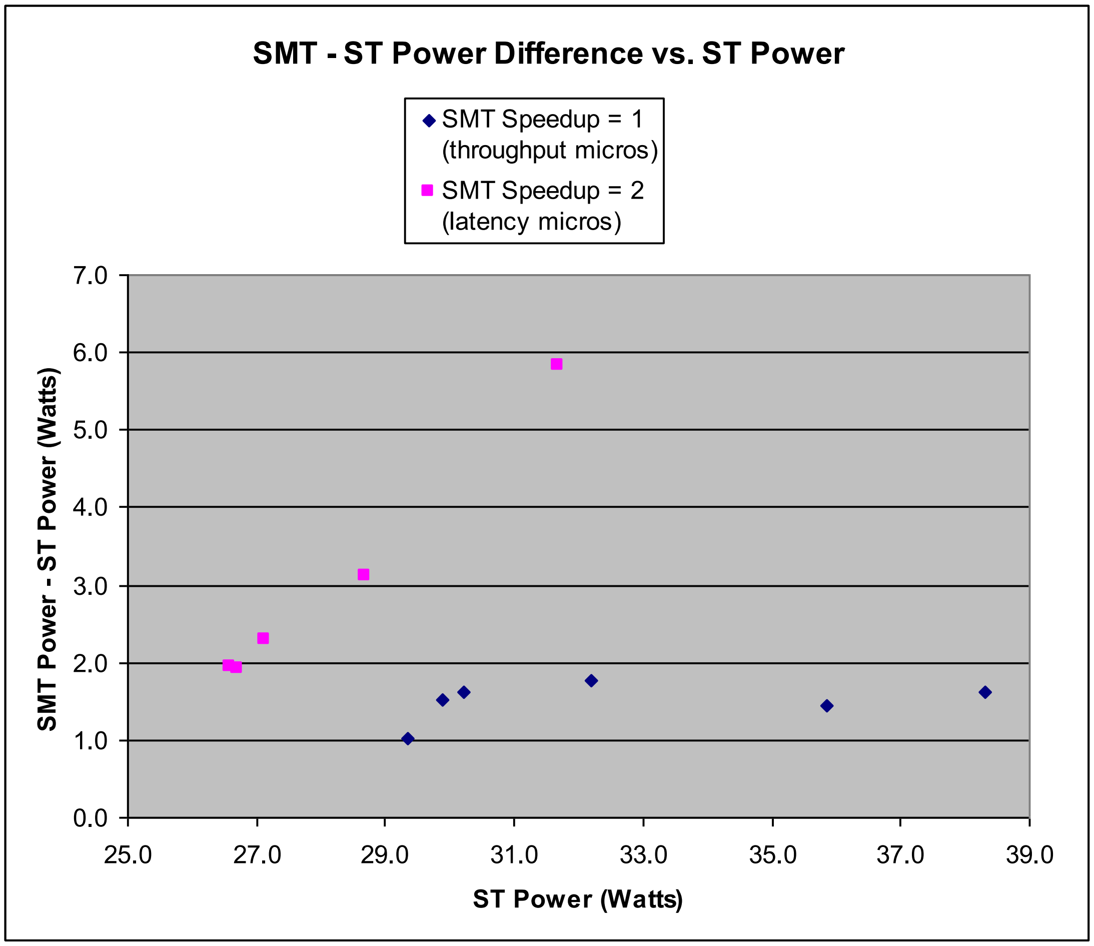

Figure 26. Throughput Micros and Latency Micro measured SMT speedups on an Intel Pentium 4 Processor System.

Another important metric to look at is the SMT Energy Benefit, which tells us whether batter life improves with SMT. The dramatic performance improvement but small power increase means that it is much better to enable SMT. We get up to 1.9x energy improvement, or 1.9x improvement in battery life!

Figure 27 shows the SMT Energy Benefit, and graphs this vs. ST Power for all the micros. Here, we see in graph form the huge potential SMT Energy Benefit for the latency micros are ranging from 1.7 to 1.9 when SMT speedup is close to 2.0. We can do 70-90% more work on the same battery! For the throughput micros whose speedup is ~1.0, the SMT energy benefit is <1 due to the power increase of 3-5% for the same execution time, so we can do less work on the battery.

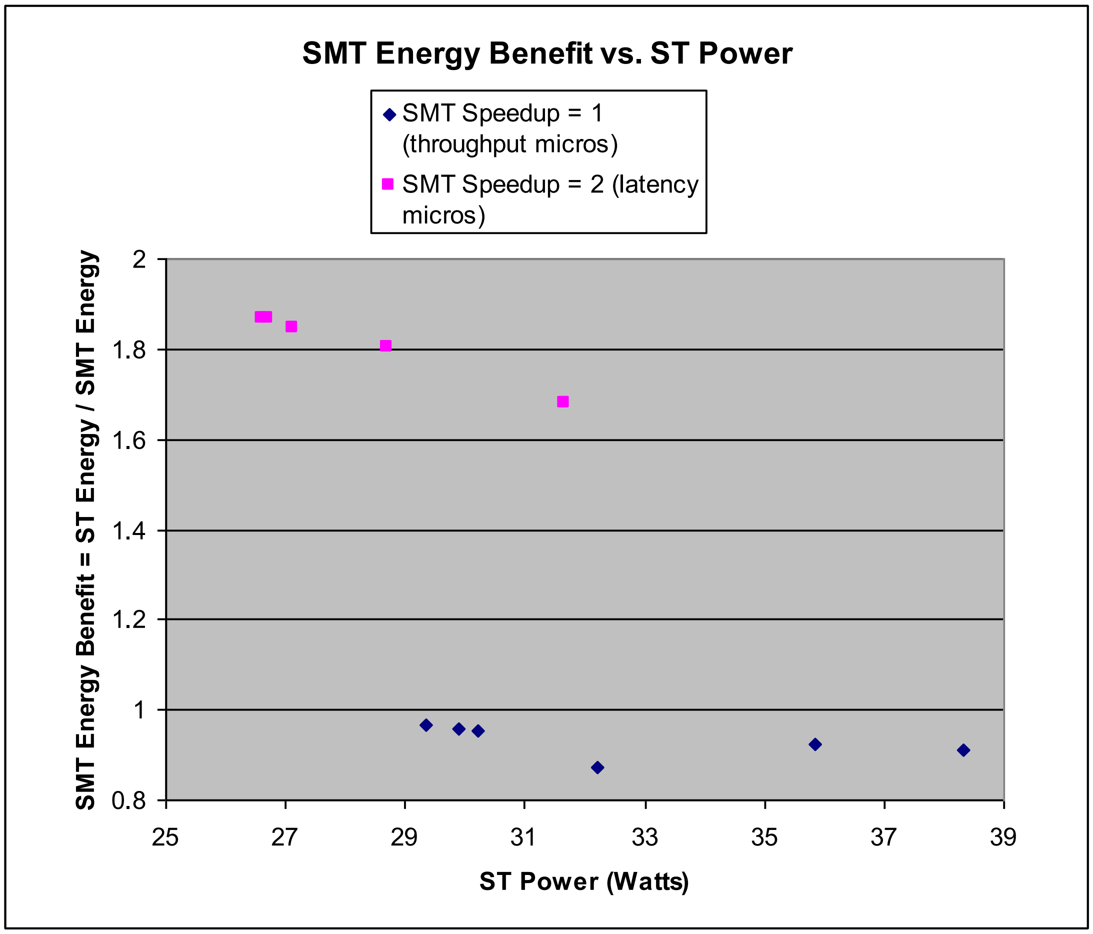

Figure 27. Throughput Micros and Latency Micros measured SMT Energy benefit on an Intel Pentium 4 Processor System. SMT Energy Benefit is extremely compelling for the latency micros.

The important take-away from this analysis is that the increase in power from SMT is low compared to the potential speedup. The conclusion is that the potential SMT power efficiency of the Pentium 4 processor is excellent. Finally, we would like to plot the SMT Energy Benefit vs. SMT Speedup for the micros. This is shown in Figure 28. The better the SMT speedup of the workload, the higher the SMT energy benefit, or the longer the battery life.

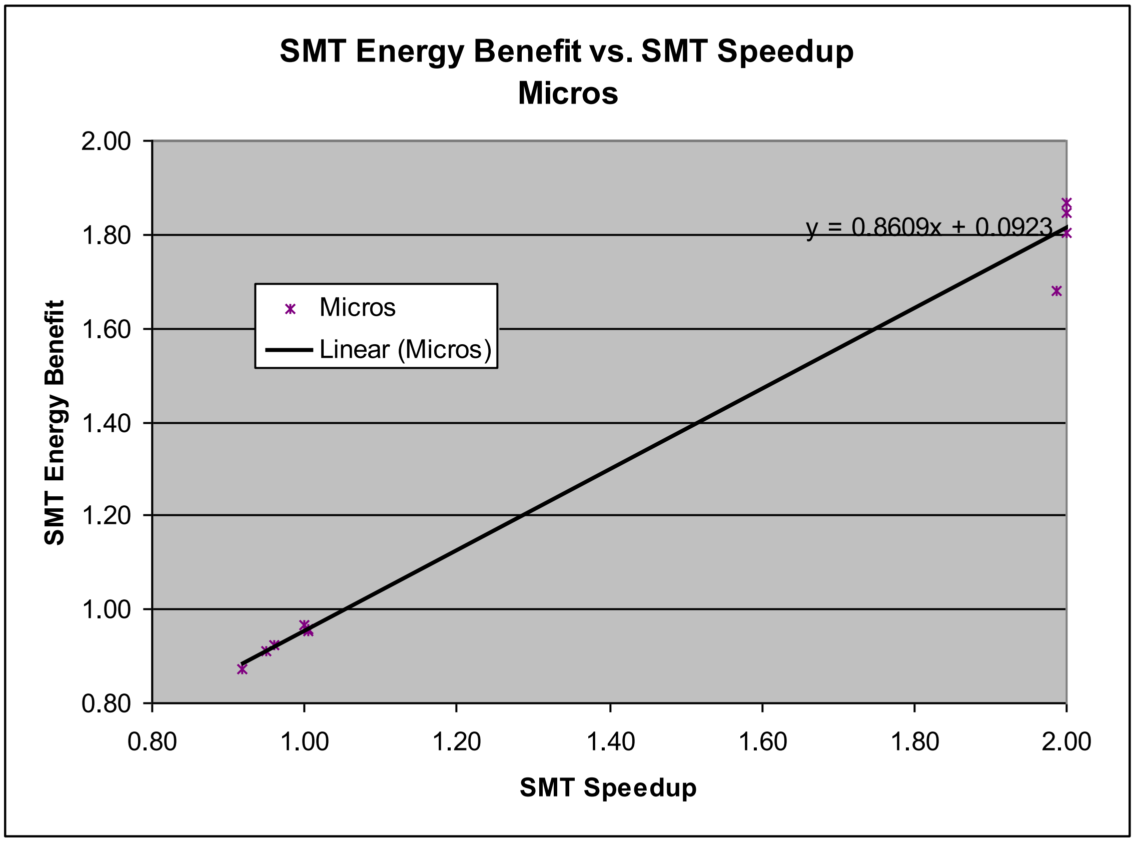

Figure 28. SMT Energy Benefit vs. SMT Speedup shows a slope

In the next few sections, we will look at the SMT Energy Benefit of real workloads, and see how they compare to the microbenchmark data.

6.3 SPEC 2000 Integer

Armed with the intuition from the micros analysis, we would like to now see what the performance vs. power and energy of other applications look like.

Table 8. SPECint_rate performance measured on an Intel Pentium 4 system.

Base Benchmarks	# Copies	Base Run Time	Base Rate	Peak # Copies	Peak Run Time	Peak Rate
SMT Off:						
164.gzip	2	232	14	2	228	14.2
175.vpr	2	233	13.9	2	232	14
176.gcc	2	108	23.7	2	108	23.7
181.mcf	2	174	24	2	174	24
186.crafty	2	150	15.5	2	151	15.4
197.parser	2	259	16.1	2	258	16.2
252.eon	2	107	28.3	2	108	28
253.perlbmk	2	168	24.8	2	169	24.7
254.gap	2	110	23.2	2	110	23.2
255.vortex	2	122	36.1	2	122	36.1
256.bzip2	2	226	15.4	2	226	15.4
300.twolf	2	329	21.2	2	329	21.2
SPECint_rate_base2000			20.4			
SPECint_rate2000						20.5
SMT on:						
164.gzip	2	176	18.4	2	195	16.7
175.vpr	2	228	14.3	2	228	14.2
176.gcc	2	93.2	27.4	2	93.2	27.4
181.mcf	2	190	22	2	190	22
186.crafty	2	147	15.8	2	156	14.9
197.parser	2	217	19.3	2	236	17.7
252.eon	2	100	30.1	2	107	28.2
253.perlbmk	2	182	23	2	185	22.6
254.gap	2	99.6	25.6	2	99.6	25.6
255.vortex	2	127	34.7	2	127	34.7
256.bzip2	2	175	19.8	2	174	20
300.twolf	2	274	25.4	2	276	25.3
SPECint_rate_base2000			22.3			
SPECint_rate2000						21.7

Before we started collecting our power measurements, we did an "official" SPECrate 2000 run to make sure that the overall system performance matches that of the published numbers on the official SPEC website. Table 8 shows the results. These SPEC numbers are as expected, within 1% of the published numbers of equivalent systems.

