RESOURCE BANKING: AN ENERGY-EFFICIENT, RUN-TIME ADAPTIVE PROCESSOR DESIGN TECHNIQUE

by

JACOB STAPLES
B.S. Midwestern State University, 2008

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2011

Major Professor: Mark Heinrich

© 2011 Jacob Staples

ABSTRACT

From the earliest and simplest scalar computation engines to modern superscalar out-of-order processors, the evolution of computational machinery during the past century has largely been driven by a single goal: performance. In today's world of cheap, billion-plus transistor count processors and with an exploding market in mobile computing, a design landscape has emerged where energy efficiency, arguably more than any other single metric, determines the viability of a processor for a given application.

The historical emphasis on performance has left modern processors bloated and over provisioned for everyday tasks in the hope that during computationally intensive periods some performance improvement will be observed. This work explores an energy-efficient processor design technique that ensures even a highly over provisioned out-of-order processor has only as many of its computational resources active as it requires for efficient computation at any given time. Specifically, this paper examines the feasibility of a dynamically banked register file and reorder buffer with variable banking policies that enable unused rename registers or reorder buffer entries to be voltage gated (turned off) during execution to save power. The impact of bank placement, turn-off and turn-on policies as well as rail stabilization latencies for this approach are explored for high-performance desktop and server designs as well as low-power mobile processors.

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES x

CHAPTER 1. INTRODUCTION 1

1.1. Motivation 1

1.2. Contributions 2

1.3. Organization 3

CHAPTER 2. BACKGROUND 4

2.1. The Register File 4

2.2. The Re-Order Buffer 7

2.3. Relevant Related Work 8

2.4. Potential For Energy Reduction 11

CHAPTER 3. DYNAMIC RESOURCE BANKING 17

3.1. The Banked ROB 20

3.2. The Banked RF 22

3.3. Bank Latencies 24

3.4. Bank Deactivation Policies 25

3.4.1 Simple Deactivation ... 25
3.4.2 Threshold Deactivation ... 25
3.5. Bank Activation Policies ... 27
3.5.1 Simple Activation ... 27
3.5.2 Threshold Activation ... 27
3.6. Insertion (Physical Register Allocation) Policies ... 28
3.6.1 First Free Entry ... 29
3.6.2 First Free Entry in Used Bank ... 29
3.6.3 First Free Entry in Most Used Bank ... 30
3.6.4 First Free Entry in Least Used Bank ... 32
3.6.5 Temporal Insertion ... 32
3.7. Implementation ... 34
3.8. Banking Power and Area Overhead ... 37
CHAPTER 4. EXPERIMENTAL RESULTS ... 40
4.1. Simulation Methodology ... 40
4.2. Baseline Measurements ... 42
4.3. Bank Policy Impact ... 54
4.3.1 Bank Activation Policies ... 54
4.3.2 Bank Deactivation Policies ... 57
4.3.3 Bank Insertion Policies ... 59
4.4. Impact of Bank Latencies ... 65
4.5. Impact of Bank Manager Energy Consumption ... 67
4.6. Testing a realistic low-power banked processor ... 68
4.7. Testing a hypothetical high-power banked processor ... 72

4.8. Future Work 75

CHAPTER 5. CONCLUSIONS 76

REFERENCES 77

LIST OF FIGURES

Figure 1 Register file read logic abstraction ... 5

Figure 2 Register file write logic abstraction ... 6

Figure 3 Dynamic resource utilization of the crafty benchmark, input I ... 12

Figure 4 Dynamic resource utilization of the crafty benchmark, input II ... 13

Figure 5 Dynamic resource utilization of the bzip2 benchmark ... 14

Figure 6 Crafty 2-issue energy savings using ideal banking ... 15

Figure 7 Crafty 8-issue energy savings using ideal banking ... 16

Figure 8 Various hypothetical resource allocation policies ... 18

Figure 9 Banking power distribution abstraction ... 20

Figure 10 ROB banking can introduce stalls if the tail points to a bank which is not in the TURNED_ON state ... 21

Figure 11 Banked RF in operation ... 23

Figure 12 RAT operation ... 23

Figure 13 RF mappings before and after allocation of a new register entry ... 35

Figure 14 Bank FSM ... 37

Figure 15 Area overhead approximation for a 2-issue machine ... 39

Figure 16 Crafty 2-issue ideal case energy reduction ... 44

Figure 17 Crafty 3-issue ideal case energy reduction ... 44

Figure 18 Crafty 4-issue ideal case energy reduction ... 45

Figure 19 Crafty 6-issue ideal case energy reduction ... 45

Figure 20 Crafty 8-issue ideal case energy reduction 46

Figure 21 H264 4-issue ideal case energy reduction 47

Figure 22 MP3 4-issue ideal case energy reduction 47

Figure 23 Bzip2 4-issue ideal case energy reduction 48

Figure 24 Namd 4-issue ideal case energy reduction 48

Figure 25 ROB and RF size impact on speedup for a 2-issue machine (Crafty) 50

Figure 26 ROB and RF size impact on speedup for an 8-issue machine (Crafty) 50

Figure 27 Contribution of RF banking to energy savings 51

Figure 28 Contribution of ROB banking to energy savings 52

Figure 29 Fraction of total processor power consumed in the (unbanked) RF and ROB for various benchmarks 53

Figure 30 Energy deltas at various activation policies compared to non-banked ROB/RF 55

Figure 31 Energy-delay product deltas at various activation policies compared to non-banked ROB/RF 56

Figure 32 Energy, delay-squared product deltas at various activation policies compared to non-banked ROB/RF 57

Figure 33 Deactivation policy energy and execution time impact 58

Figure 34 Bank insertion probability density functions for various policies (small bank latency) 60

Figure 35 Bank insertion probability density functions for various policies (large bank latency) 61

Figure 36 Bank insertion probability density functions for various policies (very large bank turn-on delay) 62

Figure 37 Delta energy by benchmark for various insertion policies 63
Figure 38 Insertion policy impact on execution time 63
Figure 39 Impact of insertion policy on energy efficiency (ET^2) with very large bank turn-on delay (512 cycles) 64
Figure 40 Impact of the D_ON (and D_OFF) latencies on banking performance in terms of E, ET, and ET^2 for the h264 benchmark normalized to the banked, zero-delay case (h264) 66
Figure 41 Impact of the D_ON (and D_OFF) latencies on banking performance in terms of E, ET, and ET^2 for the crafty benchmark normalized to the banked zero-delay case (crafty) 67
Figure 42 bank energy manager power consumption impact 68
Figure 43 Banking performance cost (execution time) 70
Figure 44 Banking performance with a realistic configuration accounting for bank latencies and bank manager power overhead 71
Figure 45 Impact of banking on aggregate processor E, ET, and ET^2 metrics 72
Figure 46 Banking impact on execution speed, energy, energy-delay product and energy delay-squared product for an aggressive processor design 74

LIST OF TABLES

Table 1 Basic processor core configuration 40

Table 2 Memory structure configuration 40

Table 3 Bank structure configurations 41

Table 4 Benchmarks 42

Table 5 Ideal banking configuration 42

Table 6 Processor aggressiveness design points 43

Table 7 Optimistic banking configuration 54

Table 8 Realistic banking configuration 58

Table 9 Bank latency test configuration 65

Table 10 Realistic banking configuration 69

Table 11 Aggressive core configuration 73

CHAPTER 1. INTRODUCTION

1.1. Motivation

A fundamental limit on Instruction Level Parallelism (ILP) in current and future processor designs is the power wall [1, 2], a theoretical limit beyond which any improvement in performance comes at the cost of thermal runaway caused by the increased power consumption required to achieve the performance gain. Historically, the power wall has not been the primary limiting factor in the evolution of processor performance. However, increases in transistor density due to improvements in lithographic and chemical processes [3] as well as the increasing number of transistors per processor as described by Moore's Law [4, 5, 6] coupled with sub-linear reductions in power as transistor sizes decrease due to leakage currents and other non ideal behaviors [5, 6] imply that as time progresses processors are actually moving closer to the power wall rather than away from it as described in [2]. The design of current and future processor generations must therefore carefully balance performance and energy efficiency if they are not to exceed the limitations imposed by the power wall.

For mobile devices, typically used in applications where achieving desktop or server levels of performance is not necessary, a more imminent concern than the power wall is finite battery capacity. Indeed, because the energy storage density of chemical batteries roughly doubles only every ten years [7], processors which scale with performance trends based on Moore's Law would inevitably consume their battery power geometrically more quickly with each successive generation if they were limited only by the power wall. To combat this,

processors intended for battery-limited applications are typically designed to achieve higher levels of energy efficiency (reduced Joules per instruction) relative to their higher performance desktop counterparts. This is achieved primarily by reducing the pipeline complexity and aggressiveness far below what is theoretically possible given current technology and research as in the ARM Cortex [8] or Intel's Atom [9].

At the high end of the performance spectrum, processor design is constrained by the power wall; at the low end, finite battery capacity. In both cases energy efficiency has emerged as a common design constraint. In other words, reducing energy consumption without adversely impacting performance would be a boon to designers of both high- and low-performance silicon. That is the primary objective of this thesis—to improve energy efficiency of a processor by reducing the energy required to execute each instruction without adversely affecting performance.

1.2. Contributions

This thesis extends existing work in register file and reorder buffer banking by examining the impact of various banking policies and design parameters on the energy efficiency of the scheme. Additionally, this work is, to the best of our knowledge, the first to combine ROB banking with register file banking. Also it is the first to propose a banked register file where some banks of registers in the register file are voltage gated during execution to conserve power. It is the first work to examine the impact of voltage gating transition latencies on such a scheme and the first to examine the impact of dynamic placement (insertion) policies.

1.3. Organization

Chapter 2 describes existing work in energy-efficient processor design and presents the mathematical underpinnings of the energy efficiency metrics used in this work as well as a brief primer on relevant processor design topics. Chapter 3 describes the proposed reorder buffer and register file banking scheme as well as policies for bank activation and deactivation and the potential impact of various voltage rail stabilization latencies. Additionally, Chapter 3 details the novel contributions of this approach and explores hardware implementation issues such as area overhead and additional power consumption. Chapter 4 describes the simulation methodology and analyzes the results obtained. Chapter 5 draws conclusions from these results and explores future avenues of research in this area.

CHAPTER 2. BACKGROUND

In this chapter we provide a brief introduction to the topics necessary to understand the proposal in this thesis. Section 2.1 provides an overview and description of the Register File (RF), both its function and design. Section 2.2 discusses the Re-Order Buffer (ROB) in a similar fashion. Section 2.3 describes the components of power consumption in a modern processor. Section 2.4 discusses and compares several popular energy efficiency metrics. Section 2.5 discusses relevant work to the energy-efficient computation field. Section 2.6 provides a preliminary best-case energy reduction using the proposed technique and serves as a springboard into the discussion of the new approach described in Chapter 3.

2.1. The Register File

The Register File (RF) is an array or tag indexed multiported fast memory structure where instruction results are stored during execution. The RF typically requires at least two read ports and a write port per instruction per cycle in a load/store architecture: one read port per operand source and one write port per destination target. In this section we present a cursory description of RF operation. For a more in-depth description of the RF see [10, 13].

The RF read logic consists of two sets (one per operand) of N*b read bit lines connected to the RF where N is the number of architectural registers and b is the number of bits per register (note that b may vary between integer and floating point registers depending upon the architecture) which feed two N:1 multiplexors, each indexed by a log2(N) bit tag. The outputs of

the read logic are two values corresponding to the two operands named in a dynamic instruction (Rx and Ry). This simplified abstraction of the RF read logic is shown below.

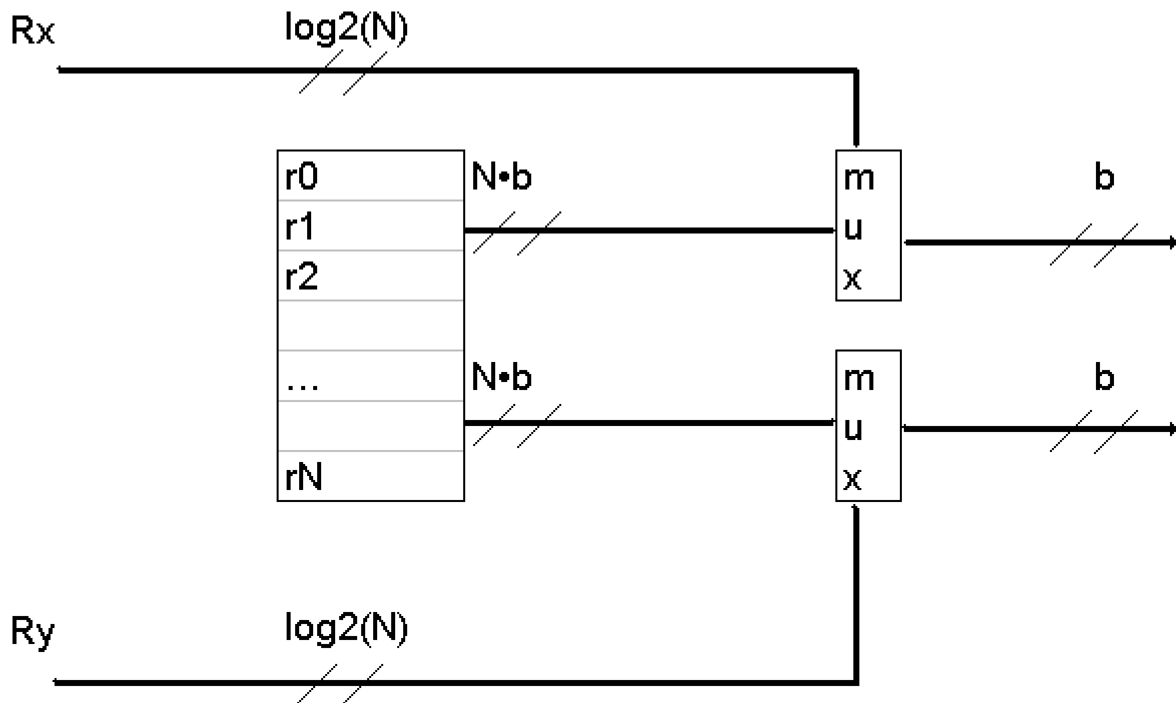


Figure 1 Register file read logic abstraction

The RF write logic consists of a target tag of log2(N) bits that is fed into a demultiplexer (whose input is simply the signal WRITE or READ) that sets the appropriate write enable line for the target register. The b-bit data line containing the write information is fed to all registers in the RF although only the register with its write enable line set to WRITE will actually be written. This simplified abstraction of the RF write logic is shown below.