For the power and energy measurements, we combined the SPEC 2000 applications in two ways. The first was the way SPECrate is measured, where two copies of the same application are run at the same time. The second way was to combine different applications, for example gap and gcc. To minimize idle time (where one logical processor is active and the other is idle) one thread would run gap followed by gcc, and the other thread would run them in reverse order, gcc followed by gap. Since the applications have different run times, this means that there is some overlap time when the same application is running on both threads, one near the end and the other just beginning. We then compared the performance, power, and energy with SMT on and off.

Table 9 shows the data. SMT speedup is ST time / SMT time. The speedup ranges from 0.92 (performance loss) to 1.36. The lower speedup numbers tend to be SPECrate data because the same workload is run twice and an optimized workload should use 100% of some critical resource. So if the workload is highly optimized we wouldn't expect any speedup, but would expect some performance loss due to conflicts in the cache/TLB/branch predictors. In general, running combinations of applications would give better speedups because two different applications are more likely to stress different physical resources.

Table 9. SPECint SMT Power measurements for two copies of the same application and combinations of different applications.

		SMT speedup (ST time/SMT time)	ST Time (sec)	SMT Time (sec)	ST Power (Watts)	SMT Power (Watts)	Power Difference	Power Diff %	ST Energy (Watt-Hrs)	SMT Energy (Watt-Hrs)	Relative Energy Benefit (ST Energy / SMT Energy)
SPEC Rate Integer	mcf	0.92	176	191	34.7	35.3	0.5	1%	1.69	1.87	0.91
	perlbmk	0.97	170	176	39.0	41.1	2.2	6%	1.84	2.01	0.92
	vortex	0.97	123	127	36.7	39.5	2.8	7%	1.26	1.39	0.90
	crafty	1.02	150	146	37.1	40.8	3.7	10%	1.54	1.66	0.93
	vpr	1.05	235	224	36.4	38.1	1.7	5%	2.37	2.37	1.00
	eon	1.06	106	101	36.2	41.0	4.7	13%	1.07	1.15	0.94
	gap	1.11	110	99	38.5	41.6	3.1	8%	1.18	1.15	1.03
	gcc	1.15	108	93	36.8	40.1	3.4	9%	1.10	1.04	1.05
	parser	1.18	259	219	38.1	41.0	2.9	8%	2.74	2.49	1.10
	twolf	1.20	328	275	36.5	40.5	4.0	11%	3.33	3.09	1.08
	gzip	1.32	233	177	38.6	42.0	3.4	9%	2.49	2.06	1.21
	bzip2	1.32	231	175	37.8	40.2	2.3	6%	2.43	1.96	1.24
SPEC Integer Combos	crafty_perlbmk	1.11	321	288	38.0	41.9	3.9	10%	3.39	3.36	1.01
	eon_vortex	1.14	231	204	36.4	41.0	4.5	12%	2.34	2.32	1.01
	perlbmk_eon_gap	1.16	390	337	38.0	42.0	4.0	11%	4.12	3.93	1.05
	vortex_bzip_twolf	1.22	684	562	36.8	40.2	3.4	9%	7.00	6.28	1.11
	mcf_crafty_parser	1.24	585	470	36.7	40.1	3.3	9%	5.97	5.23	1.14
	bzip_vpr	1.25	473	380	37.0	39.5	2.5	7%	4.86	4.16	1.17
	parser_gcc_twolf	1.26	714	568	37.0	40.9	4.0	11%	7.34	6.46	1.14
	gap_gcc	1.26	240	190	37.4	41.1	3.7	10%	2.49	2.18	1.14
	mcf_perlbmk	1.26	347	275	36.7	40.0	3.3	9%	3.54	3.06	1.16
	gzip_vpr	1.30	477	368	37.4	40.8	3.5	9%	4.96	4.17	1.19
	gzip_vpr_gcc	1.30	610	468	37.2	40.7	3.6	10%	6.30	5.30	1.19
	bzip_gzip	1.36	462	340	38.0	41.6	3.6	9%	4.88	3.93	1.24

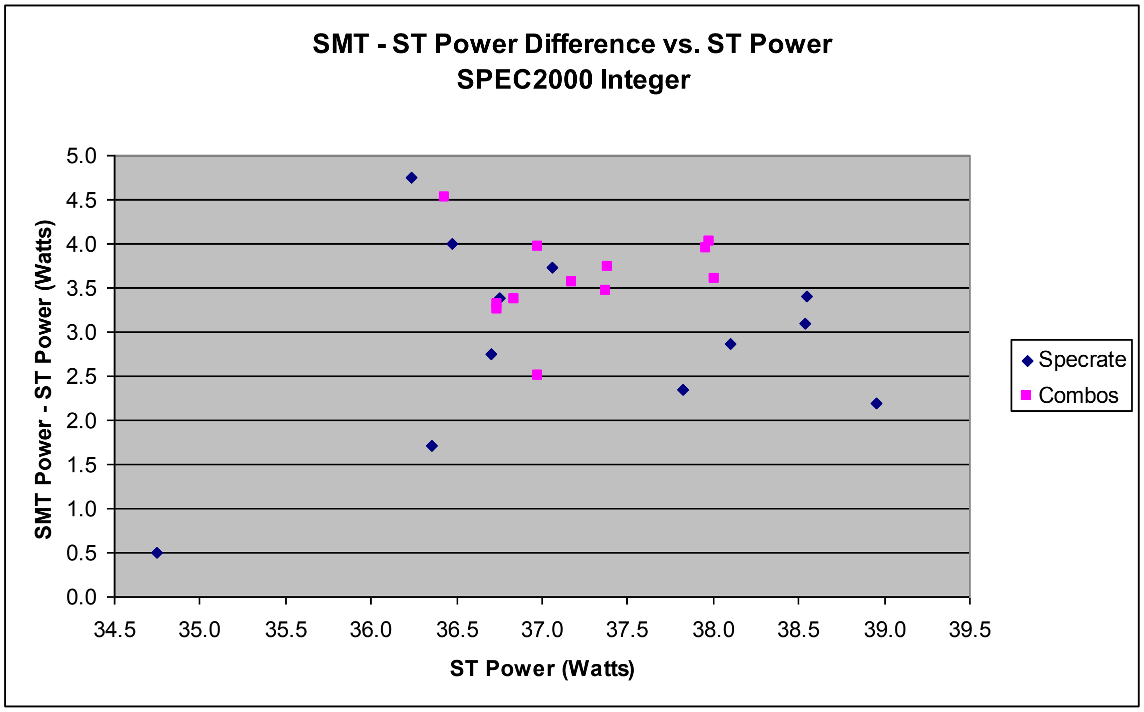

Figure 29. SPEC2000 Integer SMT-ST Power Difference vs. ST Power

We were interested in how much more power was consumed when running the workload with SMT on vs. off.

Figure 29 shows the increase in power when run in SMT-enabled mode. The power increase ranges from 0.5-5 watts, and is in the range expected based on our previous experiments on micros.

The next questions to ask are whether the power increase or energy efficiency of these workloads is impacted by SMT speedup?

Figure 30 shows the relative SMT power increase (SMT power / ST power) vs. SMT speedup (ST execution time / SMT execution time). At first glance, it looks like SMT power is on average 7-8% higher than ST. This makes sense because when

SMT is enabled, we have twice the number of key structures active (e.g. register alias tables), and conflicts between the threads are likely to cause additional work due (e.g. cache misses).

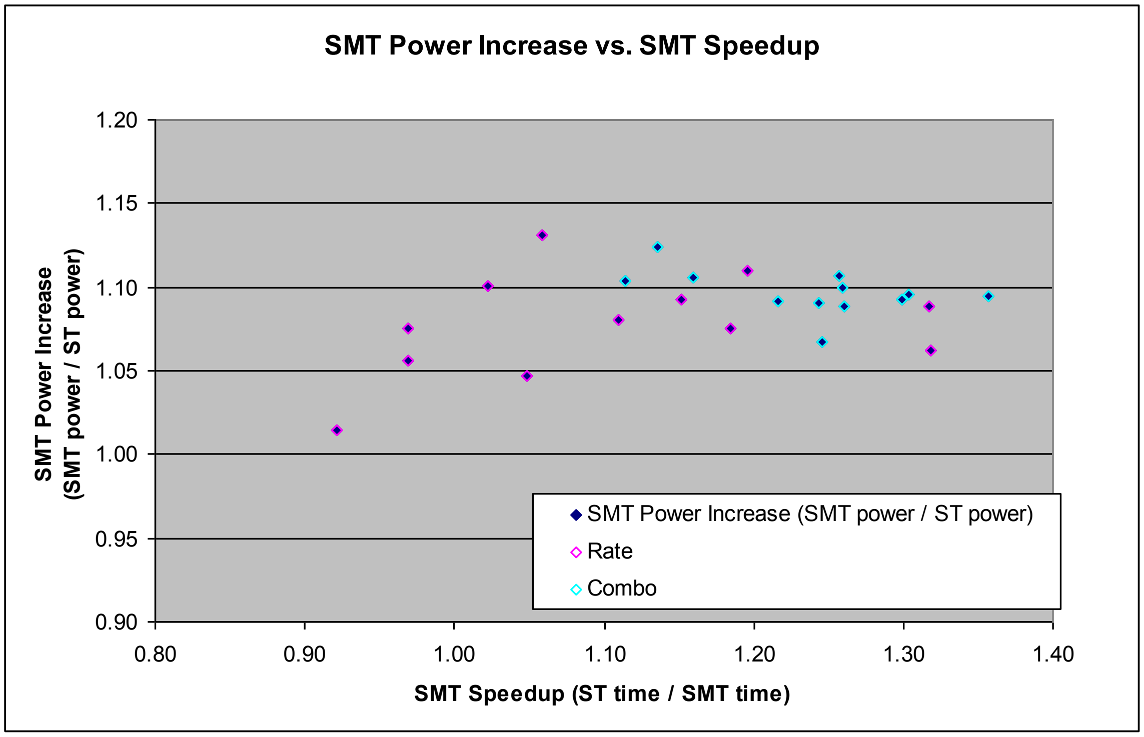

Figure 30. SMT Power Increase vs. SMT Speedup for SPECint

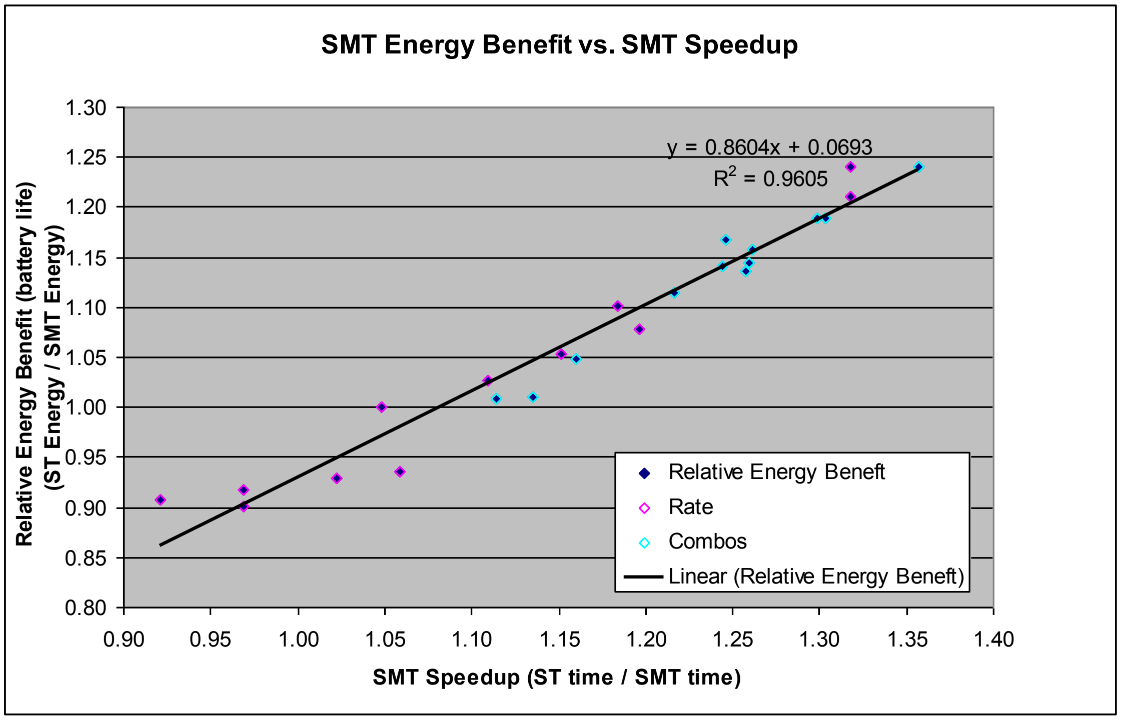

Figure 31. SMT Energy Benefit vs. SMT Speedup for SPECint

Finally, we calculated the SMT Energy Benefit, a metric related to battery life, as defined in section 6.1.2. Figure 31 shows the SMT Energy benefit of the SPEC 2000. As shown, there is a very strong correlation between SMT speedup and the energy efficiency. This is because the SMT power increase has very little increase with SMT speedup.

Energy **efficiency** is defined as a value between 0 and 1 estimating how well the SMT performance speedup is translated into SMT energy benefit. To find the energy efficiency of SMT for the SPEC2000 workloads, we can do a linear fit to the points in Figure 31. The slope is the energy efficiency, and as shown is 0.86. As the SMT speedup improves, 0.86 of that speedup transfers directly to improved energy savings! And 0.14 (1 – 0.86) of the SMT speedup is wasted due to inefficiencies such as additional cache misses, extra hardware, and other things. SMT is extremely energy efficient!

6.4 SPEC2000 Floating Point

Next, we ran the SPEC 2000 floating point applications.

Table 10. Performance of SPEC2000_fp_rate with and without SMT enabled on an Intel Pentium 4 Processor System.

Base Benchmarks	# Copies	Base Run Time	Base Rate	Peak # Copies	Peak Run Time	Peak Rate
SMT Off:						
168.wupwise	2	83.3	44.6	2	83.2	44.6
171.swim	2	252	28.5	2	252	28.5
172.mgrid	2	230	18.1	2	230	18.1
173.applu	2	241	20.2	2	241	20.2
177.mesa	2	187	17.4	2	187	17.4
178.galgel	2	155	43.3	2	162	41.5
179.art	2	110	54.9	2	110	54.9
183.equake	2	87.3	34.5	2	91.3	33
187.facerec	2	182	24.2	2	190	23.2
188.ammp	2	346	14.7	2	355	14.4
189.lucas	2	185	25	2	193	24.1
191.fma3d	2	269	18.1	2	269	18.1
200.sixtrack	2	309	8.26	2	324	7.87
301.apsi	2	379	15.9	2	399	15.1
SPECfp_rate_base2000			23.3			
SPECfp_rate2000						22.8
SMT on:						
168.wupwise	2	78	47.6	2	77.8	47.7
171.swim	2	247	29.1	2	247	29.1
172.mgrid	2	203	20.5	2	203	20.5
173.applu	2	238	20.4	2	238	20.4
177.mesa	2	179	18.1	2	179	18.1
178.galgel	2	177	37.9	2	178	37.8
179.art	2	146	41.2	2	146	41.2
183.equake	2	79	38.2	2	80.4	37.5
187.facerec	2	162	27.3	2	160	27.5
188.ammp	2	333	15.3	2	325	15.7
189.lucas	2	177	26.3	2	177	26.3
191.fma3d	2	250	19.5	2	250	19.5
200.sixtrack	2	250	10.2	2	246	10.4
301.apsi	2	327	18.5	2	337	17.9
SPECfp_rate_base2000			24.3			
SPECfp_rate2000						24.3

Table 11. Power measurements of SPEC2000_rate_fp on an Intel Pentium 4 Processor System.

		SMT speedup (ST time/SMT time)	ST Time (sec)	SMT Time (sec)	ST Power (Watts)	SMT Power (Watts)	Power Difference	Power Diff %	ST Energy (Watt-Hrs)	SMT Energy (Watt-Hrs)	Relative Energy Benefit (ST Energy / SMT Energy)
SPEC Rate Floating Point	art	0.73	107	146	36.1	35.9	-0.2	-1%	1.07	1.45	0.74
	galgel	0.91	156	172	38.4	39.8	1.4	4%	1.67	1.90	0.88
	applu	0.98	232	236	36.1	38.0	2.0	5%	2.32	2.50	0.93
	swim	1.02	252	248	34.0	34.5	0.5	1%	2.38	2.38	1.00
	mesa	1.03	190	184	36.7	40.6	3.8	10%	1.94	2.07	0.94
	fma3d	1.05	257	245	35.9	39.0	3.1	9%	2.56	2.65	0.97
	lucas	1.05	186	177	36.3	38.5	2.2	6%	1.87	1.89	0.99
	ammp	1.09	352	324	35.5	37.3	1.7	5%	3.47	3.36	1.04
	equake	1.10	88	80	36.4	38.6	2.2	6%	0.89	0.86	1.03
	mgrid	1.12	229	204	37.7	40.8	3.1	8%	2.40	2.32	1.04
	facerec	1.15	182	158	36.4	39.7	3.2	9%	1.84	1.74	1.06
	wupwise	1.16	84	73	36.4	39.2	2.8	8%	0.85	0.79	1.07
	apsi	1.17	381	327	35.7	39.4	3.7	10%	3.78	3.58	1.06
	sixtrack	1.20	301	251	36.7	41.1	4.4	12%	3.07	2.86	1.07
SPEC Floating Point Combos	galgel_art	0.86	263	306	37.5	38.5	1.0	3%	2.74	3.27	0.84
	art_equake_facerec	1.11	378	340	36.3	39.1	2.9	8%	3.81	3.70	1.03
	mgrid_applu	1.12	464	414	36.8	40.0	3.2	9%	4.74	4.60	1.03
	applu_mesa_galgel	1.13	589	523	36.9	40.2	3.3	9%	6.04	5.84	1.03
	swim_wupwise_mgrid	1.14	569	499	35.9	37.9	2.0	6%	5.67	5.25	1.08
	wupwise_equake	1.15	172	150	36.4	39.2	2.7	8%	1.74	1.63	1.07
	lucas_fma3d_ammp	1.17	790	674	35.8	38.3	2.5	7%	7.86	7.16	1.10
	facerec_lucas	1.18	368	312	36.3	39.7	3.3	9%	3.71	3.44	1.08
	mesa_lucas	1.19	376	317	36.5	40.4	3.9	11%	3.81	3.56	1.07
	facerec_mesa	1.19	376	315	36.6	40.3	3.7	10%	3.82	3.53	1.08
	swim_fma3d	1.20	510	426	35.0	37.4	2.4	7%	4.97	4.42	1.12
	ammp_apsi	1.27	725	571	35.6	39.1	3.4	10%	7.17	6.20	1.16
	sixtrack_mgrid_equake	1.29	629	486	36.9	41.7	4.8	13%	6.45	5.63	1.15
	sixtrack_apsi_wupwise	1.30	772	596	36.2	41.3	5.1	14%	7.76	6.83	1.14

Again, to ensure that the performance of our system is consistent with equivalent systems, we compared the SPECrate for floating point applications with published results. Table 10 shows the results of our runs. These are comparable (within 1%) with published results.

Just as we did for the integer applications, for our power measurements we ran the floating point applications two different ways. The way Specrate is run, where we run the same application on both threads, and in combination.. Table 11 shows the results.

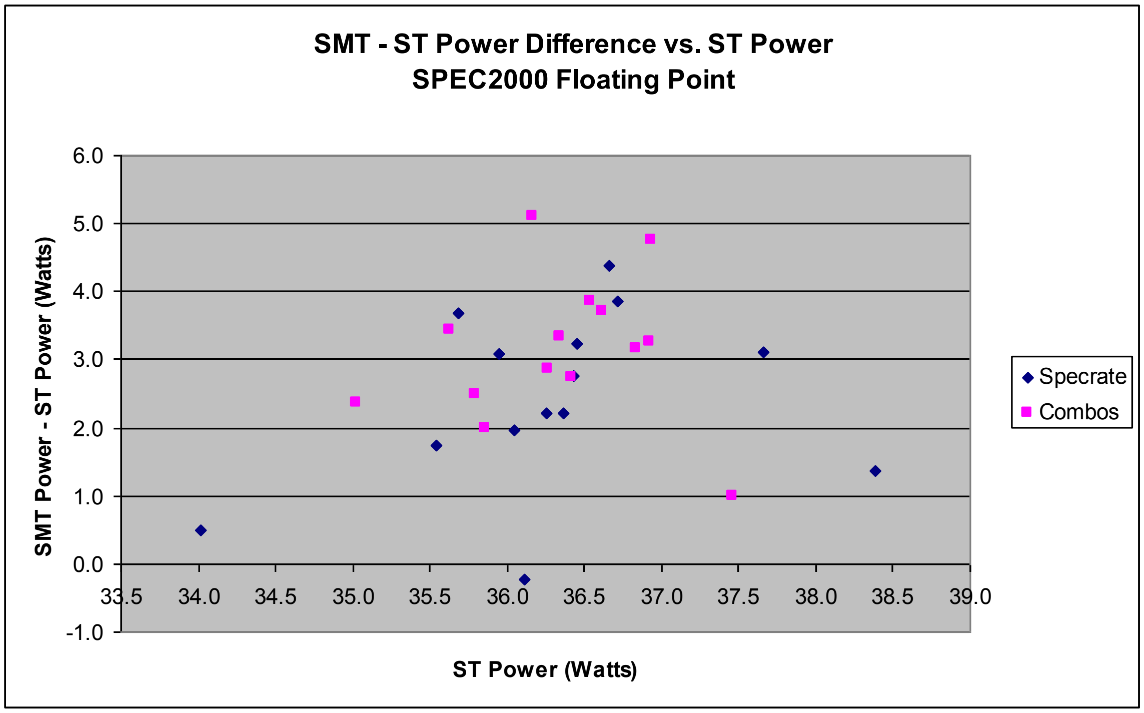

Figure 32. SMT-ST Power Difference vs. ST Power for applications in SPEC2000_fp.

Figure 32 plots the power increase from SMT vs. ST power. Like the integer applications, most points are in the 1-5 watt range. There is one outlier where the SMT energy was actually better than the ST energy. This was for the application art, when it was running on both threads.

Figure 34 plots the power increase vs. SMT speedup, and this is interesting because we see that there is a correlation between the power increase due to SMT vs. the SMT speedup. There was no such correlation for the integer applications. The reason for this in the floating point applications is because the Pentium 4 processor does not issue floating point uops on a speculative data cache miss. In other words floating point uops will wait in the scheduler until it knows that any load operations that it depends on are available (either L0 data cache hit or ready to be forwarded). This is not the case for integer operations which are scheduled speculatively assuming a L0 data cache hit. If it turns out to be a cache miss, the dependent integer uops will need to be reissued and re-executed, wasting power. Since we have less of that going on, the floating point uops see increasing SMT power with SMT speedup.