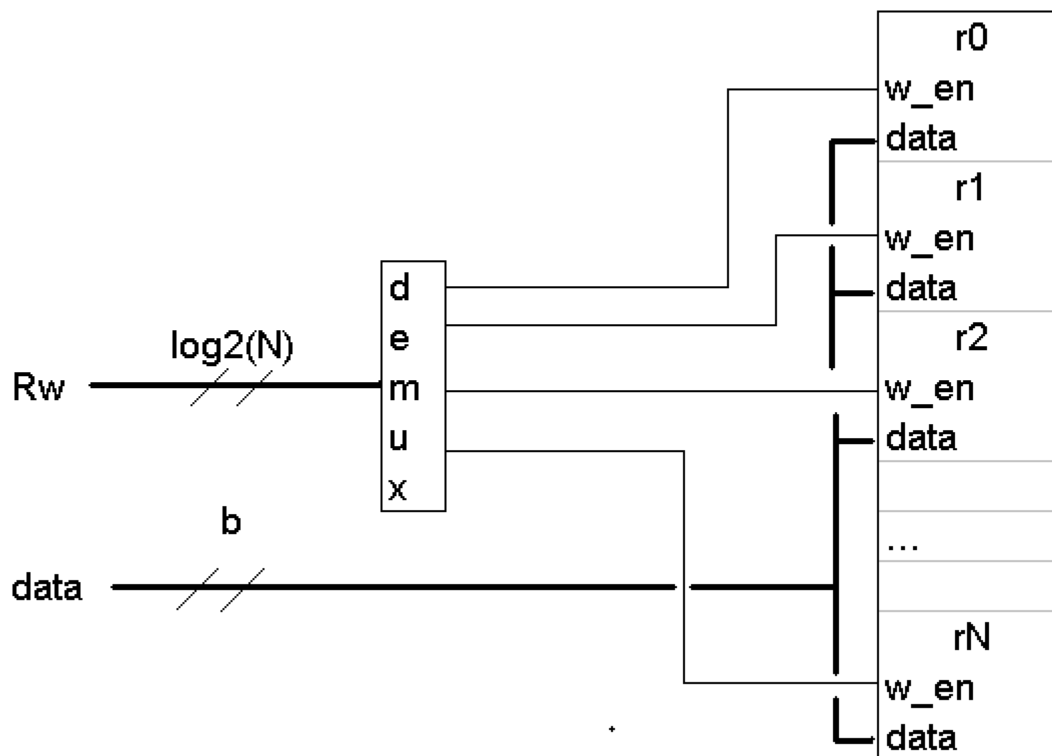


Figure 2 Register file write logic abstraction

Note that a processor capable of issuing multiple instructions per cycle effectively requires a duplicate of the RF write and read logic per instruction to be executed per cycle. For example, an 8-issue processor would require 8 write ports and 16 read ports in its register file. The read/write logic abstractions presented earlier, while simplistic, already hint at a mechanism for saving power. Because the number of ports in the RF quadratically increases the power consumption of the register file [11], the ability to power gate unused register file entries would be quite beneficial.

Modern processors commonly have more registers on-chip than their assembly language might suggest. For example, in the Alpha Instruction Set Architecture (ISA) there are 32 general-purpose integer registers and 32 floating point registers that may be referenced by an

instruction. However, the Alpha 21264, which implements the Alpha ISA, has significantly more than 32 integer registers—in fact, it has 80 [12]. These additional registers are utilized in a process called renaming where the source registers referenced by an instruction are renamed dynamically at run time by the execution logic to different architectural registers which are available for use. The operands of subsequent instructions using that source register are renamed to reflect this change for correctness. This technique can eliminate false (write after read) dependencies as long as there are free registers available for renaming [13] and allows the processor to continue without stalling even when more registers than exist in the ISA have been utilized. Renaming has been shown to be quite beneficial from a performance perspective but it is also a power-intensive technique [14] and can be wasteful if some fraction of the architectural registers are unused most of the time.

2.2. The Re-Order Buffer

After instructions are executed by the out-of-order functional units they must be committed in-order. Additionally, this in-order commitment must be done in such a fashion that an interrupt occurring on an instruction executed out of program order will not damage the machine state in terms of correctness. The traditional means of enforcing these constraints in out-of-order processors is to utilize a structure known as the reorder buffer.

The Reorder Buffer (ROB) can be viewed abstractly as a queue-like structure indexed using head and tail pointers whose entries are allocated when each instruction to be executed out of order is released for execution (dispatched) and freed when that instruction is committed in-order and its results written to the register file. The number of entries in the ROB therefore

imposes an upper limit on the total number of outstanding instructions at any given time. As described earlier, to preserve coherence and consistency the ROB retirement process occurs in-order even though some instructions will likely finish before others in the out-of-order engine (there are approaches which allow speculative commitment of ROB entries but they are not germane to this discussion). Each ROB entry contains two fields relevant to the renaming process: a value (the result of some possibly out of order computation) and a target (the register where the value in the ROB entry will be stored when the valid flag has been set for that ROB entry and all other prior instructions in the ROB have committed). Note that this register is a physical register (not an ISA or architectural register) which is mapped to using a structure called the Register Alias Table (RAT). The RAT maintains the mapping between ISA and physical registers.

While the queue abstraction is useful for understanding how the ROB operates, in actuality the ROB is designed similarly to the register file [15]. Thus, it makes sense that a technique used to save power in the register file, if sufficiently general, can also be applied the ROB.

2.3. Relevant Related Work

Clock gating [16, 17] is an energy efficiency design technique in which unused processor components are disconnected (typically using a simple AND gate with an enable line input) from the clock signal to prevent them from consuming dynamic power wastefully. Clock gated units may be quickly re-enabled by driving the enable line high in a subsequent cycle and typically have a one or zero cycle reactivation delay. Clock gating has historically been a formidable tool

in the design of an energy-efficient processor. As process sizes shrink, however, leakage current becomes an increasingly large component of power consumption [6]. Indeed, modern processors are already heavily influenced by these leakage currents, referred to as static power dissipation. It is important to note that static power dissipation will occur even in clock gated units.

Power gating, or voltage gating [18] is a technique that attacks both dynamic and static power simultaneously. Voltage gating involves disconnecting the supply rail from a unit using a power gating transistor when the unit is no longer needed. Power gated units consume virtually no static or dynamic power but incur costly shut down and warm up latencies as the voltage in the unit falls and stabilizes, respectively. In [19], transition latencies of 6-20 cycles are used. In this work we explore the impact of a wider range of latencies.

More abstract work in energy-efficient design has also been conducted. Martinez and Ipek [20] apply machine learning concepts to the problem of shared resource allocation to determine how much memory bandwidth, for example, should be given to each core in a Chip Multi Processor (CMP) during execution to maximize aggregate performance. They use a neural network with feedback links from various shared resources to implement this scheme. In their work the metric used to gauge performance is speedup rather than energy efficiency.

Additionally, a wide body of work has been conducted in the area of reducing register file and ROB complexity. Tseng and Asanovic [21] propose a banking technique for register files orthogonal to the work conducted in this thesis. Similar to this work they divide the register file into discrete units called banks and refer to their scheme as banking, but they do not voltage gate unused banks. Instead they gate individual multiplexor and demultiplexor data lines connected to each bank. This technique can reduce power dissipation in the data lines in turned-on banks and has been shown to reduce register file power by up to 40%. Additionally, turning off

multiplexor lines makes the register file access time faster by up to 25%, potentially speeding up execution. Their technique has the downside that it may introduce conflicts in the register file if the number of writes or reads to a given bank exceeds the number of turned on write or read multiplexors. It also complicates the pipeline considerably since a conflict resolution policy must be performed in subsequent cycles in the case of a conflict. In this work we do not model their structural banking approach but consider it a technique that can achieve additive power savings on top of the energy reduction we obtain.

Kucuk et al [22] proposed a distributed ROB design that can reduce ROB power by nearly 50%. They divide the ROB into discrete units that are assigned the task of reordering the output of each of the functional units. They use another level of renaming to uniquely identify in which sub bank an ROB entry has been allocated. The same group, in prior work [23], determined that only 6% of operands are satisfied using ROB read ports so they propose a scheme to eliminate these read ports entirely using small retention buffers to capture the majority of read operands. Similar to the work done by Tseng and Asanovic, Kucuk's work focuses primarily on reducing the number of ports in a multiported structure.

Ponomarev, Kucuk et al [24] propose a ROB that is dynamically resized and implements several other energy reduction mechanisms to save nearly 75% of the ROB power. They use the relationship between dispatch rate and ROB occupancy to determine when the ROB should be enlarged or contracted. Their work is intriguing, particularly given the large energy reductions it achieves, but it assumes the bank turn-on and turn-off latencies are zero, an assumption that is no longer applicable in an era where clock speeds approach 4GHz on commodity processors (and the bank latencies could be 32 or more processor cycles). Additionally, their work does not consider the effects of bank turn-on and turn-off policies.

In summary, at the most abstract level there are three categories of work relevant to this thesis: dynamic resource allocation, voltage gating, and power-efficient RF and ROB design. Each of these fields are mature and well explored but have not been combined in a single technique in the manner proposed herein. This work is an attempt to bridge the gap between these disparate areas of research by creating a RF and ROB which resize dynamically with support for power gating to maximize energy efficiency at run time.

2.4. Potential For Energy Reduction

As discussed earlier, modern processors are typically over provisioned for the work they do in the average case in the hope that during some computationally bound interval their performance will be greater than if the processor were designed such that all resources were used on average. This scenario is depicted in the figure below where ROB entry and architectural integer register demand are averaged over and plotted against various epochs (an epoch consists of 1024 cycles in this case) for the crafty benchmark from SPEC2000.

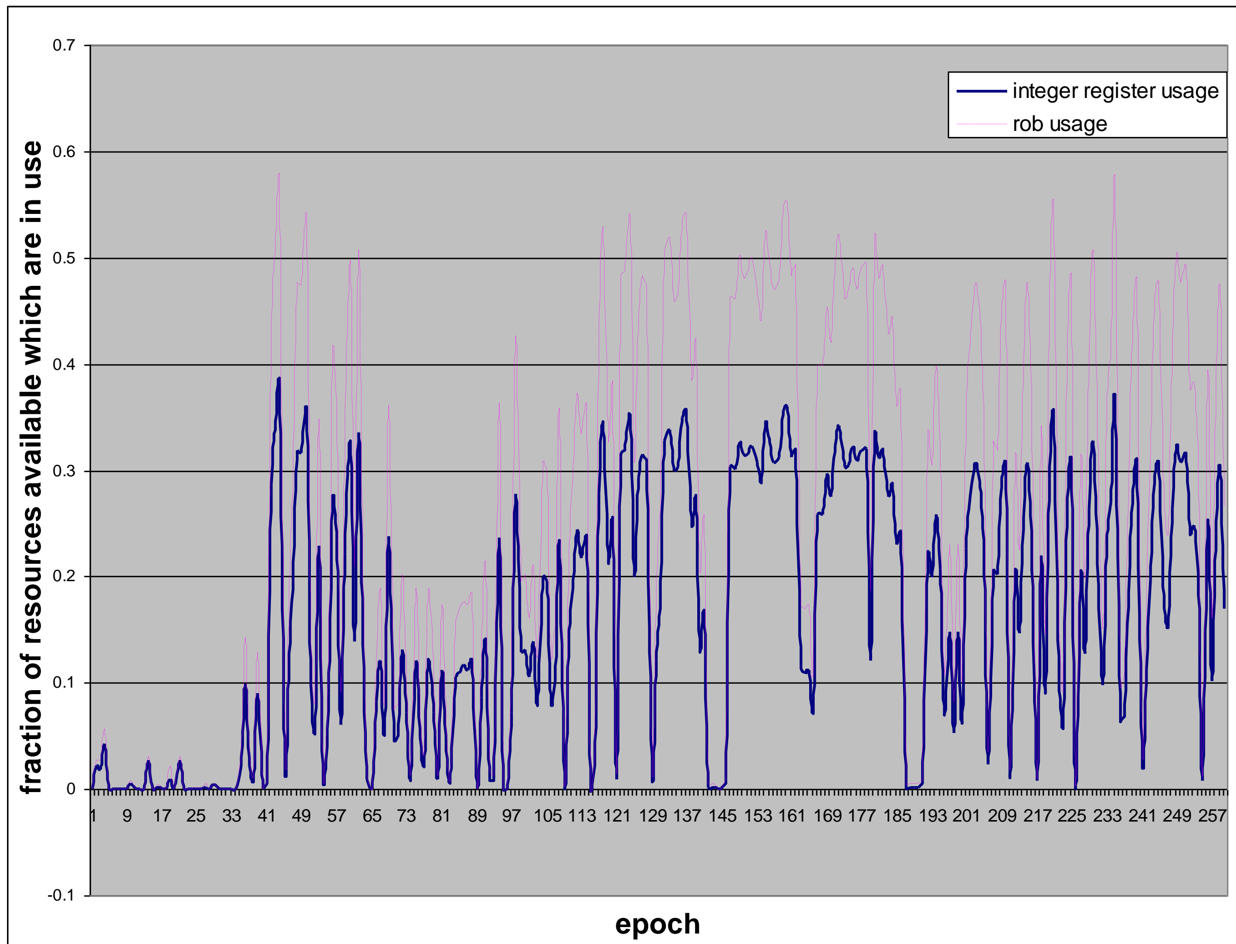


Figure 3 Dynamic resource utilization of the crafty benchmark, input I

We see that the number of integer registers and ROB entries in use vary widely during program execution. This utilization pattern can also change depending upon the input used. Figure 4 below shows the resource demand for the same benchmark with a different input to demonstrate that even a profiling run may not be able to adequately determine the resource needs of a complex program in execution. Additionally, the resource utilization depends upon the nature of the binary itself as demonstrated in Figure 5 below for the bzip2 benchmark from SPEC2006. Given that a fixed-size RF or ROB structure capable of allocating the maximum demanded number of RF or ROB entries would for the vast majority of time be wasteful, containing many entries that consume power but are not utilized, and the difficulty and

impracticality of profiling binaries whose resource utilization patterns may vary depending upon their inputs, it was determined that the only feasible way to attack this problem was dynamically at run time. In other words, the processor itself determines how many resources it should have powered on at any given instant.

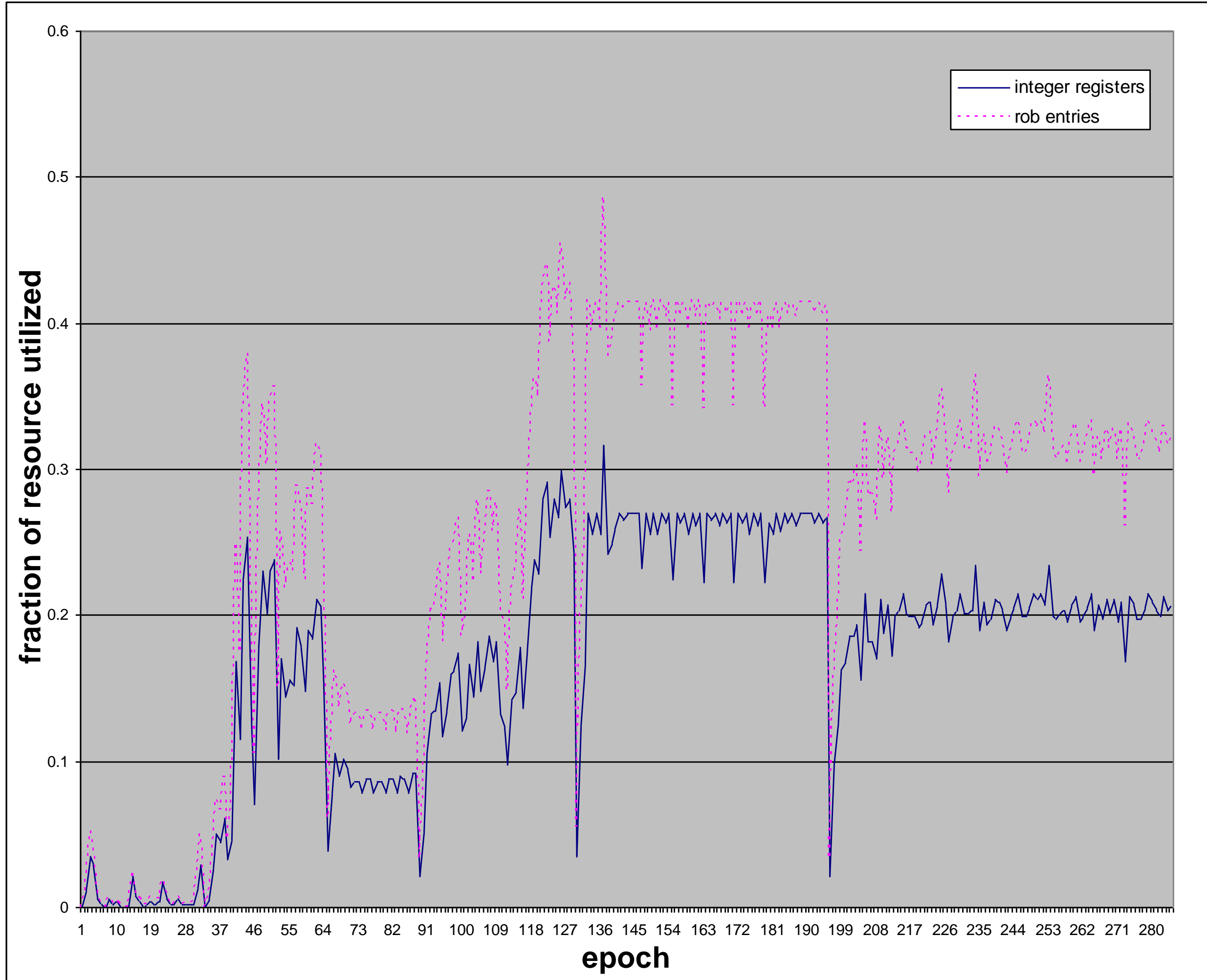


Figure 4 Dynamic resource utilization of the crafty benchmark, input II

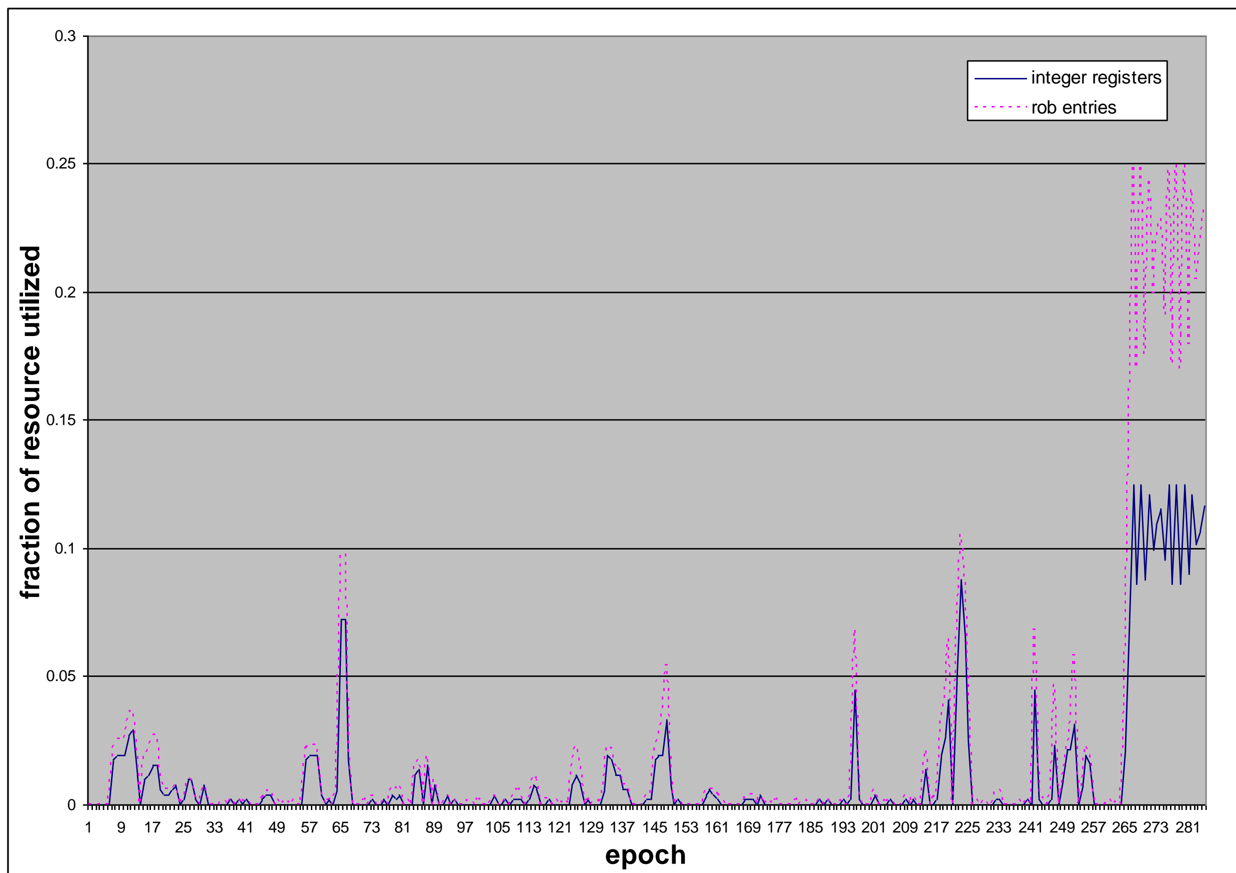
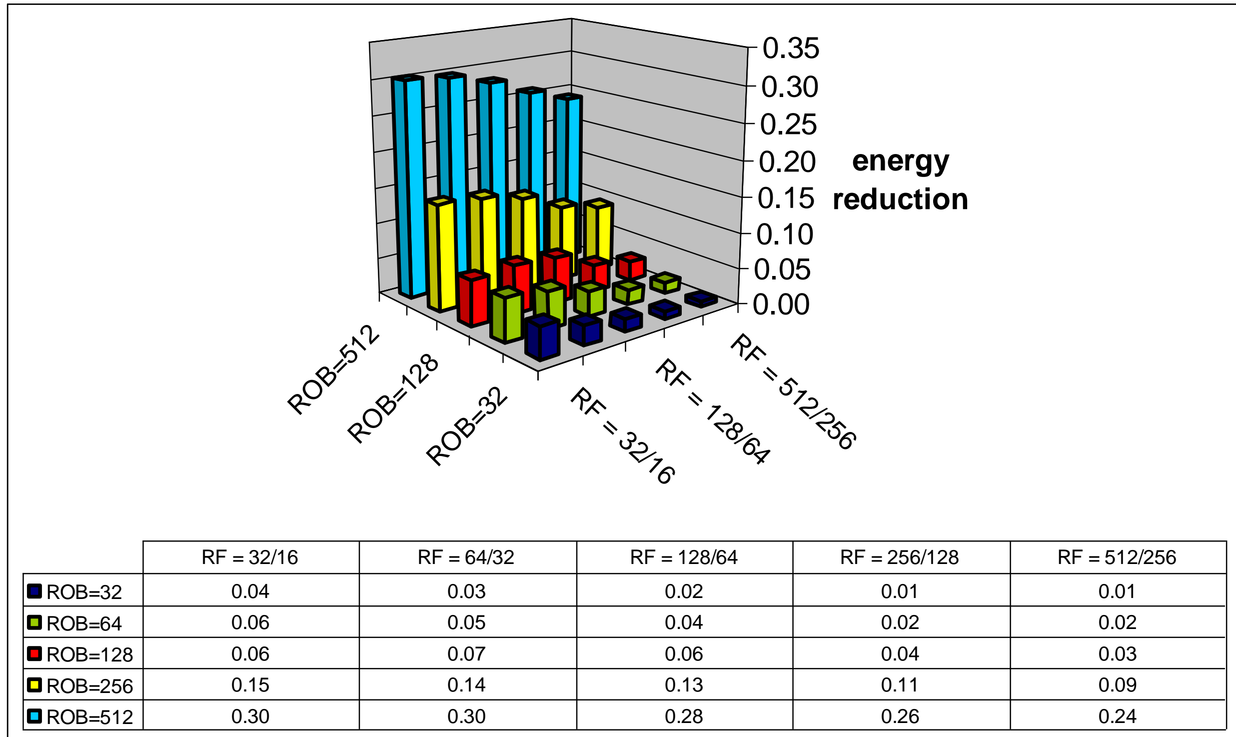


Figure 5 Dynamic resource utilization of the bzip2 benchmark

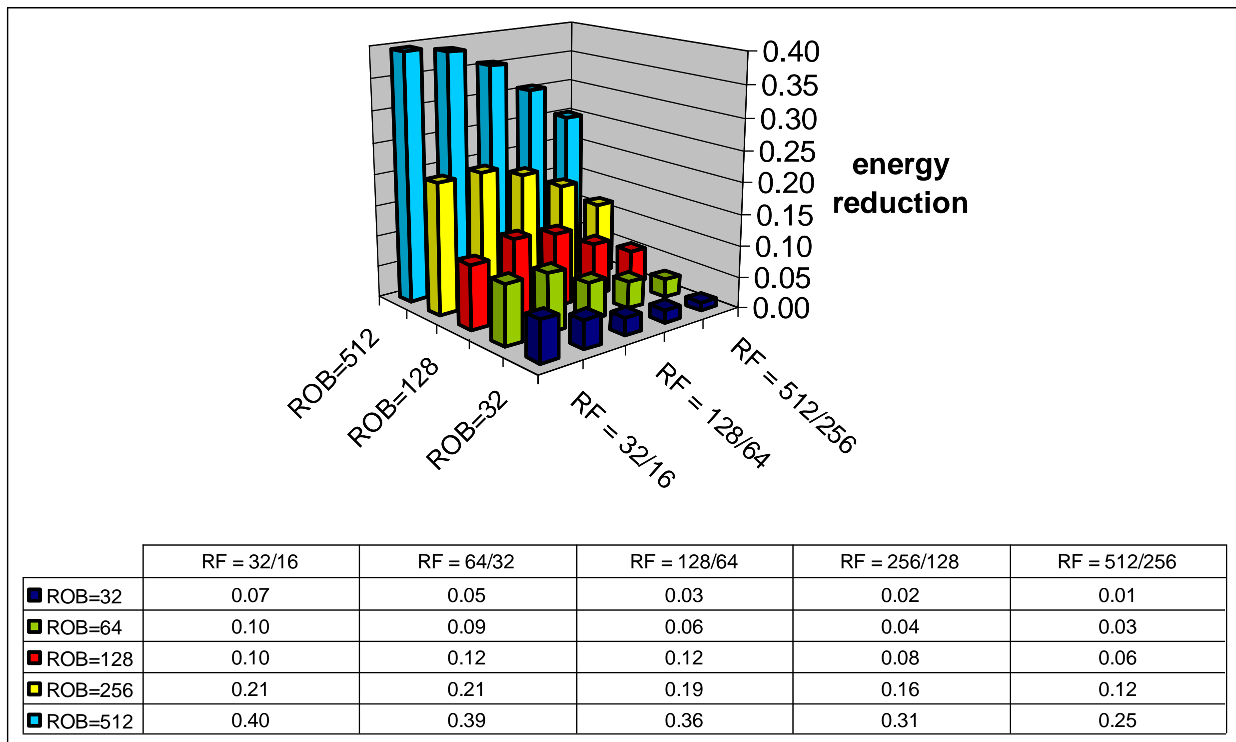
The following question now arises: how much energy can we hope to save by turning off unneeded register and reorder buffer entries? To answer this, we examine the total energy consumption of a processor in the ideal case where we have at any given instant exactly as many integer/floating point rename registers and ROB slots powered on as needed and compare that number to a processor that always has all of its resources turned on. Additionally, for the ideal case we assume zero turn-on delay and zero turn-off delay. Figure 6 below shows the potential energy savings for a 2-issue machine at various processor design points ranging from a small embedded processor (ROB=32, 32/16 integer/floating point rename registers) to an aggressive hypothetical server design with 512 ROB entries and 512/256 rename registers. Each bar

represents the energy reduction at that design point compared to a processor that does not turn off unused resources. Figure 7 below shows the same but for an 8-issue machine. A more detailed discussion of the simulation techniques and processor design parameters used to obtain this data is provided in Chapter 4.