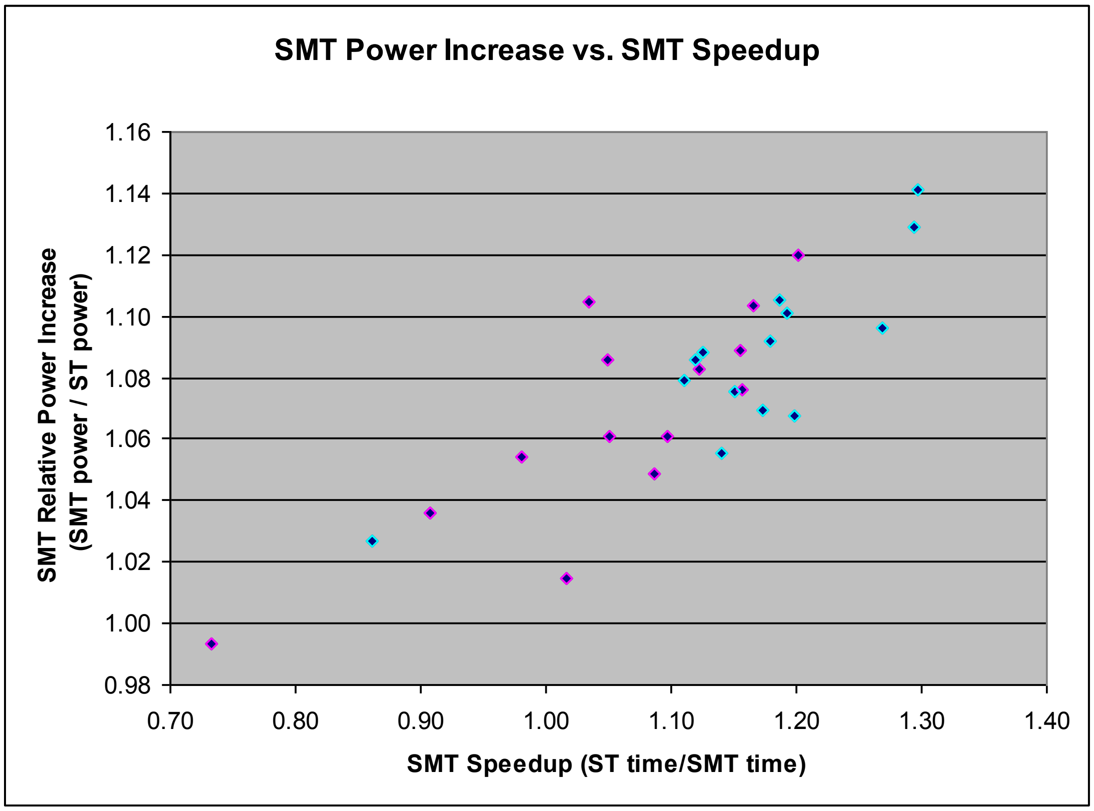

Figure 33. SMT Power Increase vs. SMT Speedup for SPECfp

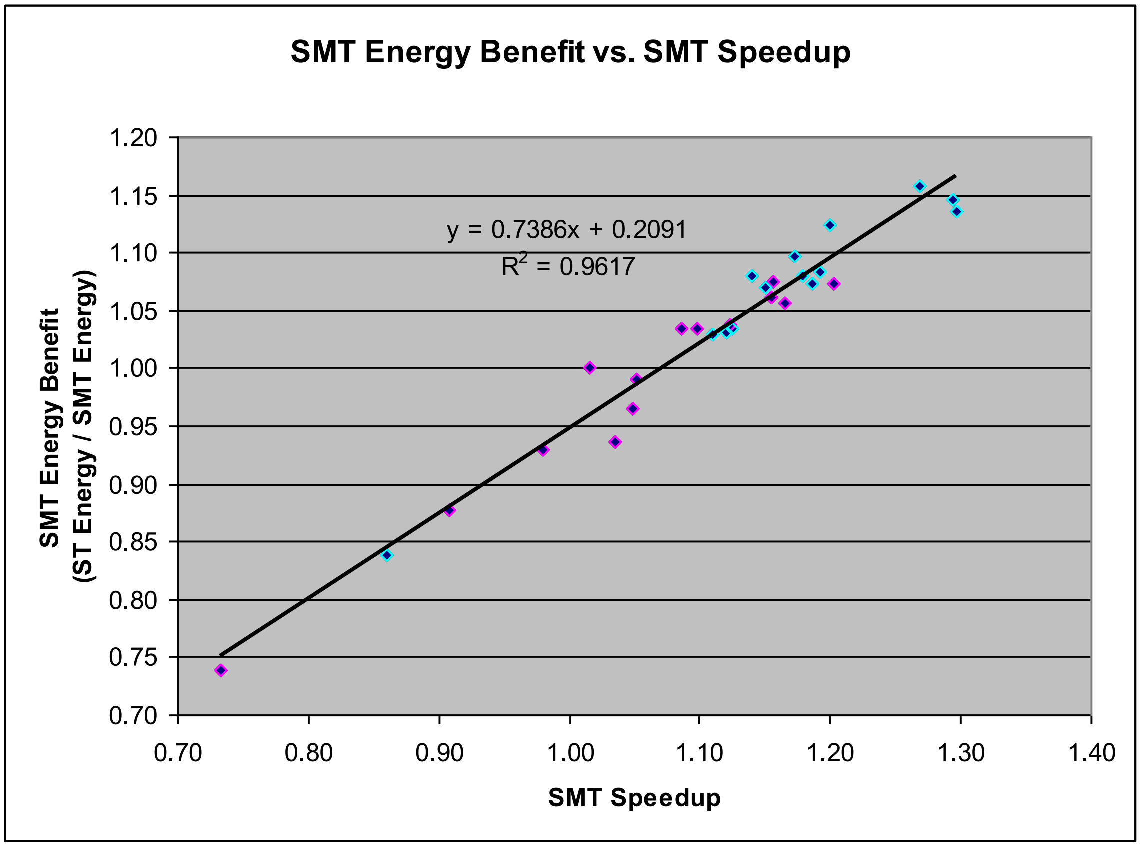

Figure 34. SMT Energy Benefit vs. SMT Speedup for applications in SPEC200_fp on an Intel Pentium 4 Processor System.

Figure 34 shows the SMT energy efficiency vs. SMT speedup, and again we see that there is a strong correlation between SMT energy efficiency and SMT speedup.

SMT energy efficiency would be the slope of the line of Figure 34. The slope for SPECfp is 0.73, somewhat lower than SPECint. As SMT speedup increases, 0.73 of that translates directly into battery life savings, and 0.27 is “wasted”. In this case, the inefficiencies include the energy spent on execution resources.

6.5 Multimedia Applications

Finally we look at a number of multimedia applications. Unlike the SPEC workloads where we arranged the workloads so that both logical processors were active simultaneously for nearly 100% of the workload, when we run individual multimedia applications, the parallelism varies due to sequential sections of code. Therefore we looked at a number of multimedia applications and selected those that were at least 50% multi-threaded. The way we determined this was to start Windows Task Manager, click the performance tab, and look for applications where the CPU utilization was $\geq$ 75%. When the CPU utilization shows 75%, it means that the application is 50% threaded because half the time one logical processor was 100% busy, and half the time the second logical processor was 50% busy, giving a total of 75% overall CPU utilization.

The following is a list of the applications selected for use in this study, including a description of the application, and the test workload description.

Autodesk* 3ds Max* 9
This is a popular animation modeling and rendering solution for film, television, games, and design visualization. It contains the essential high-productivity tools required for creating eye-catching film and television animation, cutting-edge games, and distinct design visualizations.

Test workload description: The workload used in our analysis is called Dragon_Character_Rig.max. The workload consists of a scene of a Dragon_Character_Rig.max rendered at 1920x1080. One frame is rendered. The render options set are: Atmospherics, Effects and Displacement. The advanced lighting options are also set.

Apple* iTunes* 7.0.2

iTunes is the industry leading application used to convert, save, and manage digital music. iTunes allows the user to record, organize, and play songs on a PC or use the high quality encoder to convert music to MP3 or other formats.

Test workload description: The tested workload for iTunes 7.0.2 measures the time needed to convert a 634.746KB .wav file to an mp3 with 160kbps bitrate.

MainConcept* H.264 Encoder v2.1

The H.264 Encoder v2 with v2.1 codec for Microsoft* Windows* offers video encoding and decoding of the highest quality. H.264/AVC (Advanced Video Coding), also known as MPEG-4 Part 10, is poised to be a major video standard because it can replace several popular formats while offering significant advantages over them.

Test workload description: The input file is a 24 second 1920x1080 HD (high-definition) MPEG2 video clip with a bitrate of about 18000kbps. The output is an H.264 format video clip encoded at 6000kbps.

POV-Ray* 3.7

POV-Ray, aka the Persistence of Vision Ray-Tracer*, is a free high-quality ray tracer tool for creating stunning three-dimensional graphics. Many scenes are included with POV-Ray, which can be modified so you do not have to start from scratch. In addition to pre-defined scenes, a large library of pre-defined shapes and materials is provided.

Test workload description: The POV-Ray 3.7 includes a built-in benchmark test developed by the creators of POV-Ray for evaluating system performance.

Apple* Quicktime* Pro 7.1.3

Quicktime Pro allows users to create video using the H.264 video codec, record audio for producing podcasts, create movies for iPod, convert media to more than a dozen formats, and playback a wide variety of video formats.

Test workload description: The input file is a 2 minute 1 second 416MB DV file with 720x480 resolution and 29.97 fps (frames per second). The output movie file is created using the Quicktime Broadband-High profile with H.264 compression, 672kbps video bitrate, multi-pass encoding, 480x360 resolution, AAC audio, and a 128kbps audio bitrate.

Microsoft* Windows Media* Encoder 9.0

Windows Media Encoder 9.0 with Advanced Profile is a powerful tool for content producers. It features high-quality multi-channel sound, high-definition video quality, support for mixed-mode voice and music content, advanced capture abilities, power server integration for live broadcasts, and optimized compression for a wide range of delivery scenarios including multiple bit rate streaming and delivery on CD or DVD.

Test workload description: The workload is the creation of a streaming video file for Windows Media Servers from a raw DV video file. Windows Media Encoder 9 encodes a 416MB DV file with 720x480 resolution to a streaming WMV9 file. The input DV file is a 2 minutes and 1 second video of kite-surfing. The encode rate is 282kbps, 320x240 resolution, and 29.97fps (frames per second).

XMPEG* 5.03 with DivX* 6.4

XMPEG is a multipurpose video encoding application which takes MPEG-1 and MPEG-2 streams, or DVD-IFO video format, and converts them to AVI or bbMPEG Encoder format, changing video parameters, frame rate, and audio frequency. One of the most popular uses of XMPEG is to convert unencrypted DVD VOB files to either MPEG-1 (compatible with most Panasonic/LSX Encoders) or to an AVI file (compatible with most codecs).

DivX* is a format for digital video, much like MP3 is a format for digital music. The DivX codec is based on the MPEG-4 compression standard and can reduce an MPEG-2 video (same format used for DVD) to ten percent of its original size. The DivX technology provides excellent compression and the resulting visual quality is virtually indistinguishable from a DVD.

Test workload description: The input file is a 24 second 1920x1080 HD mpeg2 video clip with a bitrate of about 18000kbps. The output is a HD (high-definition) DivX format video clip encoded at 7800kbps.

Table 12 shows the results of the SMP Power tests that we ran. The applications ranged in thread parallelism from 56% to 100%. Thread parallelism is the % of time when both logical processors are active.

Figure 35, Figure 36, and Figure 37 show the same data in graph form. The data shows the same trends as SPECint and SPECfp. In SMT-mode, the power is 2.5-4.5 watts higher than in ST-mode. Figure 37 is the important figure. It shows the SMT energy benefit. We see that the slope of the line is about 0.79 for multimedia applications. This means that as we get increasingly better SMT speedup, 0.79 of that is also transferred to energy savings, and 0.21 is wasted on the additional energy required for SMT execution. Again, the energy efficiency is compelling.

Table 12. Multimedia Applications SMT power measurements.

Application Name	Thread Parallelism	SMT speedup (ST time / SMT time)	ST Time (sec)	SMT Time (sec)	ST Power (Watts)	SMT Power (Watts)	Power Difference (Watts)	Power Diff %	ST Energy (Watt-Hrs)	SMT Energy (Watt-Hrs)	SMT Energy Benefit (ST Energy / SMT Energy)
3DS Max	98%	0.98	243	248	36.6	40.3	3.7	10%	2.47	2.77	0.89
iTunes	76%	0.88	269	307	33.4	34.5	1.1	3%	2.50	2.94	0.85
MainConcept	90%	1.23	285	231	36.1	40.2	4.0	11%	2.86	2.58	1.11
Povray	100%	1.31	651	498	37.9	42.2	4.4	12%	6.84	5.84	1.17
Quicktime	56%	1.14	802	705	37.2	39.8	2.6	7%	8.29	7.80	1.06
Media Encoder	70%	1.18	150	127	37.6	41.0	3.4	9%	1.56	1.44	1.08
XMPEG DivX	70%	1.14	271	238	36.6	39.7	3.2	9%	2.75	2.63	1.05