	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.04	0.03	0.02	0.01	0.01
ROB=64	0.06	0.05	0.04	0.02	0.02
ROB=128	0.06	0.07	0.06	0.04	0.03
ROB=256	0.15	0.14	0.13	0.11	0.09
ROB=512	0.30	0.30	0.28	0.26	0.24

Figure 6 Crafty 2-issue energy savings using ideal banking



	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.07	0.05	0.03	0.02	0.01
ROB=64	0.10	0.09	0.06	0.04	0.03
ROB=128	0.10	0.12	0.12	0.08	0.06
ROB=256	0.21	0.21	0.19	0.16	0.12
ROB=512	0.40	0.39	0.36	0.31	0.25

Figure 7 Crafty 8-issue energy savings using ideal banking

Based on the preliminary results above, resource banking has the potential to be quite beneficial. At the low end of the processor design spectrum we save 5-20% of the aggregate processor power; at the high end, nearly 40%—all without any degradation in processor performance.

Of course, in the real world where the laws of physics are assumed to hold one cannot simply switch on a bank of registers in zero time. Similarly, one cannot expect to turn off each register individually—the granularity of control must be over a group of registers to support efficient implementation. These are just some of the factors that make the ideal case an upper bound approximation of the real-world power savings achievable using banking. The focus of Chapter 3 is how such constraints can be overcome to approach the ideal behavior observed in this section.

CHAPTER 3. DYNAMIC RESOURCE BANKING

In resource banking the large, array-like structures contained within a processor are subdivided into discrete units called banks. Structures in this work that are banked are the ROB and RF. Each bank may be voltage gated independent of the others to save power. In dynamic resource banking, the processor attempts to predict at run time how many banks of a given resource will be required to continue execution without that resource becoming a bottleneck. The remainder of that resource's banks are then voltage gated (turned off) and do not consume any dynamic or leakage (static) power after some brief turn-off discharge latency. If at some future time that resource is determined to be in higher demand and has the potential to become a performance bottleneck, the processor will wake up one or more banks of that resource after a longer voltage stabilization latency during which the bank consumes power but is not usable. This is done at run time in contrast to the static banking approaches described earlier, where the compiler passes hints about computationally intensive loops in the binary itself.

Given that the processor must determine how many of various resources to allocate (turn-on) and the potential performance degradation if the turn-off policy frequently incorrectly turns off banks (since it takes longer to turn on a bank and wait for its voltage to stabilize than it does to turn off a bank and wait for it to discharge), the policies guiding this technique are critical. To illustrate the difficulty of creating effective banking policies, consider the figure below in which various resource allocation policies are plotted against integer register demand per epoch.

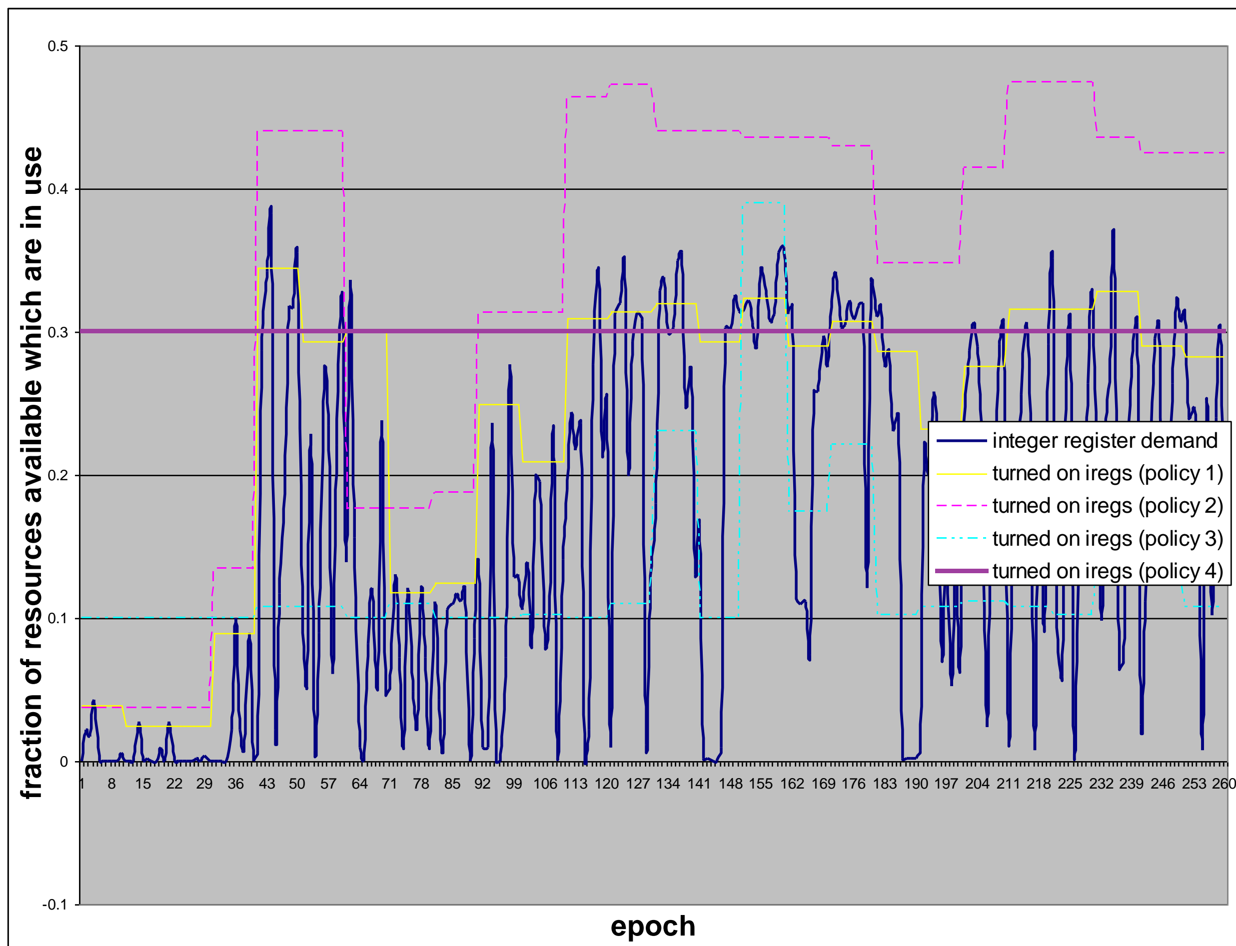


Figure 8 Various hypothetical resource allocation policies

When the number of turned on integer registers exceeds the demand for integer registers, there is no slowdown due to register allocation compared to a hypothetical machine with infinite registers. However, the turned on but unused registers will still consume power and could be turned off without impacting performance. When the number of registers demanded exceeds the number of turned on registers, slowdown will occur but no energy is wasted in unused registers (since they are all in use). The resource allocation policy must thus walk a fine line between energy efficiency and performance. In the ideal case from a performance perspective, at any given instant there are exactly as many resources turned on as needed. From an energy

efficiency perspective, the turned on resources should optimize one of the energy efficiency metrics such as energy-delay product.

In Figure 8, the simplest policy (4) is a static allocation scheme where the number of integer registers is fixed. This is the traditional processor design approach. It is easy to see that static allocation is both wasteful (during intervals when the number of available resources exceeds the demand for that resource) and creates resource bottlenecks (during intervals when the demand exceeds resources availability). Policies 1-3 are dynamic schemes that attempt to track demand and allocate only as many resources as needed at any given instant. Policy 2 is perhaps the most conservative from a performance standpoint since it very rarely limits the demand for registers at the cost of often wasting registers. Policy 1 is a compromise between performance and energy since it closely tracks the maximum demand but frequently limits the registers available. Policy 3 appears to be a rather poor choice in general as it does not track the demand for registers. However, what it does track is energy efficiency. Counter intuitively, policy 3 is actually the most energy efficient of the four in terms of energy delay product. This simple example illustrates several of the complex problems addressed in this chapter.

In Sections 3.1 and 3.2, the ROB and RF design abstractions introduced earlier are re examined from a banked perspective to determine how they must be modified to support banking. In Section 3.3 we examine the voltage stabilization latency issue—how long it could take to turn on and off a bank of resources. Sections 3.4 and 3.5 propose several bank activation and deactivation policies. Section 3.6 proposes several bank allocation policies. Section 3.7 describes the implementation of these policies in the simulator.

3.1. The Banked ROB

To support banking, we make several minor modifications to the implementation of the ROB. First, each bank must be allocated its own supply rail spur with a power gate connected to the global power distribution bus to enable voltage gating. Second, we add logic for a bank controller mechanism and connect it to the power distribution buses. Finally for N banks we add N logical control lines connecting the bank controller to the gates of the voltage gates which determine whether a bank is powered on or off. A simple power distribution abstraction showing the modifications needed to support banking is shown below.

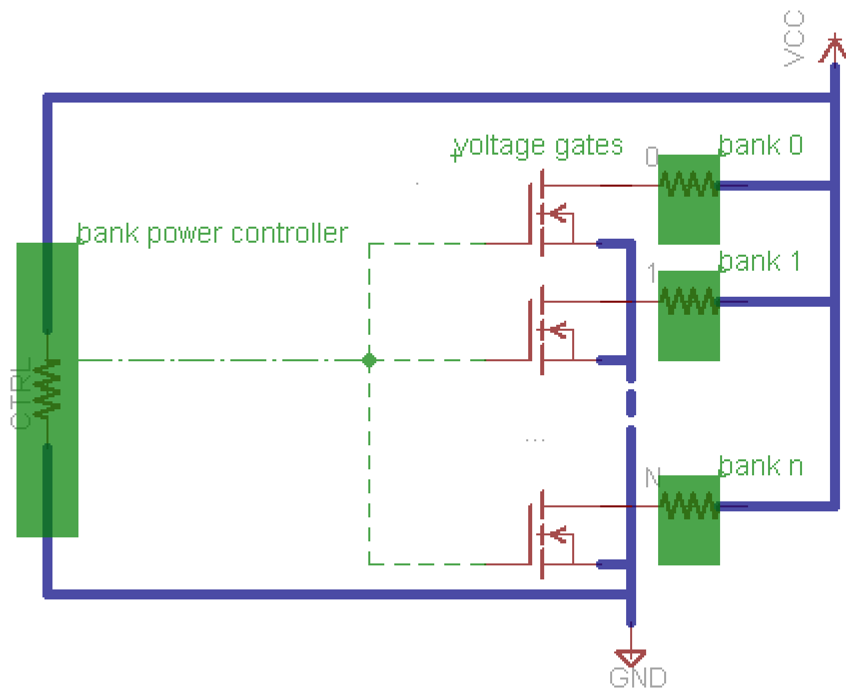
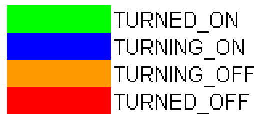


Figure 9 Banking power distribution abstraction

Note that it is possible to gate individual multiplexor and demultiplexor lines as described in [21]. This technique saves power dissipation in the data lines even in turned on banks and has been shown to reduce register file power by nearly 20%. Additionally, turning off multiplexor

lines makes the register file access time faster, potentially speeding up execution. This approach has the downside that it may introduce conflicts in the ROB if the number of writes or reads to a given bank exceeds the number of turned on write or read multiplexors. It also complicates the pipeline considerably since a conflict resolution policy must be performed in subsequent cycles in the case of a conflict. In this work we do not implement structural banking but consider it a technique that can achieve additive power savings on top of the energy reduction we obtain.

The figure below illustrates the ROB banking concept for a case which introduces a stall and one which does not.



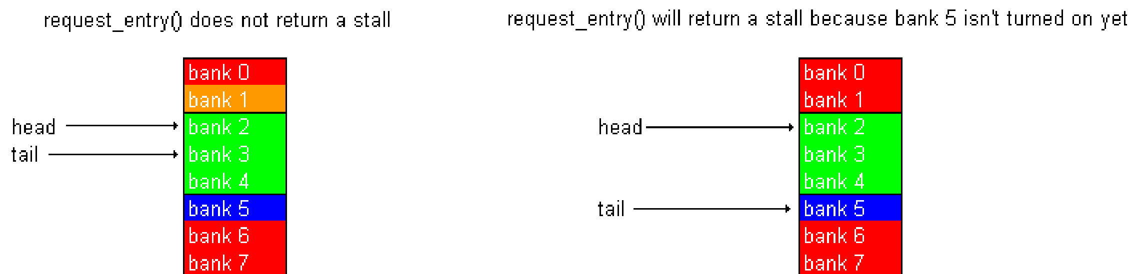


Figure 10 ROB banking can introduce stalls if the tail points to a bank which is not in the TURNED_ON state

In the first case above the tail points to a ROB entry in a turned-on bank so when the processor requests a ROB entry we simply advance the tail pointer and allocate a slot in bank 3. In the second case depicted the tail points to a bank which is in the process of turning on but is

not yet usable so a request for a ROB entry must stall the processor. As the head pointer advances, it leaves empty banks behind which may be deactivated by the bank deactivation policies described later.

It is important to note that ROB resource allocation is done in a manner consistent with the queue abstraction of the ROB rather than the mechanism used in register file renaming described later in this chapter. Thus, the advanced insertion policies described later only apply to the banked register file.

3.2. The Banked RF

The implementation of a banked register file is similar to the banked ROB described above in terms of voltage gating electronics but incurs additional complexity in the bank controller because we wish to be able to insert registers into the RF such that we can turn off unused banks. For example, consider the case where the destination register named in an instruction is R9 but we wish to instead use R17 because bank 1 (which contains R9) has been turned off. To support this we exploit the layer of indirection renaming adds to the register file using the RAT. Note that this is essentially the same idea as traditional renaming. The difference is that the policy by which registers are renamed is now aware of bank utilization. This concept is depicted in the figure below for a monolithic tag-based RF which is the design model we use in this work.

TURNED_ON
TURNED_ON_MAPPED
TURNED_ON_UNMAPPED
TURNING_ON
TURNING_OFF
TURNED_OFF

request_entry() returns P9 using the resource allocation policy

bank 0	P0	P1	P2	P3
bank 1	P4	P5	P6	P7
bank 2	P8	P9	P10	P11
bank 3	P12	P13	P14	P15
bank 4	P16	P17	P18	P19
bank 5	P20	P21	P22	P23
bank 6	P24	P25	P26	P27
bank 7	P28	P29	P30	P31

request_entry() stalls the processor because no free registers in turned-on banks

bank 0	P0	P1	P2	P3
bank 1	P4	P5	P6	P7
bank 2	P8	P9	P10	P11
bank 3	P12	P13	P14	P15
bank 4	P16	P17	P18	P19
bank 5	P20	P21	P22	P23
bank 6	P24	P25	P26	P27
bank 7	P28	P29	P30	P31

Figure 11 Banked RF in operation

The figure below shows how the RAT permits renaming of registers in the RF. In this example an instruction references three registers: r13 and r14 as sources and r17 as a destination. Registers r13 and r14 are already mapped in the RFRT to physical registers 16 and 1 respectively so the instruction is appropriately modified as shown. This is the first use of r17 however (since its mapping is NULL in the RAT) so we call the insertion policy to determine to which register it should now map (in this case the insertion policy determines that P1 is the physical register to which R17 should map). Subsequent uses of R17 are mapped using the updated RAT' to physical register P1.

Inst	RAT	RAT'	Inst'
add r17, r13, r14	r0=>P27	r0=>P27	add P2, P16, P1
	r1=>NULL	r1=>NULL	
	r2=>NULL	r2=>NULL	
	...	...	
	r13=>P16	r13=>P16	
	r14=>P1	r14=>P1	
	r15=>NULL	r15=>NULL	
	r16=>P15	r16=>P15	
	r17=>NULL	r17=>P2	
	...	...	
	rN=>NULL	rN=>NULL	

Figure 12 RAT operation

When a mapped register (non NULL RAT entry) is used as a destination, we have the opportunity to re-map the register using the insertion policy. Thus the mapping is entirely dynamic. Because the RAT allows us to insert a register into an arbitrary turned-on bank, bank insertion policy can play a critical role in the effectiveness of banking. Ideally the insertion policy should cluster entries into as few banks as possible to ensure that at any given instant the most banks are turned off.

3.3. Bank Latencies

As mentioned earlier, it is impossible to instantaneously charge or discharge a capacitive load to some arbitrary voltage. This means that for both bank turn-on and turn-off transitions there will be finite delays during which the bank is in an unstable and unusable state. In the turn-on case, the gate transistor turn-on time must be considered and the bank voltage must stabilize to the point it satisfies the transistor supply requirement for at least as long as the setup time of its slowest transistors. Additionally, the turn-on delay must allow the bank to initialize itself into some known state before it is usable. In the turn-off case the delay is far shorter and is limited only by the fall time of the voltage gate.

Because bank transition latencies have the potential to significantly impact the effectiveness of a banked approach, we opt to be as conservative as possible in selecting turn-on and turn-off latencies. We defer to work done by [19] and use their turn-on and turn-off latencies. Once we have the latencies, we simply divide them by the processor cycle to determine the latencies in terms of clock cycles. We later examine several latencies both more and less conservative to gauge the impact of these numbers on banking performance.

3.4. Bank Deactivation Policies

3.4.1 Simple Deactivation

Possibly the simplest bank deactivation policy is to greedily turn off all empty banks as soon as they become empty. This is, in essence, what the simple deactivation policy does. Each time a dynamic instruction retires, the turn-off policy scans the banks for any that contain no used entries but are in the TURNED_ON state. All such banks are transitioned to the TURNING_OFF state. A pseudocode implementation of simple deactivation is shown below:

```
//turn OFF all unused banks
for(i=0:num_banks){
      if( bank_state[i] == TURNED_ON ){

            //see if the bank is unused
            found_one_used = false
            for(j=0:elements_per_bank){
                  if(   rename[INDEX(i,j)]!= null ){
                  found_one_used = true
                  break
                  }//end used conditional
            }//end for

            //if unused, schedule it for turn off
            if( !found_one_used ){
                  bank_state[i] = TURNING_OFF
                  bank_schedule[i] = current_time + D_OFF
                  return
            }//end unused conditional

      }//end bank powered on conditional
}//end loop over banks
```

3.4.2 Threshold Deactivation

Simple deactivation has several significant pitfalls. For one, it turns off banks without regard for the number of free entries remaining. Thus it could very well turn off the last bank with free entries resulting in a processor stall if the next instruction is available to be issued before an entry is freed. It would perhaps be better to only turn off an unused bank after it has a

history of not being used. A more advanced deactivation policy accounting for this is threshold deactivation where a bank with unused entries is only turned off only after it is unused for some fixed number of cycles, γ. This is described in pseudocode below:

```
//turn OFF all unused banks
for(i=0:num_banks){
      if( bank_state[i] == TURNED_ON && current_time - bank_schedule[i] > γ){

            //see if the bank is unused
            found_one_used = false
            for(j=0:elements_per_bank){
                  if(   rename[INDEX(i,j)]!= null ){
                  found_one_used = true
                  break
                  }//end used conditional
            }//end for

            //if unused, schedule it for turn off
            if( !found_one_used ){
                  bank_state[i] = TURNING_OFF
                  bank_schedule[i] = current_time + D_OFF
                  return
            }//end unused conditional

      }//end bank powered on conditional
}//end loop over banks
```

Note that the only modification to the code is the addition of a check on the age of the bank in the bank powered on conditional block. This policy requires that upon insertion into a turned-on bank the schedule be updated even though that bank is in a stable state rather than a transitional one. This allows us to determine roughly how long the bank has been unused after that instruction is retired.

3.5. Bank Activation Policies

3.5.1 Simple Activation

Perhaps the simplest bank activation policy is to only turn-on a bank when the number of currently turned on banks becomes a bottleneck. In the simple activation policy, we do just this. Each time we do an insertion that fails due to there being no turned on banks, we do the following (in pseudocode) if no banks are currently in the TURNING_ON transition state:

```
// turn-on a single bank if possible
for(i=0:num_banks){
      if( bank_state[i] != TURNED_ON && bank_state[i] != TURNING_ON){
            //transition to the turning on state and set schedule
            bank_state[i] = TURNING_ON
            bank_schedule[i] = current_time + D_ON
            return
      }//endif
}//end bank loop
```

3.5.2 Threshold Activation

A more general case of the simple turn-on policy is threshold activation. In threshold activation, we schedule a new bank for turn-on only after an insertion causes the fraction of total free entries of a resource to fall below some threshold α, $0 < \alpha \leq 1$. Note that threshold activation for nonzero α values effectively precharges banks with variable aggressiveness to reduce or eliminate the turn-on latency if the bank becomes needed in the future. For α=1, the most aggressive activation policy, we always have nearly all the banks turned on. As the α parameter approaches 0, the precharges become more and more conservative until eventually at α=0 there is no precharging (note: this is the same as the simple activation policy described earlier). The advantages of an aggressive precharge policy potentially come at a severe energy

cost, since there is no guarantee that precharged banks will ever be used. The pseudocode for threshold activation is shown below:

```
turned_on_count=0
used_count=0

//count the number of turned on entries
for(i=0:num_banks){
      for(j=0:elements_per_bank){
            if(   bank_state[i]==TURNED_ON ){
                  turned_on_count++

                  if( rename[INDEX(i,j)]!=null ){
                        used_count++
                  }//end used conditional
            }//end turned on conditional
      }//end bank element loop
}//end bank loop

if( used_count / turned_on_count > α ){
      // turn-on a single bank if possible
      for(i=0:num_banks){
            if( bank_state[i] != TURNED_ON && bank_state[i] != TURNING_ON){
                  //transition to the turning on state and set schedule
                  bank_state[i] = TURNING_ON
                  bank_schedule[i] = current_time + D_ON
                  return
            }//endif
      }//end bank loop
}//end α conditional
```

3.6. Insertion (Physical Register Allocation) Policies

For the renaming process in the context of a banked ROB, there are many possible mechanisms by which a register may be allocated. For example, if there are three free physical registers and a rename is requested, the rename allocation policy determines which one should be used. In this section we describe four such policies whose performance will be examined in the subsequent chapter.

3.6.1 First Free Entry

Perhaps the simplest allocation policy is the First Free Entry (FFE) policy. FFE simply grabs the first free register in a turned on bank regardless of any other factors and is described in pseudocode below.

```
//insert into first free entry in turned on bank
tag = &dinst
for(i=0:num_banks){
	if( bank_state[i] == TURNED_ON ){
		for(j=0:elements_per_bank){
			if(	rename[INDEX(i,j)]== null ){
				INSERT(INDEX(i,j), tag)
				return
			}//end unused conditional
		}//end loop over elements within bank
	}//end bank powered on conditional
}//end loop over banks
```

It is easy to think of cases where this is a potentially bad idea. For example if there are two banks and one is nearly full but the first to be searched is empty, the register will be taken from the previously empty bank which might have been able to be turned off next cycle had the insertion occurred in the nearly full bank instead. The interplay is more complex than this, however, since inserting into an empty bank effectively keeps a bank which would be turned off in the future turned on which may actually be beneficial.

3.6.2 First Free Entry in Used Bank

A potentially better policy than FFE is First Free Entry in Used Bank (FFEUB). FFEUB tries to insert into any bank that already has entries. This prevents the rename banks from spreading out as in the case of FFE and encourages clustering within banks to maximize the usefulness of the banks which are turned on. A pseudocode description of FFEUB is shown below.

```
//insert into the most used bank
for(i=0:num_banks){
      if( bank_state[i] == TURNED_ON){
            use_counter=0

            //count the number of entries used in the bank
            for(j=0:elements_per_bank){
                  if(   rename[INDEX(i,j)]!= null ){
                        use_counter++
                  }//end used conditional
            }//end for

            //if the bank is full, set its utilization absurdly low so
            //we dont insert into it
            use_counter = use_counter==elements_per_bank ?
                                    -1 : use_counter

            //if the bank is insertable, do the insertion
            if(use_counter >= 0){
                  for(j=0:elements_per_bank){
                        if(   rename[INDEX(i,j)]== null ){
                              INSERT(INDEX(i,j),tag)
                              break
                        }//end used conditional
                  }//end for
            }
      }//end turned on conditional
}//end loop over banks

FFE()//ensures that if no free entries in used banks we still insert
```

FFEUB requires in the worst case a search of all the banks to find an entry for insertion and may not be optimal in terms of clustering if banks that are searched later retire numerous entries. As described earlier, however, FFEUB ensures that useful entries are clustered together within banks in groups of at least two (since one used entry gives a bank priority over an unused bank). FFEUB is a simplified version of a more complicated policy, First Free Entry in Most Used Bank (FFEMUB).

3.6.3 First Free Entry in Most Used Bank

FFEUB guarantees bank clusters of only two used entries. A better policy, FFEMUB, ensures that useful entries are clustered as closely together as possible up to the maximum bank

size. Essentially what we do in FFEMUB is count the number of used entries in each bank. The bank with the most used entries which also has at least one free entry is selected for insertion. The pseudocode for FFEMUB is described below:

```
//insert into the most used bank
//first build a use table
bank_utilizations[] = new int[num_banks]
for(i=0:num_banks){
	if( bank_state[i] == TURNED_ON){
		use_counter=0

		//count the number of entries used in the bank
		for(j=0:elements_per_bank){
			if(   rename[INDEX(i,j)]!= null ){
				use_counter++
			}//end used conditional
		}//end for

	//if the bank is full, set its utilization absurdly low so we dont
	//insert into it
	bank_utilizations[i] = use_counter==elements_per_bank? -1 : use_counter
	}//end turned on conditional
	else{
		bank_utilizations[i]=-1
	}
}//end loop over banks

//second find the best utilized bank that is not full
best_utilization = -1
best_bank = -1
for(i=0:num_banks){
	if(bank_utilizations[i]>best_utilization){
		best_bank=i;
		best_utilization=best_utilizations[i]
	}
}

//finally insert into that bank
INSERT(INDEX(best_bank),tag)
```

By grouping as many useful entries as possible together in the same banks using FFEMUB, we hope to be able to turn off more banks which are unused. For example, if there are four banks of eight entries per bank and four renamed instructions in flight, if one used entry was inserted per bank then no banks would be able to be turned off. However, if we place all

four used entries in the same bank, the remaining three banks can be voltage gated with significant power savings.