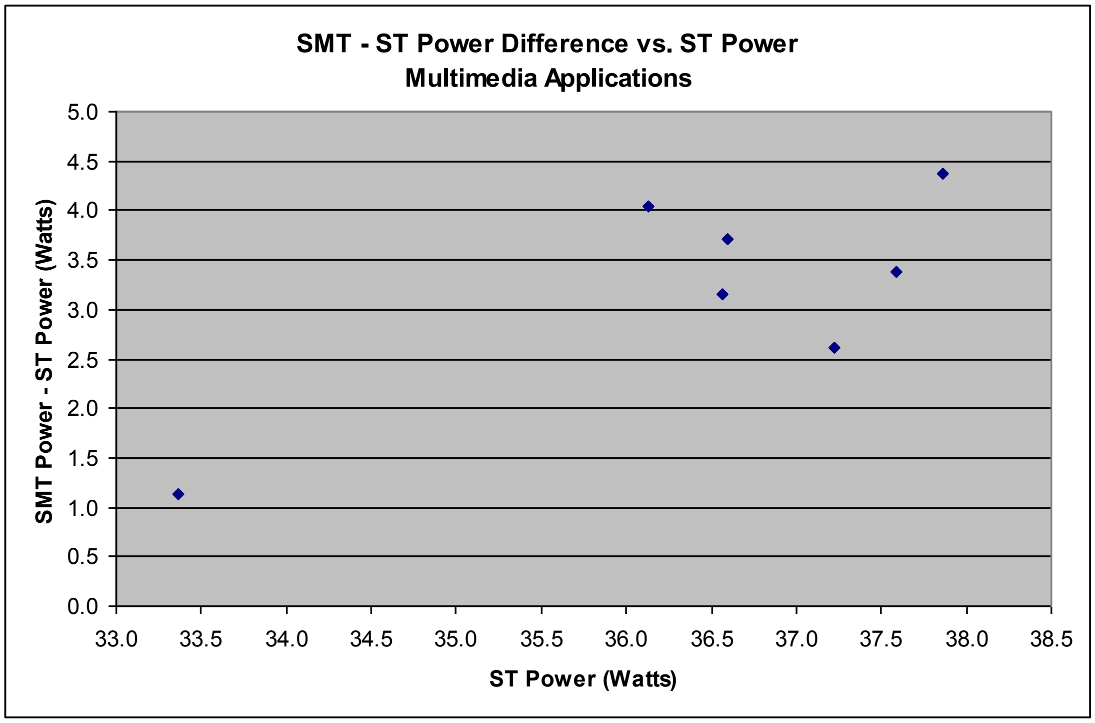

Figure 35. Multimedia applications SMT-ST Power Difference vs. ST Power

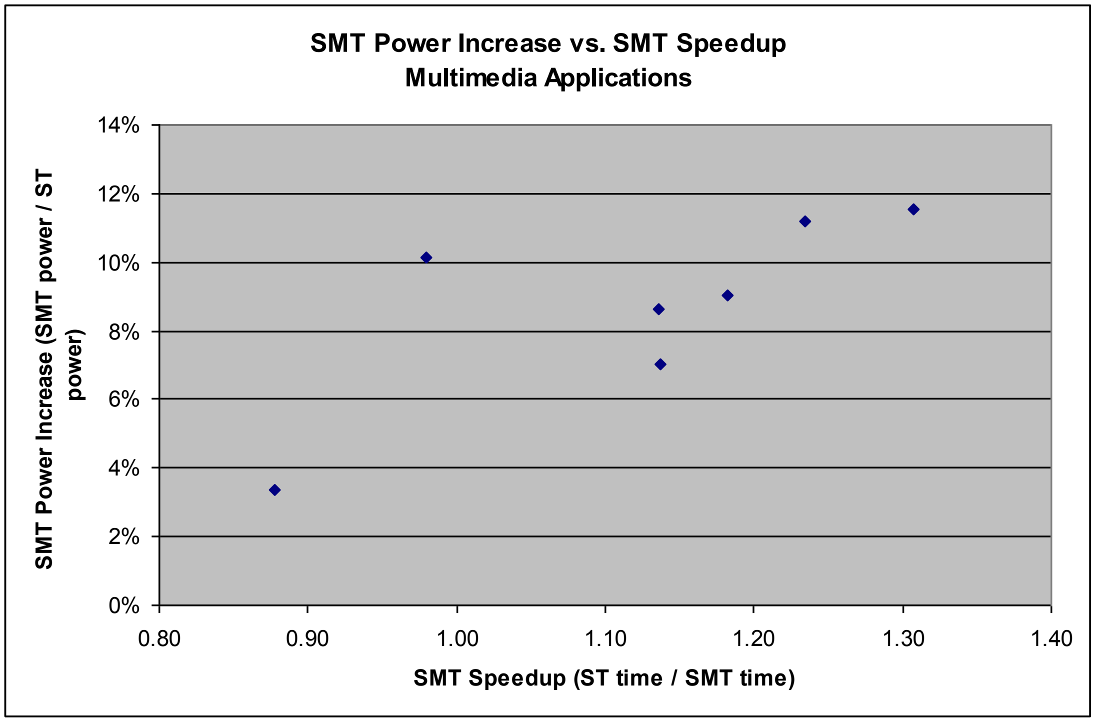

Figure 36. SMT Power Increase vs. SMT Speedup for Multimedia Applications

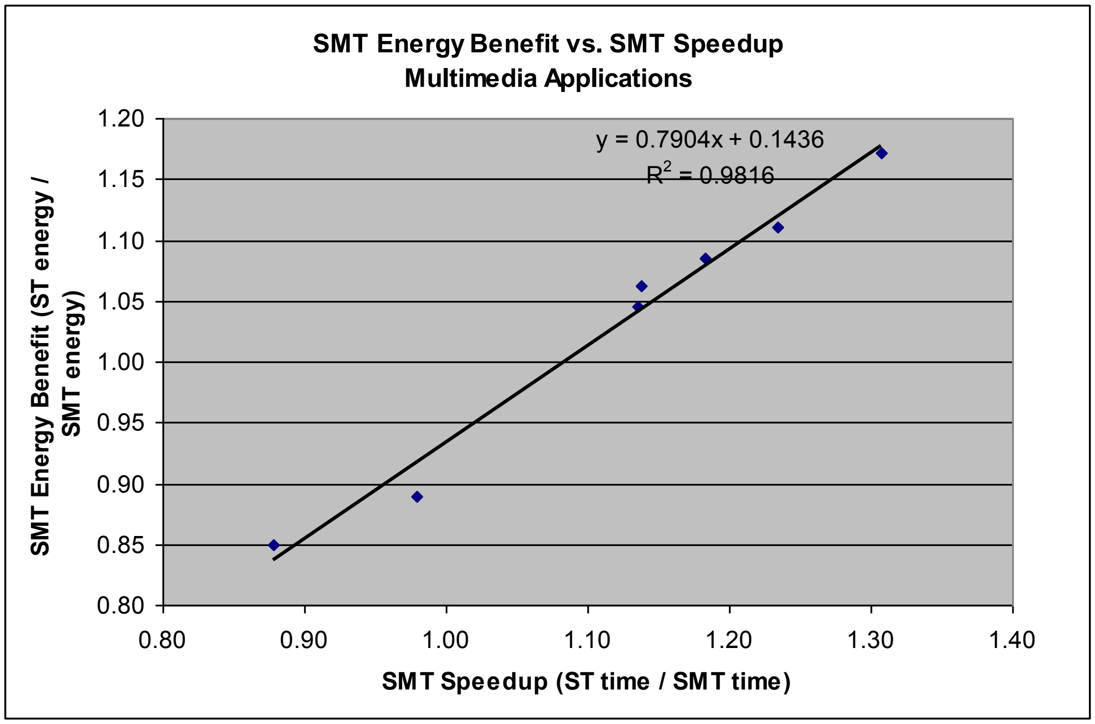

Figure 37. SMT Energy Benefit vs. SMT Speedup for multimedia applications.

6.6 Analytic Model of SMT Energy Benefit

Thus far in this chapter, we have achieved a good understanding of the SMT energy benefit of the Intel Pentium 4 Processor on a number of workloads and microbenchmarks. The next step would be to offer a model to predict the expected SMT energy benefit of other applications.

To do this, let us take all of the results of the previous sections and plot all of the points on the same graph. Figure 38 and Figure 39 plot all the points without and with the microbenchmarks respectively. Note that the slopes of the linear fit in the two figures are 0.79 and 0.86 which are not radically different. This gives us a reassuring feeling that we should be able to reasonably predict SMT energy benefit of other applications.

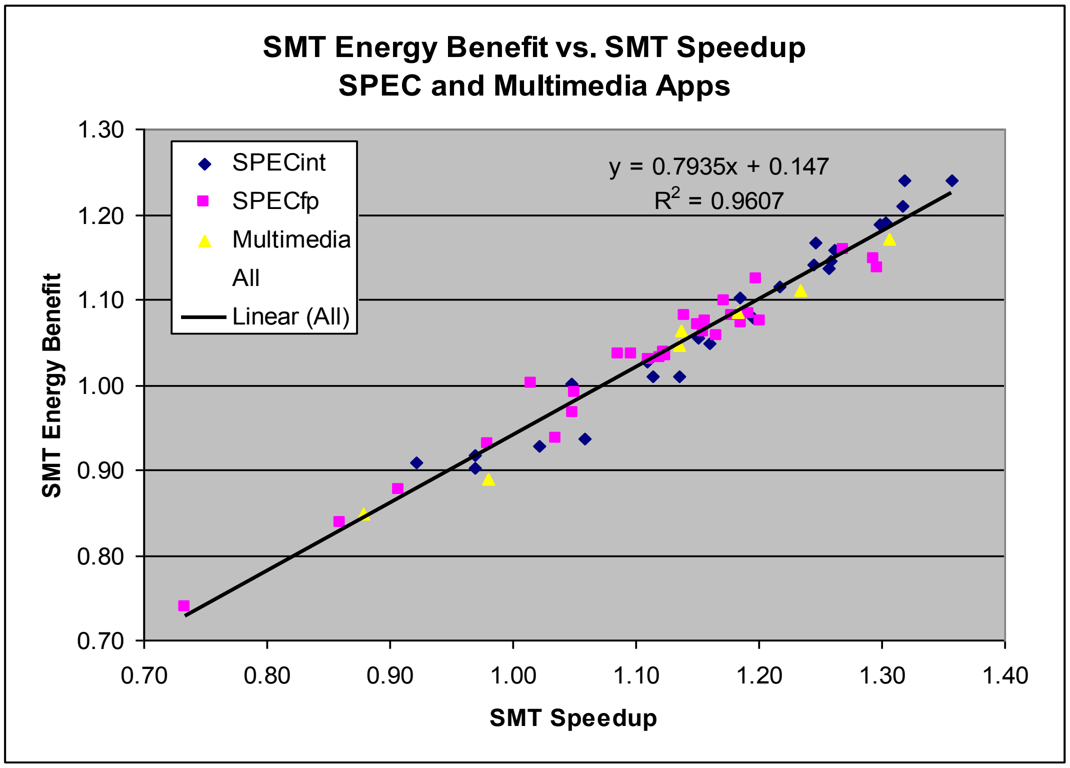

Figure 38. SMT energy benefit for all workloads measured, except the micros.

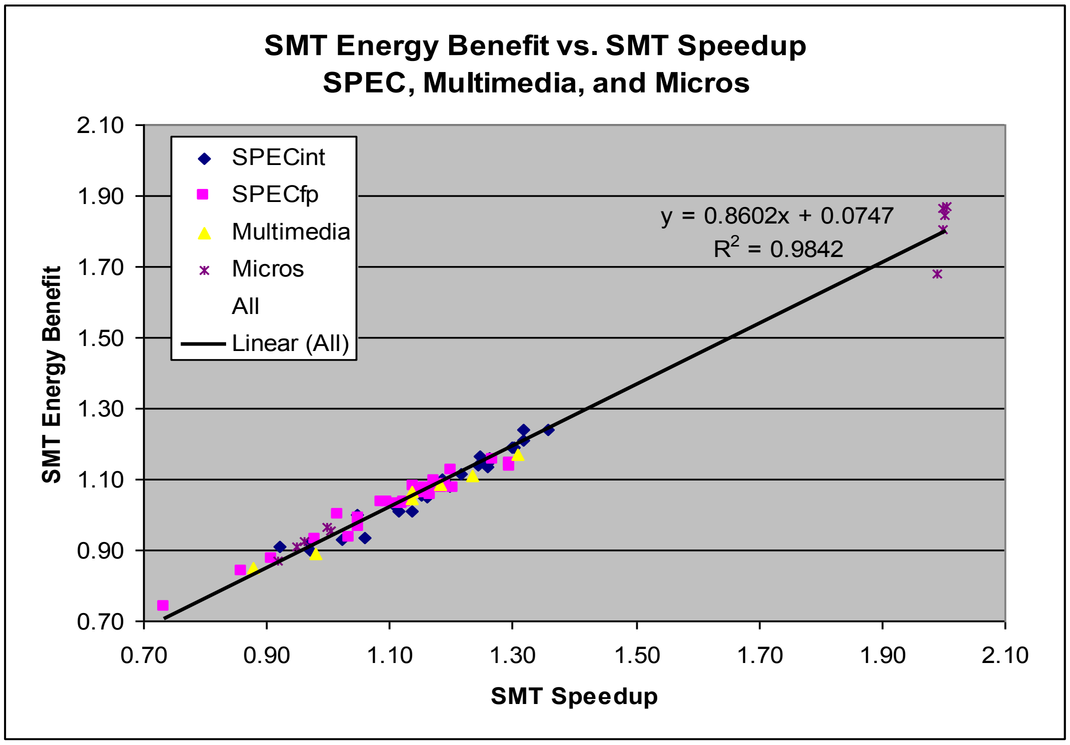

Figure 39. SMT energy benefit for all workloads measured, including micros.

Let us discuss briefly the accuracy of the linear regression to the data set (how close the data points are to the best-fit line). The R^2 value is a measure of the goodness-of-fit of the linear regression. The value R^2 is a fraction between 0.0 and 1.0, and has no units. An R^2 value of 0.0 means that knowing X does not help you predict Y; there is no linear relationship between X and Y. When R^2 equals 1.0, all points lie exactly on a straight line with no scatter. Knowing X lets you predict Y perfectly. The closer the R^2 value is to 1, the more precise the linear regression, or best-fit line is.

Figure 38 and Figure 39 show R^2 values of 0.96 and 0.98, respectively, meaning that the linear regression line is a very good fit. It is interesting that the micros and the real applications all fall on a fairly nice linear line! This means that as we get better SMT speedup, we get better battery life. SMT on Pentium 4 is indeed an extremely power-efficient feature.