3.6.4 First Free Entry in Least Used Bank

A simple extension of FFEMUB is to prefer the bank with the least used entries. We call this policy FFELUB. This seems like a very bad idea initially (indeed, it was originally selected to demonstrate a poor insertion policy) but it actually works quite well in some scenarios, particularly if the D_ON latency is large. Because it distributes entries more evenly between banks, it is more difficult for a bank to become empty and be powered off. In the case of a large bank turn-on delay, this can be quite beneficial because it ensures that banks which are powered off genuinely are not needed. By the same token, this policy can be quite power hungry because it inhibits banks from emptying and subsequently being gated off.

3.6.5 Temporal Insertion

Ideally new registers would not be allocated in banks whose remaining entries are likely to be retired soon. The reason is that the new entry will then be the only used entry in that bank and may then pose a barrier to that bank being voltage gated. In other words, spatial partitioning alone may be a poor mechanism for allocating new entries because very old and similarly aged instructions more likely to be retired together may be placed in the same bank as younger instructions which will cause the bank to be forced on by the younger instructions. This sounds complex but it is really an analog of the concept of temporal locality. A pseudocode implementation of a simple temporal insertion policy based on mean bank age is shown below.

```
//insert into the bank with the oldest average age
mean_bank_ages = new float[num_banks]
```

```
for(i=0:num_banks){
      temp = 0.0f
      count = 0.0f

      //compute the average age of this bank
      for(j=0:elements_per_bank){
            if(rename_table[INDEX(i,j)]!=0 && bank_states[i]==POWERED_ON){
                  temp+=last_used_history[INDEX(i,j)]
                  count+=1.0f
            }
      }//end loop over elements per bank

      if(count != 0){
            mean_bank_ages[i]=temp/count
      }
      else{
            mean_bank_ages[i]=0
      }
}//end loop over banks

best_bank=-1
best_bank_element=-1
best_value=0.0f

for(i=0:num_banks){
      if(mean_bank_ages[i]>=best_value){
            for(j=0:elements_per_bank)//check  if  any  in  the  best  bank  are
free{
                  if(rename_table[INDEX(i,j)]==0                              &&
bank_states[i]==POWERED_ON){
                        best_bank=i
                        best_bank_element=j
                        best_value=mean_bank_ages[i]
                  }
            }
      }
}

if(best_bank < 0 || best_bank_element < 0){
      return -1//insertion fails
}
else{
      return INDEX(best_bank,best_bank_element)//insertion succeeds
}
```

This policy is difficult to analyze theoretically but its performance implications are explored in the subsequent chapter.

3.7. Implementation

The architectural simulator used in this work is SuperESCalar (SESC) [25]. SESC was designed primarily for speedy simulation and thus does not model actual register renaming. SESC simulates the power and usage of the register file, but does not actually place anything in it. Instead, SESC abstracts the renaming process by maintaining a counter of the number of free registers and a linked list dependency chain that connects dependent dynamic instructions together. Using this approach it can be quickly determined whether a pending instruction can be issued by checking this counter—if greater than zero then there are free registers available this cycle; if not, the processor must stall and wait for a register to become available. Updates to a renamed register can be quickly and easily propagated along the dependency chain.

This poses a problem for simulating a banked scheme, however, since banking requires knowledge of where in the register file new entries are placed. In a real processor, the RAT keeps track of renamed registers ensuring previously renamed destination registers are reflected in the source fields of newer instructions. As with the RF, SESC models the power consumption and accesses to the RAT but the RAT itself has no functionality. Instead, SESC links dependent instructions together in a linked list structure and awakens dependencies in the list when values are updated out of order. Again, this linked list implementation proves problematic since we wish to design a banking scheme that requires these registers be placed physically rather than abstractly to ensure proper modeling.

Implementing renaming in SESC required significant modification to the SESC code. First, it was observed that SESC uses an array of dynamic instruction objects to represent the ROB. Since each dynamic instruction object has at most a single write operand and each is

created dynamically from a static instruction during execution, no two dynamic instruction objects in the pipeline can have identical pointer values (barring corner cases like replay queue trapping and mispredicted path instructions). It is more accurate to say that no two dynamic instruction objects in the pipeline that will be retired will have identical pointer values. Thus we can use the pointer value of a dynamic instruction as a tag that uniquely maps into some implementation of a register file or ROB.

Now we have the machinery needed to simulate a more realistic register file or ROB structure. In either case, the resource is viewed abstractly as an array of pointers. When a dynamic instruction requests a destination register and the free resource counter is greater than zero, the array of pointers representing the entries available for that resource is searched using one of the aforementioned allocation policies until a free entry is found. The pointer to the dynamic instruction requesting the register is then copied into that entry. This process is shown in the figure below. Note that the allocated resource could be assigned any entry whose tag is null but the second entry is arbitrarily selected for this example.

&dinst = 0xAFFA

bank 0		tag
	entry 0	0xFAFA
	entry 1	**null**
	entry 2	0xFFAA
	entry 3	0xFAAF
bank 1	**entry 4**	**null**
	entry 5	**null**
	entry 6	**null**
	entry 7	**null**

bank 0		tag
	entry 0	0xFAFA
	entry 1	**0xAFFA**
	entry 2	0xFFAA
	entry 3	0xFAAF
bank 1	entry 4	null
	entry 5	null
	entry 6	null
	entry 7	null

Figure 13 RF mappings before and after allocation of a new register entry

Later, when the dynamic instruction is retired, the array is searched for its pointer and the tag value at its entry is cleared (set to null). An instruction issued in a later cycle may use that entry.

To enforce banking, the array of pointers is subdivided into logical banks that are independently in one of four states: POWERED_OFF, POWERING_OFF, POWERED_ON, and POWERING_ON. The valid transitions between states are shown in the figure below. The initial state of the resource is powered on. A resource may only be allocated from a register that is in the POWERED_ON state. Each cycle, a function is called whose objective is to find banks to turn on or off based upon the banking policies described earlier. If a bank is chosen for turn on it is transitioned immediately to the powering on state; similarly, if a bank is chosen for turn off it is placed immediately in the powering off state. In either case, a schedule is entered for that bank specifying at which cycle the bank becomes either fully on or fully off based upon the rail latencies D_ON and D_OFF. Note that a bank in the POWERING_ON state can be immediately transitioned to the POWERING_OFF and vice versa. Each cycle the banks are checked against this schedule to determine whether a bank which was scheduled to turn on or off this cycle exists and if so the bank transitions to the appropriate state.

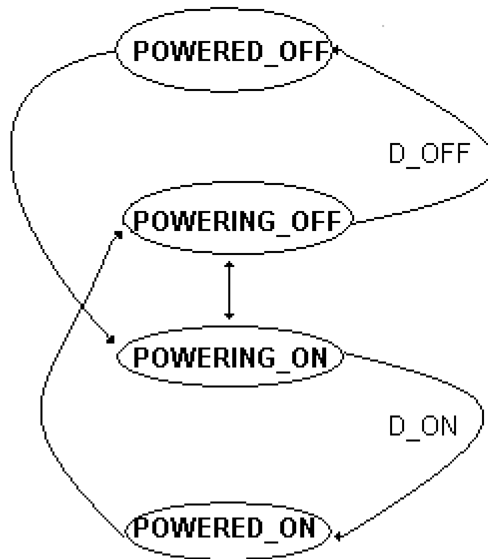


Figure 14 Bank FSM

3.8. Banking Power and Area Overhead

For simplicity, and to provide a conservative power estimate, we assume that the power consumed by the resource is not reduced until the bank transitions to the powered off state. More optimistic models are possible (e.g., the resource begins discharging as soon as it transitions to the powering off state but we assume it still consumes power) but are not explored in this work since our main objective is determining the feasibility of banking rather than developing an ultra precise power model. Additionally, if this conservative estimate produces favorable results then the addition of a more complex state-based power model would only make the results more favorable. Coupled with the rough power estimates produced by the simulator itself, this is a reasonable simplification.

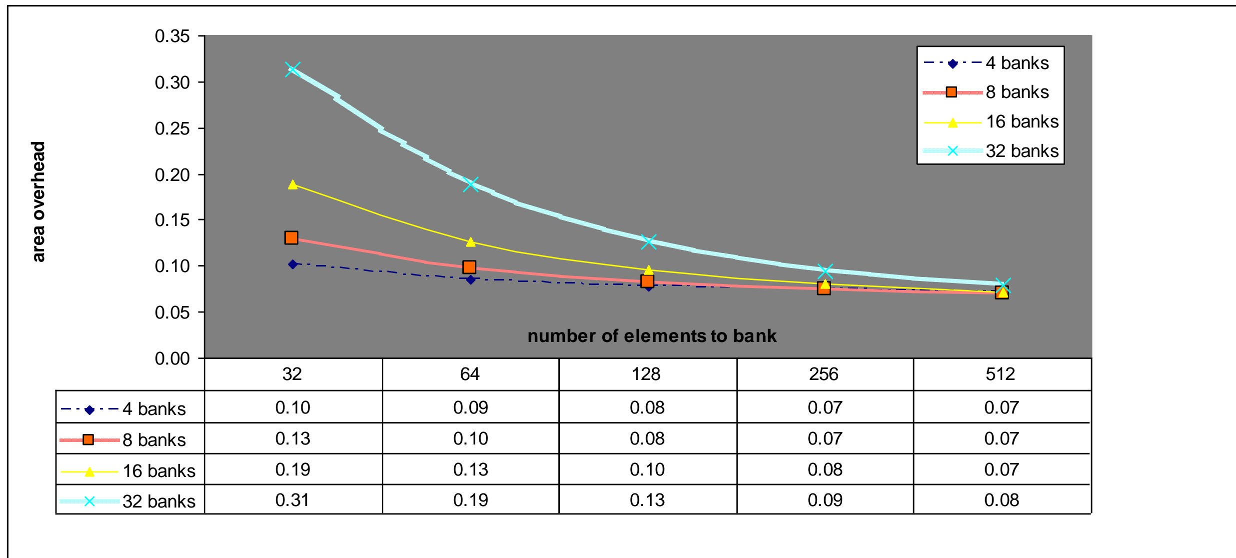
The next implementation consideration is how energy consumption is scaled when using banking (*i.e.*, given the energy consumed by the resource when all its banks are turned on, how

much energy will the fraction $0 < \alpha \leq 1$ consume). For hierarchical bit lines the power reduction can be fitted to a quadratic model based upon the ratio of turned on to turned off banks as described in [21]. For monolithic bit lines the power reduction can be fitted to a linear model as in the same work. The linear model is more conservative in terms of implementation complexity and energy savings so again, this is the model we use.

Another question is how much area overhead banking introduces. Banking requires only minor modifications to existing RF and ROB implementations so its area overhead should be quite small. The most complex policy in terms of hardware implementation, temporal insertion, is used to compute the banking area overhead.

Each bank keeps a resource usage bit vector with a single bit per entry (e.g., a 64/32 register RF arranged into 8 banks would contain 96 bits in its controller vector subdivided into 8/4 sub vectors). Upon allocation of an entry, the corresponding bit is set; upon its removal the bit is cleared. We also add a status bit per bank which indicates to the bank controller whether that bank is currently usable or not. Additionally we add a register per bank with as many bits as there are entries each bank. These registers are used to compute the average age of the entries in the bank. Finally, each bank requires a temporal history register accurate up to T time units of t bits where t=log2(T). For example, if we sample the bank age every 256 cycles and need a temporal history with a resolution of 2^16 cycles we would use an 8-bit temporal history register per bank. If we assume the number of bits used by the banking scheme is linearly proportional to the size of the structure it controls in bits (a very conservative estimation since the size of a structure such as the RF increases quadratically with issue width as discussed earlier), we obtain the area overheads shown in the figure below. We see that with 8 banks (the number of banks

used later for simulation purposes) we have a worst-case area overhead of 13% which is quite acceptable, especially given the number of highly conservative assumptions used to achieve it.



	32	64	128	256	512
4 banks	0.10	0.09	0.08	0.07	0.07
8 banks	0.13	0.10	0.08	0.07	0.07
16 banks	0.19	0.13	0.10	0.08	0.07
32 banks	0.31	0.19	0.13	0.09	0.08

Figure 15 Area overhead approximation for a 2-issue machine

A final consideration in our implementation is the energy consumed by the banking hardware itself (which must enforce the turn-on/turn-off policies). As mentioned earlier, we opt for a conservative, easy to approximate upper limit of energy consumption for this category and will examine the impact of a wider range of controller energies based upon this approximation later. As an initial estimate we assume that the bank controller power consumption is linearly proportional to the area overhead it incurs. For example, we assume that a 32-entry banked structure with 8 banks consumes 13% additional energy using the area estimates given earlier. Again, the actual power consumption of a banked structure will be far lower because the banking hardware is doing less power intensive work than a powered on RF with many ports active but this gives us a reasonable ceiling estimate of power consumption.

CHAPTER 4. EXPERIMENTAL RESULTS

4.1. Simulation Methodology

The proposed banking scheme for the RF and ROB was implemented using SESC [25], a fast cycle-accurate simulator for the MIPS instruction set written in C++. SESC's power models are based upon Wattch [26] and Cacti [27] estimates. The less interesting Wattch and Cacti parameters not defined below (such as the electrical conductivity of doped silicon) were left at the defaults defined in the SESC package. The core code of SESC was significantly modified to support the enhancements described in this work. The processor configurations utilized in all test runs are shown below in Table 1. Parameters for memory structures such as the caches are shown below in Table 2. Parameters in braces indicate that various values for that parameter were tested. Parameters in brackets indicate that the parameter value is defined elsewhere.

Table 1 Basic processor core configuration

Issue width, pipeline depth, retire width	{2,3,4,6,8}
Fetch width	([issue]/6+1)*6
IQ size	2*[fetch width]
Integer registers	{32,64,128,256,512}
Floating point registers	[iregs]/2
Branch predictor	ogehl, 6 tables of 2048 entries, 2-associative BTB
Clock distribution network	H-tree
Miscellaneous	32-bit architecture out-of-order pipeline 1GHz clock frequency 45nm process size

Table 2 Memory structure configuration

L1 I$	64KB 2-way set associative cache with WB LRU and 64B line size, 2 cycle hit delay, 1 cycle miss delay
L1 D$	64KB 2-way set-associative with WB LRU and 64B line size, 2 cycle hit delay, 1 cycle miss delay

L2 U$	512KB, 8-way set-associative with WB LRU and 64B line size, 10 cycle hit delay, 4 cycle miss delay
Main memory	64B blocks, 490 cycle hit delay

The cache and branch predictor configurations are aggressive but reasonable for modern processors. The branch predictor is large to eliminate mispredicted branch path effects as much as possible from the results. The caches are large to reduce the effect of misses. In setting the parameters for both structures our objective is to reduce the number of stall cycles the processor will incur to arrive at a conservative estimate of the ROB and RF energy savings achievable using banking. We observed that with frequent cache misses, for example, the bank turn-on latency played a less critical role because banks had more time to transition into the TURNED_ON state (during the miss service latency) before their resources would be needed resulting in larger reductions in ROB and RF energy consumption than in a processor with larger caches and fewer cache misses.

Table 3 below describes the parameters relevant to ROB/RF banking.

Table 3 Bank structure configurations

Number of ROB, RF banks	{4, 8, 16}
ROB, RF bank activation policy	{simple, threshold}
ROB, RF bank deactivation policy	{simple, threshold}
RF insertion policy	{FFE, FFEUB, FFEMB, temporal}
Bank turn-on latency	{0, 1, 2, 4, 8, 16, 32, 64, 128} cycles
Bank turn-off latency	{0, 1, 2, 4, 8} cycles
Bank manager energy overhead	{none, optimistic, conservative}

Table 4 below lists the benchmarks used to gauge the effectiveness of banking. A variety of integer and floating point benchmarks from a wide range of scientific and practical applications were selected.

Table 4 Benchmarks

benchmark	class	Description	suite
crafty	integer	chess engine	SPEC2000
h264	integer	h.264 video decoding	SPEC2006
bzip2	integer	file compression	SPEC2006
specrand	floating point	pseudo random number generation	SPEC2006
namd	floating point	molecular bond simulation	SPEC2006
mp3 decode	integer	mp3 audio decoding	ISO reference implementation
judoku	integer	Sudoku puzzle solver	author's programming project

We use a MIPS cross compiler on a 32-bit machine with O2 level optimization to build the above binaries from C and C++ source code. The SPEC benchmarks use their reference or test inputs and each benchmark is run for 100 million instructions, a simplification of the approach taken in [28]. We observed that simulating even 50 million instructions was enough to get an accurate indication of the energy savings achieved using banking for the entire benchmark run but instead use 100 million instructions to be as conservative as possible.

4.2. Baseline Measurements

To gauge the theoretical gains achievable using RF and ROB banking, we ran several benchmarks with the banking scheme configured as shown in Table 5 below.

Table 5 Ideal banking configuration

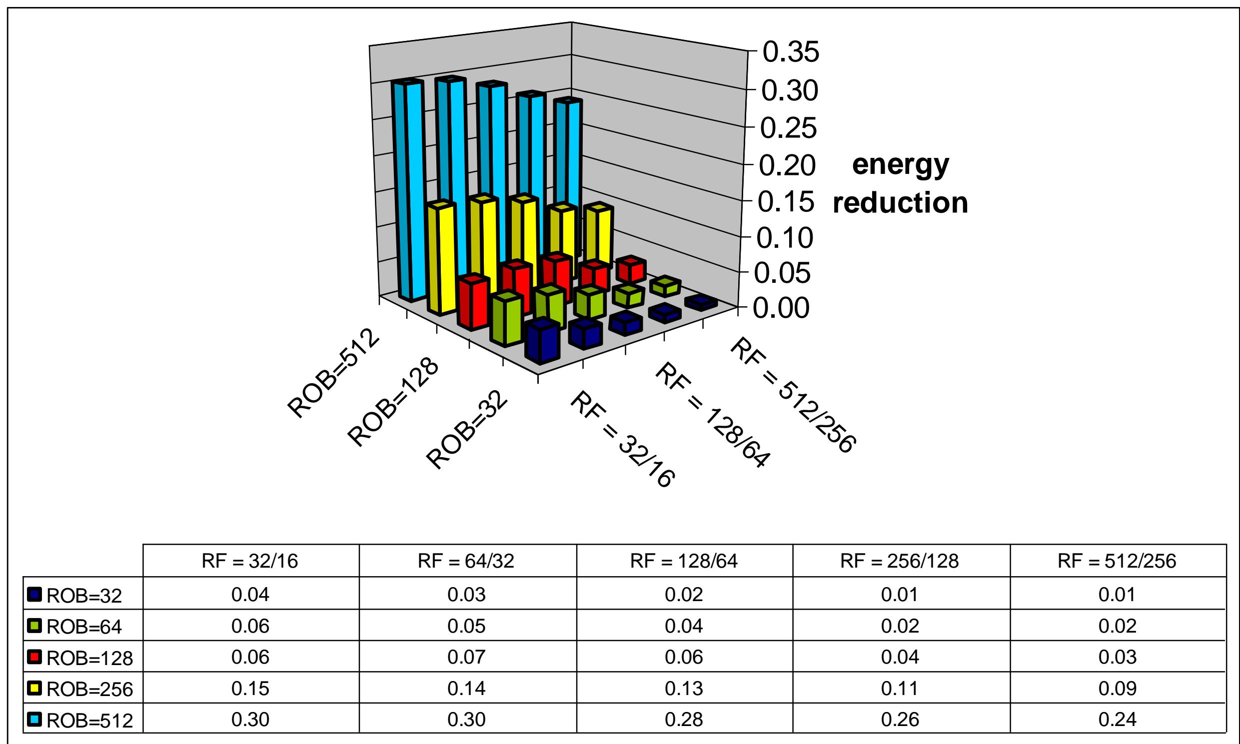
Number of ROB, RF banks	8
ROB, RF bank activation policy	simple
ROB, RF bank deactivation policy	simple
RF insertion policy	FFE
Bank turn-on latency	0 cycles
Bank turn-off latency	0 cycles
Bank manager energy overhead	None

Although this configuration is quite unrealistic, it serves as a good starting point to determine how much energy reduction we can hope to achieve using banking. First we attempt to determine how well the energy savings scale with processor aggressiveness (in terms of issue width, ROB size, and RF size) to determine the regions of the processor design spectrum to which banking applies. The design space tested is shown in the table below. Cells shaded in green are considered realistic design points for modern processors and unshaded cells are hypothetical design points to gauge the scalability of banking.

Table 6 Processor aggressiveness design points

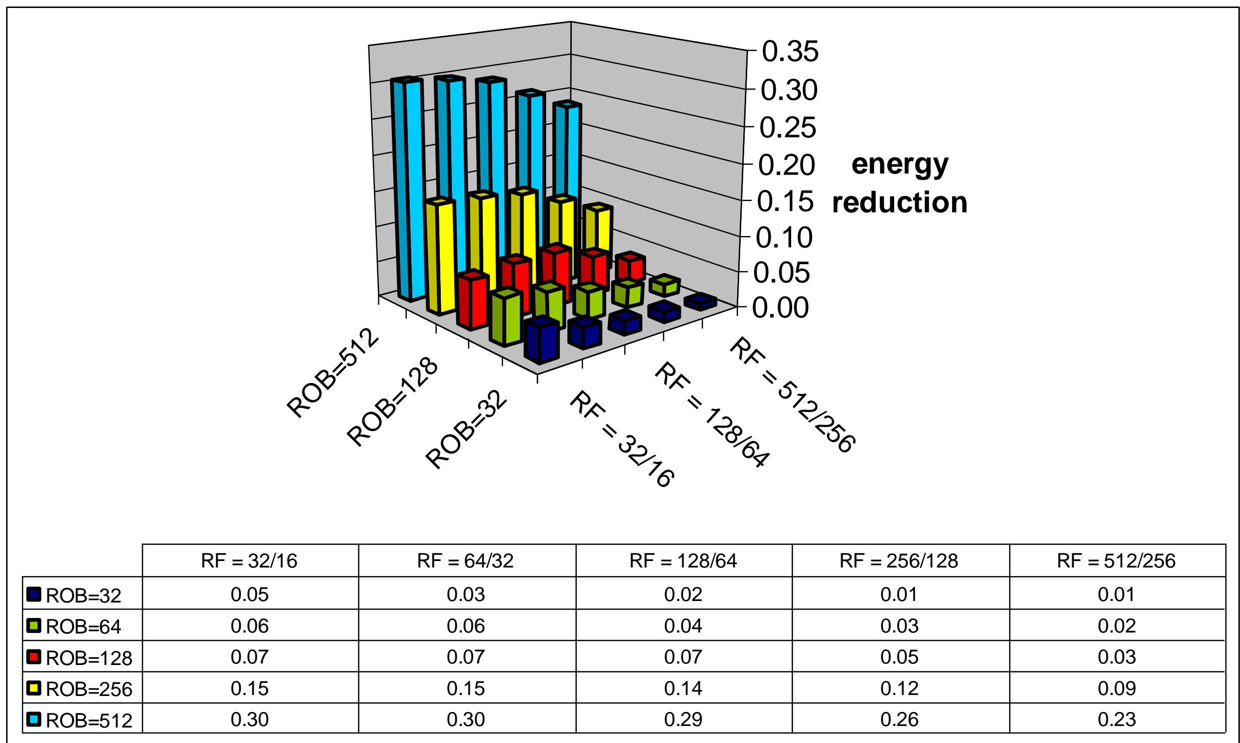
		ROB				
		32	64	128	256	512
fp/int regs	32/16					
	64/32					
	128/64					
	256/128					
	512/256					

The results of these runs are shown in the figures below. In each figure the fraction of aggregate processor energy saved using banking is shown at the ROB and RF sizes described in Table 6 (higher bars indicate larger energy savings).



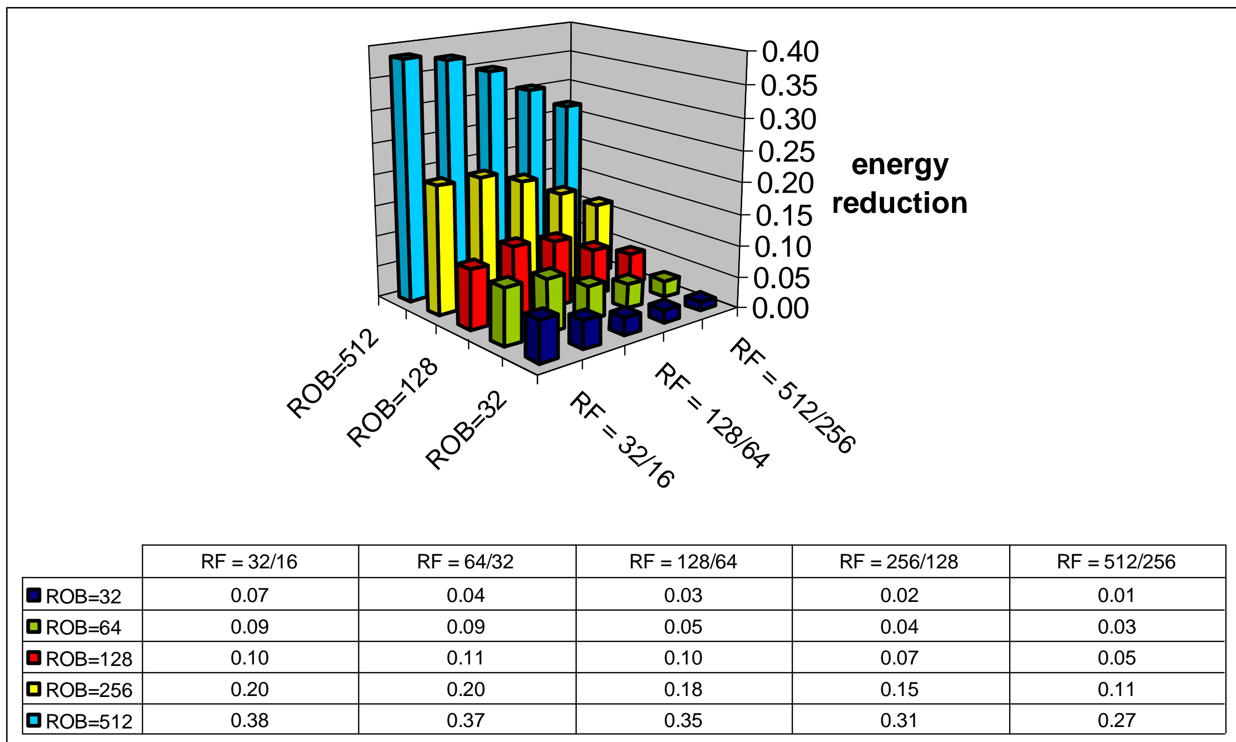
	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.04	0.03	0.02	0.01	0.01
ROB=64	0.06	0.05	0.04	0.02	0.02
ROB=128	0.06	0.07	0.06	0.04	0.03
ROB=256	0.15	0.14	0.13	0.11	0.09
ROB=512	0.30	0.30	0.28	0.26	0.24