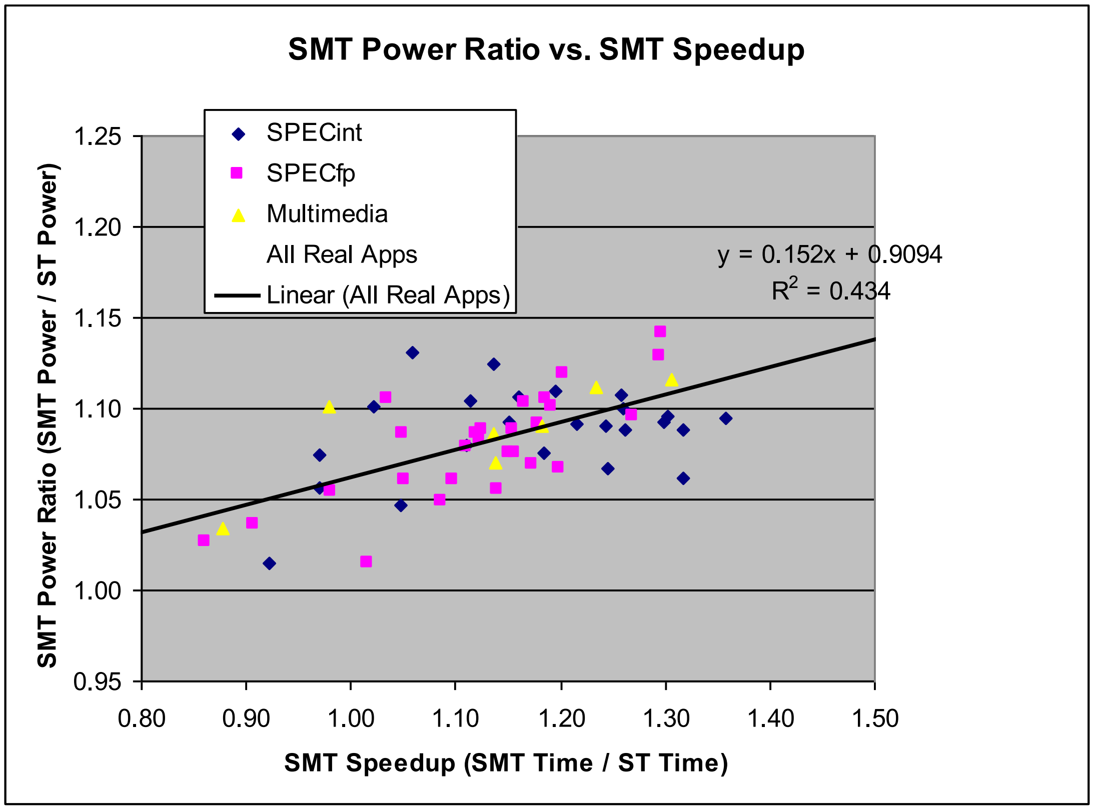

Figure 40. SMT Power Ratio vs. SMT Speedup

Now let’s take a more detailed look, and plot the SMT Power Ratio, which is the SMT Power / ST Power, as shown in Figure 40.

Conceptually, we would expect the following equation to hold:

SMT Power Ratio = 1+ SMT Fixed Overhead + (SMT Speedup-1) * % Dynamic

Where the SMT Fixed Overhead is defined by how much additional power is needed simply to have an SMT-enabled part, e.g., the area and power overhead for tracking a full set of logical registers. The % Dynamic is defined as the additional dynamic execution overhead due to executing in SMT mode because we are executing more instructions through the pipeline and increasing throughput.

The % Dynamic is approximately due to offsetting factors such as:

- We have effectively smaller caches and TLBs due to two software threads simultaneously sharing the resources, resulting in additional cache misses, and other activity.
- Our speculative execution is only half as deep, and therefore we have fewer instructions on the mispredicted path taking (and wasting) execution resources.

Doing a curve fit of the data in Figure 40, and solving for SMT Fixed Overhead and % Dynamic gives us:

SMT Fixed Overhead = 6%

% Dynamic = 15%

The key question here is what is the significance of these numbers? An extremely efficient base microarchitecture would have a high % Dynamic power component. 15% is quite low. This means that the Pentium 4 processor actually is not a power-efficient base microarchitecture. Some of the reasons include data speculation and replay, the special ½ cycle ALUs which use power-hungry circuits to attain incredible speeds, and the double-speed clocking at execution and the first-level cache.

We believe that with the emphasis on power-efficiency, future microarchitectures will be implementing more power-efficient circuits, more clock gating, more power-efficient microarchitecture features. However, at the same time process technology is reported to have more leakage in the future. Therefore there is a wide range of possible SMT Energy Benefit vs. SMT Speedup possibilities, depending on the % Dynamic power of the processors of the future.

Let us now project the SMT Energy Benefit for a variety of future processor scenarios. The SMT Energy Benefit can be written as follows:

SMT Energy Benefit = SMT Speedup / (1+ SMT_fixed_overhead + ((SMT_Speedup – 1) * % Dynamic))

Plotting this for various values of % Dynamic, we get the results in Figure 41. As the % Dynamic increases, as would be expected for future power-efficient microarchitectures, we would expect that the SMT Energy Benefit to be reduced. SMT is most power-efficient on high-powered processors by amortizing that high power fixed cost over more performance. But on more efficient processors there is less overhead to amortize and power becomes more and more proportional to performance gain.

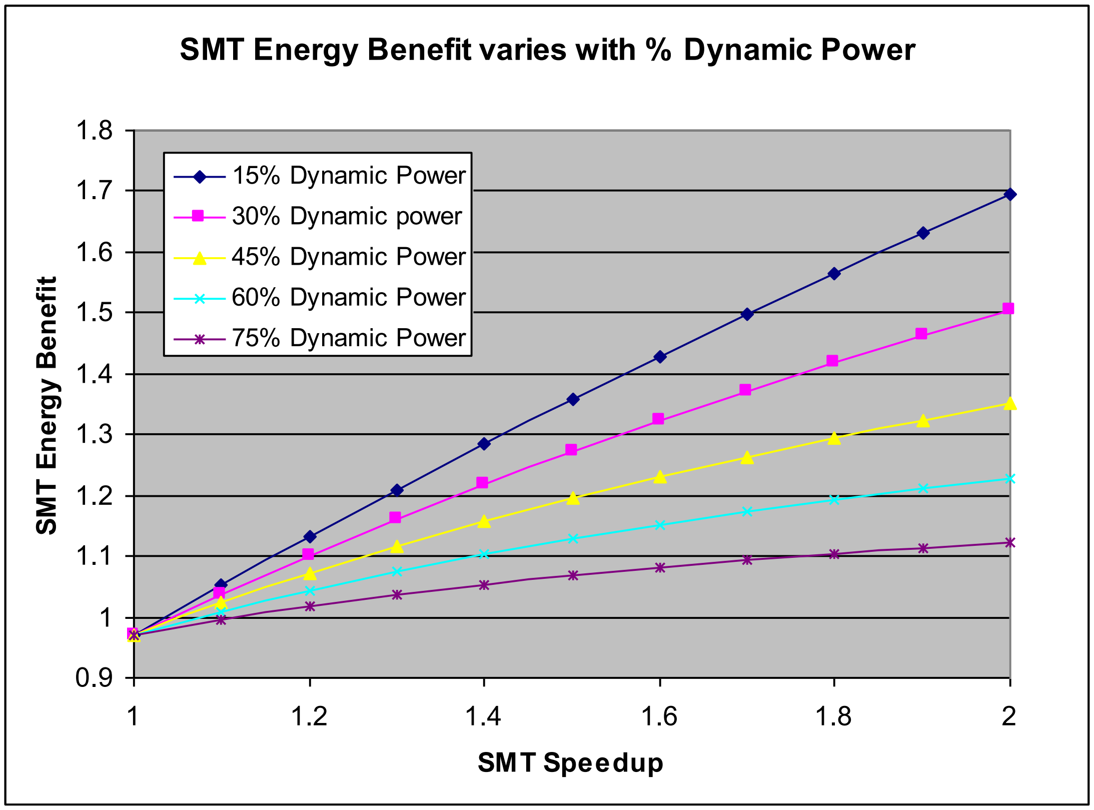

Figure 41. Projections of SMT Energy Benefit of Future Processors, for a variety of assumptions for % Dynamic power.

6.7 Related Work

There is a significant body of work comparing SMT and CMP energy efficiency, and there is also some work comparing SMT with single-thread superscalar. While clearly CMP gives higher overall performance and throughput, SMT usually has better power efficiency except when applications are compute-bound, or have bad cache thrashing behavior.

Li et al. [91] used a power simulator, PowerTimer, to analyze the power and energy efficiency of SMT. They used ten SPEC2000 integer benchmarks to make 16 pairs of workloads. Ten of the workloads were the same single thread benchmark paired with itself. Then they selected six other pairings (gzip+perlbmk, gcc+gap, twolf+mcf, parser+bzip2, bzip2+twolf, gcc+mcf), and used the average performance of these 16 pairs. They concluded that on a POWER4-like microarchitecture, designers can expect SMT performance gains of nearly 20% with an increase in power of 24% resulting in a significant improvement in energy efficiency as measured by the Energy-Delay^2 metric. Their simulations show an improvement in ED2 of nearly 30%, assuming an energy leakage of 10%. The SMT energy efficiency gets even better for higher leakage because as leakage power is fixed and SMT's increase in resource utilization and therefore active power becomes a smaller proportion of the total power.

Seng and Tullsen [92] used SMTSIM, which models an out-of-order SMT microarchitecture, and paired it with an high-level power model [93] based on silicon area, transistor density, and activity level. As the authors note, this model is only a rough estimate power of the processor. It is primarily intended to provide intuition for the power impact of a variety of optimizations to a fixed-resource SMT processor.

Cong et al. [94] use a power estimation tool similar to the wattch [95] approach to show how clustered microarchitectures might impact the SMT vs. CMP power and performance decision. They show that clustering can reduce the difference in energy consumption between SMT and CMP by reducing the number of buses and ports. CMP actually comes out ahead when leakage current is low, but SMT in a clustered microarchitecture is better when leakage current is high. For these studies they used 6 integer and 6 floating point applications from the SPEC2000 suite to create 12 pairs of traces.

Li and Martinez [96] look at the power and performance of a CMP running parallel applications from scientific application domains. For their analysis they use the wattch [95] power analytical model and a performance simulator to model an Alpha 21264-like processor in a CMP configuration. They show that while a CMP can give good power performance scalability, there is a complex dependency on a variety of factors including process technology, the application's parallel efficiency, the power budget, the performance requirements, the chip's voltage/frequency scaling properties, and the number of available processor cores.

Sasanka et al. [97] looked at comparing SMT and CMP performance for multimedia workloads. They also used the wattch [95] simulator and a microarchitecture performance simulator. In contrast to other studies, however, they concluded that CMP was more energy efficient than SMT for multimedia workloads. While no other studies focused on multimedia workloads, there are at least a few issues with their studies. First, they compared the SMT and CMP configuration at the *same performance*. Since their CMP configuration had much higher area because each core had double the area and therefore double the resources, to get the CMP configuration to have low enough performance they had to drastically scale back the frequency which dramatically decreases power. Other studies compared CMP and SMT for the same work, and some for the same die size, which I believe is a more reasonable comparison. Second, they assumed leakage of only *2%*, which is far too low for any modern high-performance processor silicon technology. Third, they assumed that 90% of the circuitry is not only clock gated, but also that the clock gating turned off power to those circuits such that they used absolutely *no power* when not active. With these assumptions it is understandable how they came to their conclusions, but the assumptions are unreasonable. While multimedia workloads may still prefer CMP over SMT, I don't believe the results would be so dramatic, and quite possibly several or more of the workloads would favor SMT.

Y. Li et al. [98] also studied the performance, energy, and thermal properties of SMT and CMP in the context of a fixed die size. They show that CMP offer significantly more throughput than SMT. As far as energy efficiency CMP could also be superior for CPU-bound benchmarks, while SMT was better on a variety of workloads but especially so on some memory-bound benchmarks due to larger L2 cache size. SMT will offer better power efficiency as leakage increases in future process technologies. Their studies were done using a power model called PowerTimer [99] together with a microarchitecture performance simulator, and used 15 SPEC2000 benchmarks, pairing each with itself, and forming 18 other pairs.

Kaxiras et al. [100] studies the problem of energy efficiency of SMT and CMP for a VLIW embedded processor for a mobile phone. They show that for mobile workloads using VLIW cores, SMT is more efficient. They used their own simulators and analytic power models.

Isci et al. [101] measured power on an Intel Pentium™ 4 Processor and developed a model to correlate performance counter and power measurements. However, even though SMT was available on the Pentium 4 processor, they did not enable it. Therefore all of their measurements and analysis were done on a single-threaded CPU. Due to the complexity of the Pentium 4 Processor, the models that Isci et. al. developed were often off by 10-20% on average in either direction. Sometimes the estimated power was too high and sometimes too low. We speculate that this inaccuracy led them to use their model to identify power phases rather than to estimate power.

Contreras et al. followed the Isci methodology and did something similar for the Intel Xscale® processor [102, 103]. They measured the power of the processor and used performance counters to develop a power model. Because the Intel Xscale® processor has a much simpler microarchitecture, it lends itself much better to modeling with performance counter values. They were able to get the average estimation error down to 4% on average across tested benchmarks. But again, this work was for a single-threaded CPU.

Bellosa [104] proposed using performance counters to analyze software thread power requirements of an energy-aware OS scheduler. He showed that the use of performance events can give a good correlation to energy use. The processor used was the Intel Pentium II processor, a single-thread CPU.

Li et al. [105] estimate run-time power of the operating system using simple IPC counts. They did not run on real systems, but used simulators to model a simple in-order MIPS CPU and showed that for such a CPU a single metric, instructions per cycle (IPC), gives a fairly accurate estimate of CPU power of the operating system's software threads.

Joseph et al. [106], Kadayif et al. [107], and Weissel et al. [108] used performance events to estimate the contribution of different microarchitecture features to the total processor power, again on single-thread CPUs. Joseph's studies were done using a simulation of an Alpha 21264. Kadayif's studies were done by using cache performance counters on an UltraSPARC CPU to measure cache hit and misses, reads and writes. Then used generic analytic equations [109] that assume some amount of energy for every cache or memory access. Since these equations themselves were not derived from real silicon nor were they validated against real silicon, the accuracy is suspect. Weissel used the Intel XScale 80200 and a system enhanced to be able to measure the power consumed by the CPU. Though he did not come up with equations to estimate the power of the CPU, he did find that four event counter values seem to correlate to increased power. These were the instructions executed per cycle, branches executed per cycle, data cache references per cycle, and memory requests per cycle.

Bircher et al. [110] [111], Contreras et al. [102], and Lee et al. [112] used runtime information as input into detailed analytic models to estimate power consumption. Again, these models were for single-threaded processors.

Gurun et al. [113] explored an adaptive feedback-driven power estimation model. Such feedback-driven models will be increasingly useful in large systems and database warehouses, where power dissipation and thermal limitation are often the limiting factor on the capacity of the system.[114] [115].

There is a variety of developing simulation frameworks that are commonly used in literature to simulate or estimate the power of a variety of processors and systems. Again, none of these were for a processor with simultaneous multi-threading. Some of the commonly used simulators include Wattch [95] developed by Brooks, Tiwarri, and Martonosi at Princeton University. Wattch is a processor simulator that consists of a suite of parameterizable power models. The user can specify the building blocks such as array structures, fully associative content-addressable memory, logic and interconnect, and the clock tree. Based on usage counts, the simulator estimates power consumption. SimplePower [116] is another simulator that does usage counts and estimates power consumption. SimplePower was developed by Vijaykrishnan et al. at Pennsylvania State University; it uses a combination of analytic models and switch capacitance energy tables to model each part of the microarchitecture. PowerTimer [99], developed by Brooks et al. at IBM, uses a parameterized set of energy functions that can be used in conjunction with any given cycle-accurate microarchitectural simulator. The energy models are for typical structures such as latches, buffers, multiplexers, register files, cache arrays, etc. SoftWatt [117], developed by Gurumurthi et al., uses analytical energy models for the entire system including the CPU, memory hierarchy, and disk subsystem.

6.8 Summary and Future Work

In this chapter, we have shown that SMT energy efficiency is very good when SMT speedup is good. We have also concluded that an analytic model to estimate SMT energy benefit is a very reasonable thing to do, and that a minimum SMT speedup of 1.1 would most likely be needed for any SMT microarchitecture before a SMT energy benefit would be observed. However, after that we can assume that SMT energy benefit will improve linearly with additional SMT speedup. In the Intel Pentium 4 Processor case, nearly 0.8 of the SMT speedup goes towards SMT energy benefit, while 0.2 of that is expended on extra energy resources needed due to the additional resource utilization. However, future processors which are more efficient will have a less dramatic impact.

The SMT implementation in the Intel Pentium 4 Processor was the first, and was done in a low cost way with only minimum hardware. In future processors, such as Intel's recently disclosed Nehalem processor, the expected SMT speedup will be much higher and it will be interesting where the SMT Energy Benefits will fall.

6.8.1 Future work

There is a lot of further work that can be done in this area. Repeating these measurements on the Intel Nehalem processor, when available, would be extremely insightful because the Nehalem processor has done a lot to improve SMT performance and power efficiency.

Also testing on other desktop applications would be very insightful. The multimedia applications and the SPEC benchmarks have been very carefully optimized for the Pentium 4 Processor, so there is not as much "wasted" resources as in more typical applications. The performance and energy benefit would be more compelling on typical less-optimized applications.

More study needs to be done on pairing two or more applications; such scenarios can represent the multi-tasking environment of typical computer users. For example, virus scan, ripping a DVD, and video decode or computer game.

CHAPTER 7
CONCLUSIONS

This thesis is concerned with hardware approaches to maximizing the number of independent instructions delivered to the execution core and thereby maximizing the processing efficiency for a given amount of compute bandwidth. The compute bandwidth is determined by the number of parallel execution units or pipelining of those units in the processor. Keeping those computing elements busy is a key to maximizing processing efficiency and thereby maximizing power efficiency.

While there are some applications that have an enormous amount of independent instructions that can be issued in parallel without inefficiencies due to branch behavior, cache behavior, or instruction dependencies, these types of applications are not the common cases.

This thesis presents research on approaches to improving the number of independent instructions that are provided to the execution core. This work has two major areas of focus to provide a large quantity of readily executable instructions to the execution core. The first approach addresses the problem of very small basic blocks due to branchy code. Our approach is to predict multiple branches simultaneously and fetch non-contiguous basic blocks simultaneously to send to the backend.

If we can correctly predict two to three branch paths every cycle and if the average basic block size is five instructions, then the average fetch size is 10 to 15 instructions. Many non-numeric applications today have an average basic block size of 5 instructions, and floating point applications tend to be much larger. The ability to fetch multiple basic blocks per cycle coupled with compiler technology to increase basic block size can result in significant performance gains. This chapter shows that just providing the ability to fetch multiple instructions without specific compiler optimizations already increases the useful instruction fetch capacity of a machine by 40% and 63% when 2 and 3 basic blocks can be fetched each cycle, respectively, for integer benchmarks. For floating point benchmarks, the improvement is 27% and 59%.

The second approach to increasing the number of independent instructions to the execution core is to introduce a separate independent software thread. Specifically, we discuss an approach called simultaneous multithreading. We present the Intel Pentium 4 Processor, and study some of the microarchitecture choices and tradeoffs to make simultaneous multithreading as efficient as possible.

Finally, we look at the power efficiency of simultaneous multithreading. More independent instructions in the processor mean better processor resource utilization. Improvements in processor resource utilization also benefit energy efficiency. We found that indeed the energy efficiency is improved. We showed that although SMT power is typically ~5-15% higher than ST power, the energy efficiency can be quite substantial when the SMT speedup is > 1.1. A new metric, the SMT Energy Benefit, was defined and used to show that for a given increment of SMT speedup, approximately 80% of that directly lowers energy usage, while 20% is spent to obtain that speedup on the Intel Pentium 4 Processor. We then generalized the results and built a model for what future processors' SMT Energy Benefit might be. We concluded that SMT will continue to be an energy-efficient feature, however as processors get more energy-efficient the relative SMT Energy Benefit will be reduced.

7.1 Future Directions

There is a lot more work that needs to be done to understand and characterize the performance and energy benefits of SMT. For real system measurements, Intel's Nehalem processor will soon be coming out, and it would be fascinating to repeat many of the studies on Nehalem. Since Nehalem's microarchitecture will be quite different from the Pentium 4 Processor, it will provide a good opportunity to compare microarchitecture features on a wide variety of real applications as well as a wide variety of microbenchmarks.

Power models should be validated with real system measurements to see how close they are. Good models using performance monitoring events should be developed to open up power analysis to a wide variety of applications. These models can also allow operating systems and other applications to understand the power characteristics of their applications and possibly dynamically adjust or schedule threads for optimum power performance distribution.

Even for the Pentium 4 Processor, there is a lot more work that can be done to develop more microbenchmarks in order to isolate and understand the power characteristics of the different microarchitecture features.

BIBLIOGRAPHY

[1] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 180-192, 1987.

[2] W. Hwu, S. Mahlke, W. Chen, P. Chang, and N. Warter, "The Superblock: An Effective technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, 1993.

[3] B. R. Rau, D. Yen, W. Yen, and R. Towle, "The Cydra 5 Departmental Supercomputer - Design Philosophies, Decisions, and Trade-offs," *IEEE Computer*, pp. 12-35, 1989.

[4] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," presented at Proceedings of the 7th International Conference on Supercomputing, Tokyo, Japan, 1993.

[5] T. Yeh and Y. N. Patt, "Two-Level Adaptive Branch Prediction," *The 24th ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 51-61, 1991.

[6] T. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, pp. 124-134, 1992.

[7] T. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, 1993.

[8] T. Yeh and Y. N. Patt, "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution," *Proceedings of the 25th International Symposium on Microarchitecture (Micro)*, pp. 129-139, 1992.

[9] S. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 76-84, 1992.

[10] B. Calder and D. Grunwald, "Next Cache Line and Set Prediction," presented at Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995.

[11] T. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," presented at Proceedings of the 22nd Annual International Symposium on Computer Architecture, 1995.

[12] S. Wallace and N. Bagherzadeh, "Multiple Branch and Block Prediction," presented at Proceedings of the 1997 ACM/IEEE Conference on High Performance Computer Architecture, 1997.

[13] J. B. Lee, S. M. Moon, and W. Sung, "An Enhanced Two-level Adaptive Multiple Branch Predictor for Superscalar Processors," *Journal of Systems Architecture*, vol. 45, pp. 591-602, 1999.

[14] D. Koppelman, "The Benefit of Multiple Branch Prediction on Dynamically Scheduled Systems," presented at Workshop on Duplicating, Deconstructing, and Debunking, 2002.

[15] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block Ahead Branch Predictors," *ACM SIGOPS Operating Systems Review*, vol. 30, pp. 116-127, 1996.

[16] R. Rakvic, B. Black, and J. P. Shen, "Completion Time Multiple Branch Prediction for Enhancing Trace Cache Performance," presented at Proceedings of the 27th Annual International Symposium on Computer Architecture, 2000.

[17] D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi, "Control Flow Prediction for Dynamic ILP Processors," presented at Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO 26), Austin, Texas, 1993.

[18] G. Reinman, T. Austin, and B. Calder, "Optimizations Enabled by a Decoupled Front-End Architecture," *IEEE Transactions on Computers*, vol. 50, pp. 330-355, 2001.

[19] A. Peleg and U. Weiser, "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line," U.S. Patent 5, 533, Ed., 1992, 1994.

[20] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," presented at Proceedings of the 29th International Symposium on Microarchitecture, 1996.

[21] S. Patel, D. Friendly, and Y. Patt, "Evaluation of Design Options for the Trace Cache Fetch Mechanism," *IEEE Transactions on Computers*, vol. 48, pp. 193-204, 1999.

[22] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium(R) 4 Processor," *Intel Technology Journal*, vol. 5, pp. 1-13, 2001.

[23] S. Borkar, N. P. Jouppi, and P. Stenstrom, "Microprocessors in the Era of Terascale Integration," presented at Design, Automation, & Test in Europe Conference & Exhibition (DATE'07), 2007.

[24] S. Rusu, "Trends and Challenges in High-Performance Microprocessor Design," in *Electronic Design Processes 2004*. Monterey, CA, 2004.

[25] S. Gunther, F. Binns, D. M. Carmean, and J. C. Hall, "Managing the Impact of Increasing Microprocessor Power Consumption," *Intel Technology Journal*, vol. 5, 2001.

[26] T. Ungerer, B. Robic, and J. Silc, "A Survey of Processors with Explicit Multithreading," *ACM Computing Surveys*, vol. 35, pp. 29-63, 2003.

[27] D. T. Marr, "Introduction to Next Generation Multiprocessing: Hyper-Threading Technology," in *Intel Developer Forum, Fall 2001*, 2001.
[28] M. Nemirovsky, "DISC, A Dynamic Instruction Stream Computer," vol. Ph.D.: University of California, Santa Barbara, 1990.
[29] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirovsky, "Performance estimation of Multistreamed, Superscalar Processors," presented at HICSS94, 1994.
[30] W. Yamamoto and M. Nemirovsky, "Increasing Superscalar Performance Through Multistreaming," presented at Proceedings of the 1995 Conference on Parallel Architectures and Compilation Techniques (PACT'95), 1995.
[31] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," presented at Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95), 1995.
[32] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96)*, pp. 191-202, 1996.
[33] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," *IEEE Micro*, vol. 17, pp. 12-19, 1997.
[34] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers, "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," *ACM Transactions on Computer Systems*, vol. 15, pp. 322-354, 1997.
[35] M. Loikkanen and N. Bagherzadeh, "A Fine-Grain Multithreading Superscalar Architecture," *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, pp. 163-168, 1996.
[36] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," presented at Proceedings SPIE Real Time Signal Processing IV, 1981.
[37] J. T. Kuehn and B. J. Smith, "The Horizon Supercomputing System: Architecture and Software," presented at 1988 ACM/IEEE Conference on Supercomputing, 1988.
[38] M. R. Thistle and B. J. Smith, "A Processor Architecture for Horizon," presented at 1988 ACM/IEEE Conference on Supercomputing, 1988.
[39] L. Carter, J. Feo, and A. Snavely, "Performance and Programming Experience on the Tera MTA," presented at SIAM Conference on Parallel Processing, 1999.
[40] W. Anderson, P. Briggs, and C. S. Hellberg, "Early Experience with Scientific Programs on the Cray MTA-2," presented at 2003 ACM/IEEE Conference on Supercomputing, 2003.
[41] R. H. Halstead and T. Fujita, "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing," presented at 15th Annual International Symposium on Computer Architecture, 1988.

[42] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-Machine Multicomputer," presented at 28th International Symposium on Computer Architecture, 1995.

[43] C. Hansen, "MicroUnity's MediaProcessor Architecture," *IEEE Micro*, vol. 16, pp. 34-41, 1996.

[44] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, and C. Lichtenau, "Building the 4 Processor SB-PRAM Prototype," presented at Proceedings of the Thirteenth Hawaii International Conference on System Sciences, 1997.

[45] A. Formella, T. Grun, and C. W. Kebler, "The SB-PRAM: Concept, Design, and Construction," presented at Proceedings 3rd Conference on Massively Parallel Programming Models, 1997.

[46] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeong, G. D'Souza, and M. Parkin, "Sparcle: An Evolutionary Processor Design for Large-Scale Microprocessors," *IEEE MIcro*, vol. 13, pp. 48-61, 1993.

[47] A. Mikschl and W. Damm, "MSparc: A Multithreaded Sparc," in *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume II*, vol. 2, 1996, pp. 461-469.

[48] A. Metzner and J. Niehaus, "MSparc: Multithreading in Real-Time Architectures," *Journal of Universal Computer Science*, vol. 6, pp. 1034-1051, 2000.

[49] H. Sullivan and T. R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I," in *International Symposium on Computer Architecture Proceedings of the 4th Annual Symposium on Computer Architecture*, 1977.

[50] W. Grunewald and T. Ungerer, "Towards Extremely Fast Context Switching in a Blockmultithreaded Processor," in *Proceedings of the 22nd Euromicro Conference*. Prague, Czech Republic, 1996.

[51] W. Grunewald and T. Ungerer, "A Multithreaded Processor Designed for Distributed Shared Memory Systems," in *Proceedings of the International Conference on Advances in Parallel and Distributed Computing*. Shanghai, China, 1997.

[52] K. Luth, A. Metzner, T. Piekenkamp, and J. Risu, "The Events Approach to Rapid Prototyping for Embedded Control System," in *Proceedings of the Workshop Zielarchitekturen Eingebetteter Systeme*. Rockstock, Germany, 1997.

[53] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, "A Multithreaded Java Microcontroller for Thread-oriented Real-time Event-handling," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. Newport Beach, CA, 1999.

[54] U. Brinkschulte, C. Krakowski, R. Marston, J. Kreuzinger, and T. Ungerer, "The Komodo Project: Thread-based Event Handling Supported by a Multithreaded Java microcontroller," in *Proceedings of the 25th Euromicro Conference*. Milan, Italy, 1999.

[55] U. Brinkschulte, J. Kreuzinger, M. Pfeffer, and T. Ungerer, "A Microkernel Middleware Architecture for Distributed Embedded Real-time Systems," in

Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems. New Orleans, LA, 2000.

[56] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel, "A Multithreaded PowerPC Processor for Commerical Servers," *IBM Journal of Research and Development*, vol. 44, pp. 885-898, 2000.

[57] M. Tremblay, "A VLIW Convergent Multiprocessor System on a Chip," in *Proceedings of the Microprocessor Forum*. San Jose, CA, 1999.

[58] Intel, "Intel Internet Exchange Architecture Network Processors: Flexible, Wirespeed Processing from the Customer Premises to the Network Core.," Intel Corporation, Santa Clara, CA, White Paper 2002.

[59] IBM, "IBM Network Processor," IBM Corporation, Yorktown Heights, NY, Product overview 1999.

[60] P. N. Glaskowsky, "Network Processors Mature in 2001," in *Microprocessor Report*, February 19 ed, 2002.

[61] W. J. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler, "The Message-driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro*, vol. 12, pp. 23-39, 1992.

[62] S. E. Raasch and S. K. Reinhardt, "The Impact of Resource Partitioning on SMT Processors," *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 15-25, 2003.

[63] D. T. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. A. Miller, and M. Upton, "Hyperthreading(TM) Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, pp. 1-12, 2002.

[64] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, and M. Upton, "The Microarchitecture of the Intel(R) Pentium(R) 4 Processor on 90nm Technology," *Intel Technology Journal*, vol. 8, pp. 1-17, 2004.

[65] R. Kalla, B. Sinharoy, and J. M. Tendler, "IBM Power5 Chip: A Dual-Core Multithreaded Processor," *IEEE Micro*, vol. 24, pp. 40-47, 2004.

[66] K. Diefendorff, "Compaq Chooses SMT for Alpha," *Microprocessor Report*, pp. 1, 6-11, 1999.

[67] J. Emer, "Simultaneous Multithreading: Multiplying Alpha Performance," *Microprocessor Forum*, 1999.

[68] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini, M. K. Gowan, P. Gronowski, D. B. Jackson, S. Mehta, S. V. Morton, J. D. Pickholtz, M. H. Reilly, and M. J. Smith, "Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading," *Proceedings of the 2002 IEEE Solid-State Circuits Conference (ISSCC 2002)*, pp. 334-335, 2002.

[69] S. Melvin, "Clearwater Networks CNP810SP Simultaneous Multithreading (SMT) Core," http://www.zytek.com/~melvin/clearwater.html 2001.

[70] D. T. Marr, "Hyper-Threading Technology in the Netburst(TM) Microarchitecture.," in *Hot Chips 14*. Palo Alto, Ca., 2002.

[71] D. A. Koufaty and D. T. Marr, "Hyper-Threading Technology in the Netburst(TM) Microarchitecture," *IEEE Micro*, vol. 23, pp. 56-65, 2003.

[72] Intel, *IA-32 Intel Architecture Software Developer's Manual, Vol. 3: System Programming Guide*: order no. 244472, http://developer.intel.com/design/pentium4/manuals, 2001.
[73] D. Marr, "Establishing Thread Priority in a Processor or the Like." USA Patent Pending (filed 1/22/2000): Intel Corp., 2000.
[74] D. T. Marr and D. Rodgers, "Method and Apparatus for Pausing Execution in a Processor or the Like." U.S. Patent 6,671,795 (filed 1/21/2000, issued 12/30/2003): Intel Corporation, 2003.
[75] D. L. Hill, D. T. Bachand, C. B. Prudvi, and D. Marr, "Dynamic Priority External Transaction System." USA Patent 6,654,837 (filed 12/28/1999 issued 11/25/2003) Patent 7,143,242 (filed 9/23/2003 issued 11/28/2006): Intel Corp., 2003.
[76] S. J. Jourdan, P. H. Hammarlund, H. Hum, and D. Marr, "System and method for employing a global bit for page sharing in a linear-addressed cache." USA Patent 6,675,282 (issued 2/12/2003 issued 1/6/2004) USA Patent 6,560,690 (filed 12/29/2000, issued 5/6/2003): Intel Corp., 2003.
[77] D. Marr, "Causality-based memory ordering in a multiprocessing environment." US Patent 6,681,320 (filed 12/29/1999 issued 1/20/2004): Intel Corp., 2004.
[78] L. Hacking and D. Marr, " Synchronization of load operations using load fence instruction in pre-serialization/post-serialization mode." USA Patent 6,862,679 (filed 2/14/2001, issued 3/1/2005), 2005.
[79] D. L. Hill, D. T. Marr, D. Rodgers, S. Kaushik, J. B. Crossland, and D. A. Koufaty, "Coherency techniques for suspending execution of a thread until a specified memory access occurs." USA Patent 7,127,561 (filed 12/31/2001, issued 10/24/2006): Intel Corp., 2006.
[80] L. Hacking and D. Marr, " Method and apparatus for synchronizing load operations." USA Patent 7,284,118 (filed 2/12/2004, issued 10/16/2007): Intel Corp., 2007.
[81] L. Hacking and D. Marr, "Globally observing load operations prior to fence instruction and post-serialization modes." USA Patent 7,249,245 (filed 2/12/2004, issued 7/24/2007): Intel Corp., 2007.
[82] D. Rodgers, D. Marr, D. L. Hill, S. Kaushik, J. B. Crossland, and D. A. Koufaty, " Method and apparatus for suspending execution of a thread until a specified memory access occurs." USA Patent 7,363,474 (filed 12/31/2001, issued 4/22/2008): Intel Corp., 2008.
[83] S. Swanson, L. McDowell, M. Swift, S. Eggers, and H. Levy, "An Evaluation of Speculative Instruction Execution on Simultaneous Multithreaded Processors," *Transactions on Computer Systems*, vol. 21, 2003.
[84] J. Brayton, "Nehalem: Talk of the Tock, Intel Next Generation Microprocessor," presented at Intel Developer Forum, Shanghai, China, 2008.
[85] R. Singhal, "Inside Intel Next Generation Nehalem Microarchitecture," presented at Intel Developer Forum, Shanghai, China, 2008.
[86] T. N. f. A. T. a. T. A. (NATTA), *Energy: A Beginner's Guide*: The Open University, 2000.

[87] Y. Liu, R. P. Dick, L. Shang, and H. Yang, "Accurate Temperature-Dependent Integrated Circuit Leakage Power Estimation Is Easy," presented at Proceedings of the Conference on Design, Automation, and Test in Europe, Nice, France, 2007.

[88] J. Hunt. Portland, Oregon, 2007, pp. Conversation with Debbie Marr on the modifications to the Lakeport/ICH7 motherboard required to measure CPU power.

[89] E. Hall, "On a New Action of the Magnet on Electric Currents," *American Journal of Mathematics*, vol. 2, 1879.

[90] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, Third edition*: Morgan Kaufmann, 2002.

[91] Y. Li, D. Brooks, Z. Hu, K. Skadron, and P. Bose, "Understanding the Energy Efficiency of Simultaneous Multithreading," in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. Newport Beach, CA: ACM, 2004, pp. 44-49.

[92] J. Seng, D. Tullsen, and G. Cai, "Power-sensitive Multithreaded Architecture," in *Proceedings of the 2000 International Conference on Computer Design*, 2000.

[93] G. Cai and C. H. Lim, "Architectural Level Power/Performance Optimization and Dynamic Power Estimation," in *Proceedings of Cool Chips Tutoria at 32nd International Symposium on Microarchitecture*, 1999.

[94] J. Cong, A. Jagannathan, G. Reinman, and Y. Tamir, "Understanding the Energy Efficiency of SMT and CMP with Multiclustering," in *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*. San Diego, CA, 2005.

[95] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[96] J. Li and J. F. Martinez, "Power-Performance Considerations of Parallel Computing on Chip Multiprocessors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, pp. 397-422, 2005.

[97] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The Energy Efficiency of CMP vs. SMT for Multimedia Workloads," in *Proceedings of the 18th Annual International Conference on Supercomputing*, 2004.

[98] Y. Li, D. Brooks, Z. Hu, and K. Skadron, "Performance, Energy, and Thermal Considerations for SMT and CMP Architectures," in *Proceedings of the Eleventh IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2005.

[99] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield, "New Methodology for Early-Stage, Microarchitecture-Level Power-Performance Analysis of Microprocessors," *IBM Journal of Research and Development*, vol. 47, pp. 653-670, 2003.

[100] S. Kaxiras, G. Narlikar, A. D. Berenbaum, and Z. Hu, "Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads," in *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[101] C. Isci and M. Martonosi, "Runtime Power Monitoring using High-End Processors: Methodology and Empirical Data," *36th ACM/IEEE International Symposium on Microarchitecture (MICRO-36)*, 2003.

[102] G. Contreras and M. Martonosi, "Power Estimation for Intel XScale Processors Using Performance Monitoring Unit Events," in *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED 05)*, 2005.

[103] G. Contreras, M. Martonosi, J. Peng, G.-Y. Lueh, and R. Ju, "The XTREM Power and Performance Simulator for the Intel XScale Core: Design and Experiences," *ACM Transactions on Computing Systems*, vol. 6, 2007.

[104] F. Bellosa, "The Benefits of Event-Driven Energy Accounting in Power-sensitive Systems," *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop*, 2002.

[105] T. Li and L. K. John, "Run-time Modeling and Estimation of Operating System Power Consumption," *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2003.

[106] R. Joseph and M. Martonosi, "Run-time Power Estimation in High Performance Microprocessors," *Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISLPED'01)*, 2001.

[107] I. Kadayif, T. Chinoda, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam, "vEC: Virtual Energy Counters," *Workshop on Program Analysis for Software Tools and Engineering*, pp. 28-31, 2001.

[108] A. Weissel and F. Bellosa, "Process cruise control: Event-driven Clock Scaling for Dynamic Power Management," *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002)*, 2002.

[109] H. Y. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Multiple Access Caches: Energy Implications," *Proceedings of The IEEE CS Annual Workshop on VLSI*, pp. 53-58, 2000.

[110] W. Bircher, J. Law, M. Valluri, and L. K. John, "Effective Use of Performance Monitoring Counters for Run-Time Prediction of Power," University of Texas at Austin Technical Report TR-041104-01, November 2004.

[111] W. Bircher, M. Valluri, J. Law, and L. K. John, "Runtime Identification of Microprocessor Energy Saving Opportunities," *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED),* 2005.

[112] B. Lee and D. Brooks, "Accurate and Efficient Regression Modeling for Microarchitecture Performance and Power Prediction," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[113] S. Gurun and C. Krintz, "A Run-Time, Feedback-Based Energy Estimation Model for Embedded Devices," *Proceedings of the International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, 2006.

[114] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, "Full-System Power Analysis and Modeling for Server Environments," *Proceedings of the Workshop on Modeling Benchmarking and Simulation (MOBS)*, 2006.

[115] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini, "Mercury and Freon: Temperature Emulation and Management in Server Systems," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[116] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower," in *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[117] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John, "Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, 2002.