Figure 16 Crafty 2-issue ideal case energy reduction



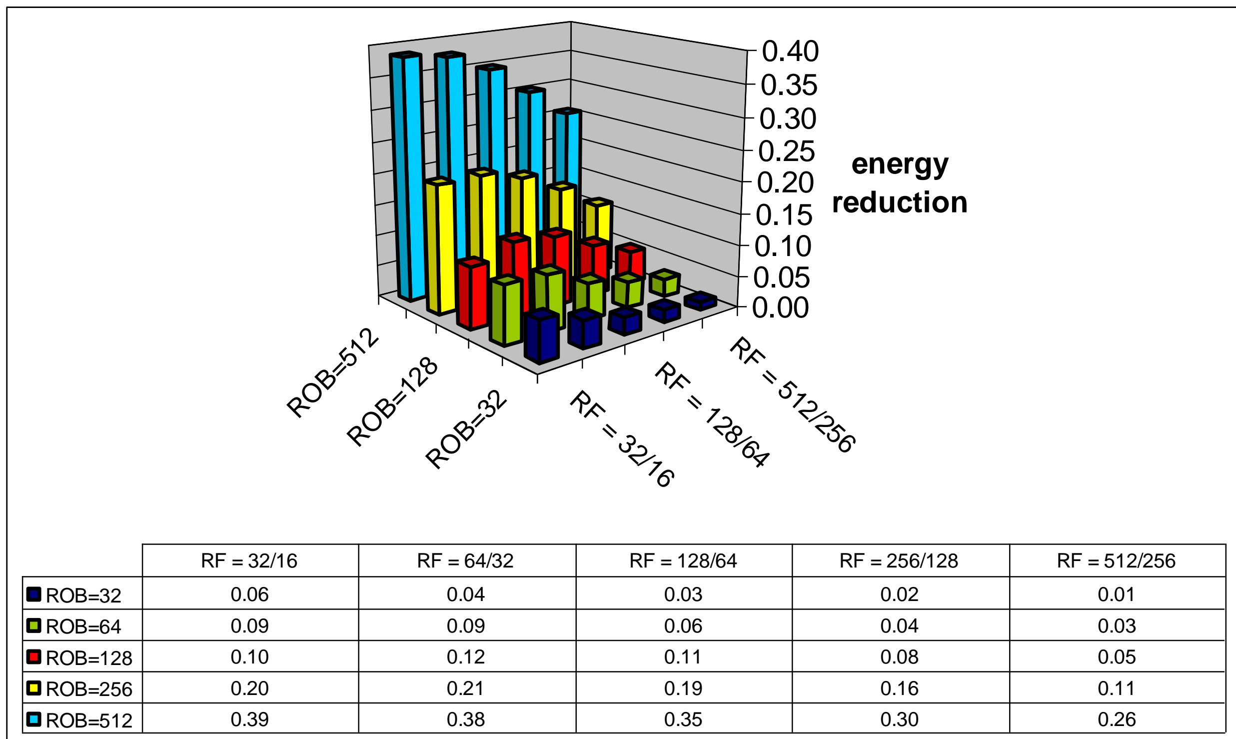
	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.05	0.03	0.02	0.01	0.01
ROB=64	0.06	0.06	0.04	0.03	0.02
ROB=128	0.07	0.07	0.07	0.05	0.03
ROB=256	0.15	0.15	0.14	0.12	0.09
ROB=512	0.30	0.30	0.29	0.26	0.23

Figure 17 Crafty 3-issue ideal case energy reduction



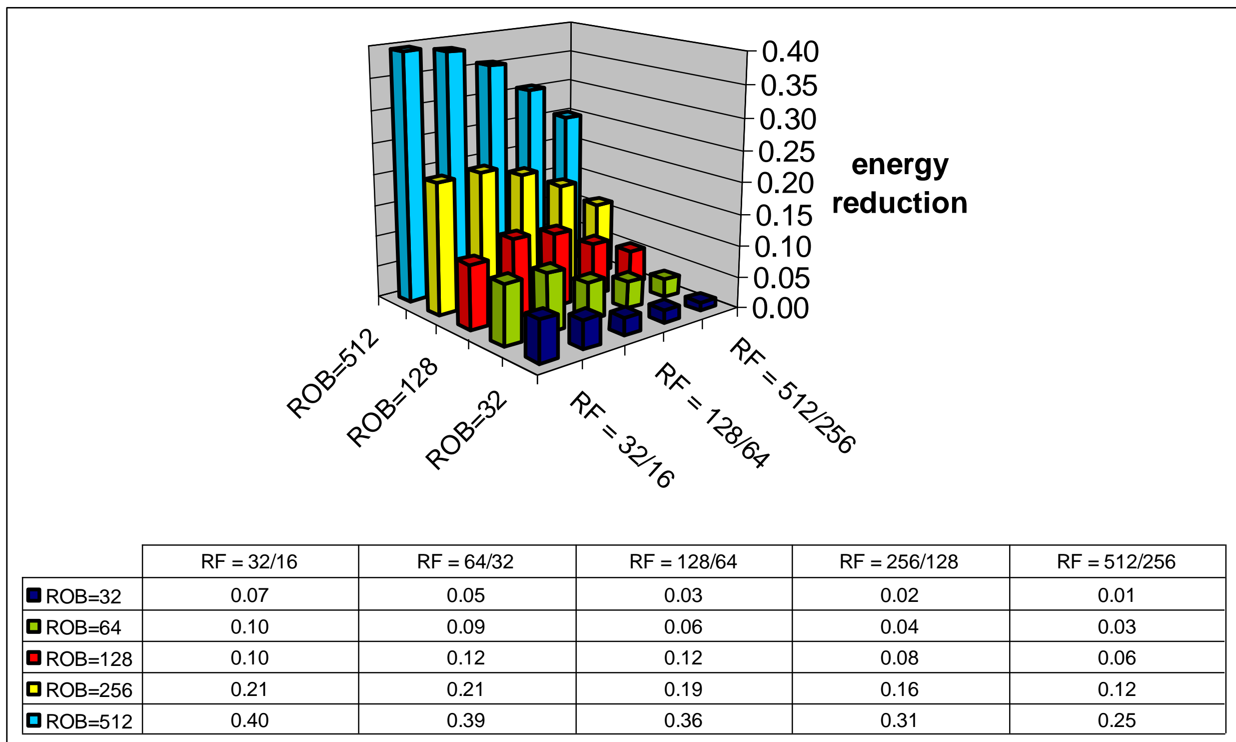
	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.07	0.04	0.03	0.02	0.01
ROB=64	0.09	0.09	0.05	0.04	0.03
ROB=128	0.10	0.11	0.10	0.07	0.05
ROB=256	0.20	0.20	0.18	0.15	0.11
ROB=512	0.38	0.37	0.35	0.31	0.27

Figure 18 Crafty 4-issue ideal case energy reduction



	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.06	0.04	0.03	0.02	0.01
ROB=64	0.09	0.09	0.06	0.04	0.03
ROB=128	0.10	0.12	0.11	0.08	0.05
ROB=256	0.20	0.21	0.19	0.16	0.11
ROB=512	0.39	0.38	0.35	0.30	0.26

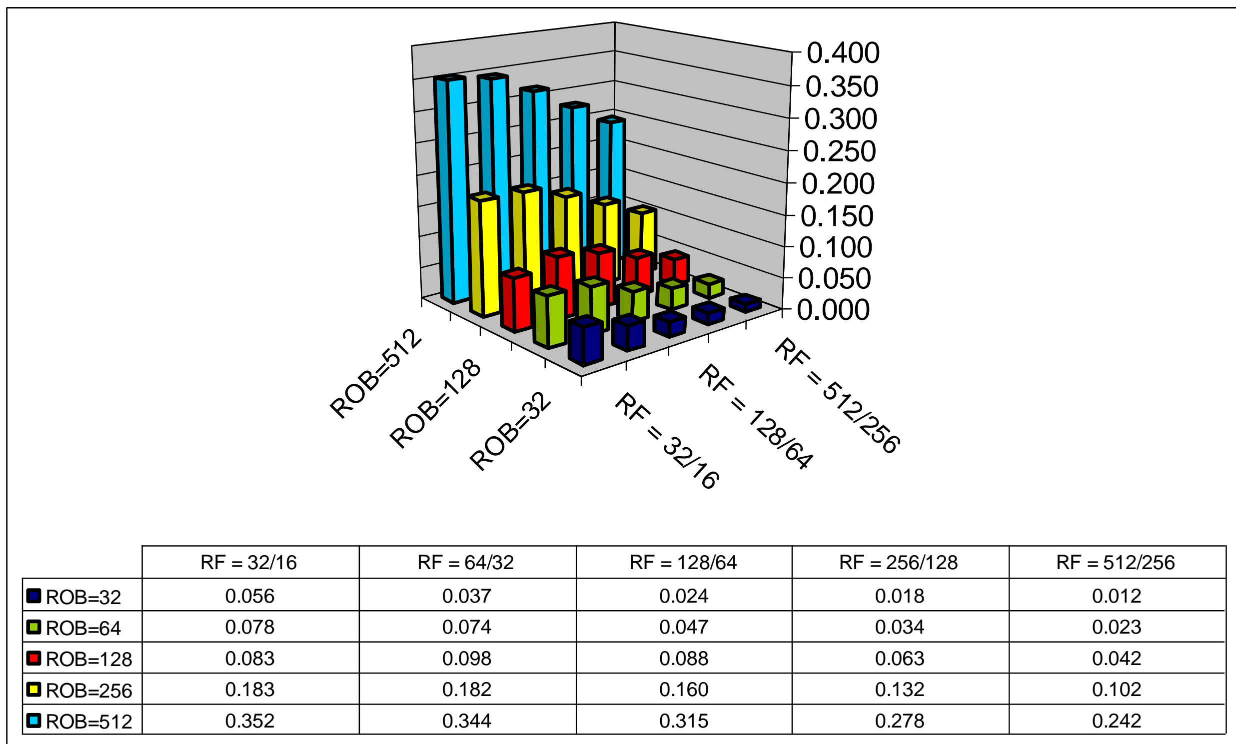
Figure 19 Crafty 6-issue ideal case energy reduction



	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.07	0.05	0.03	0.02	0.01
ROB=64	0.10	0.09	0.06	0.04	0.03
ROB=128	0.10	0.12	0.12	0.08	0.06
ROB=256	0.21	0.21	0.19	0.16	0.12
ROB=512	0.40	0.39	0.36	0.31	0.25

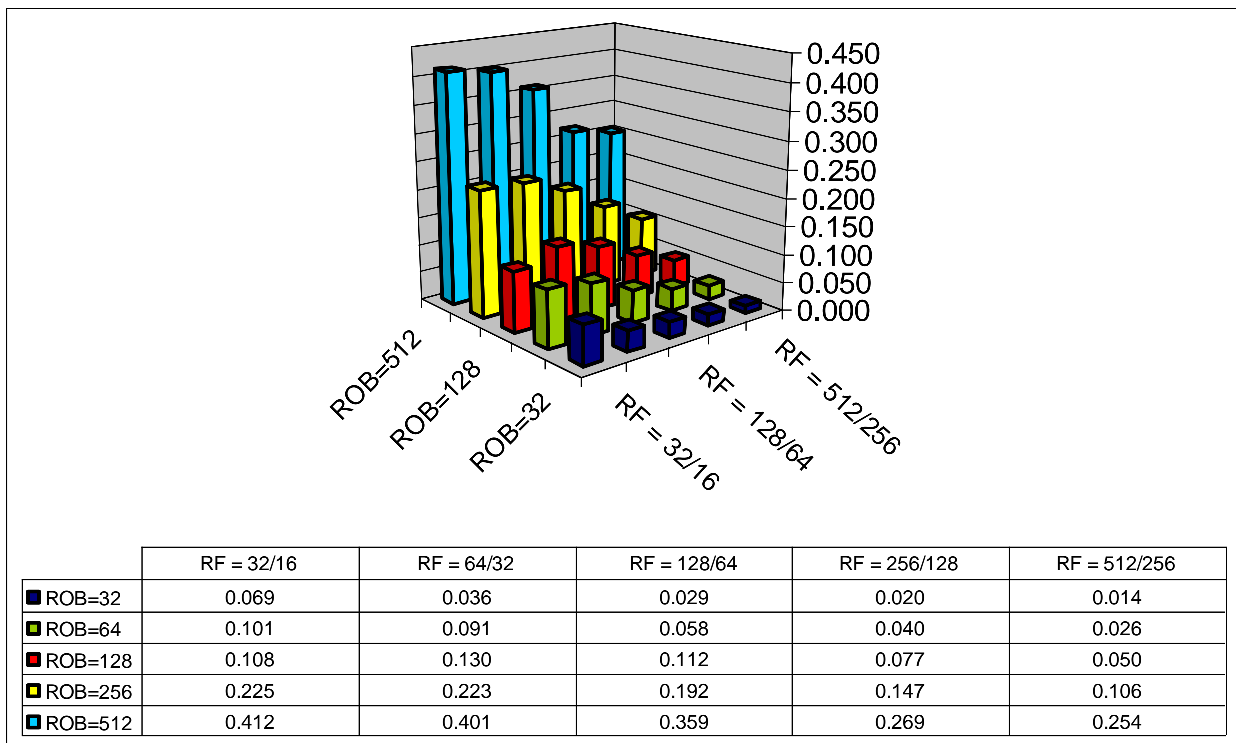
Figure 20 Crafty 8-issue ideal case energy reduction

We then performed the same test on the remaining benchmarks to see if the energy savings were an artifact of the crafty benchmark. The results of the 4-issue runs are shown below.



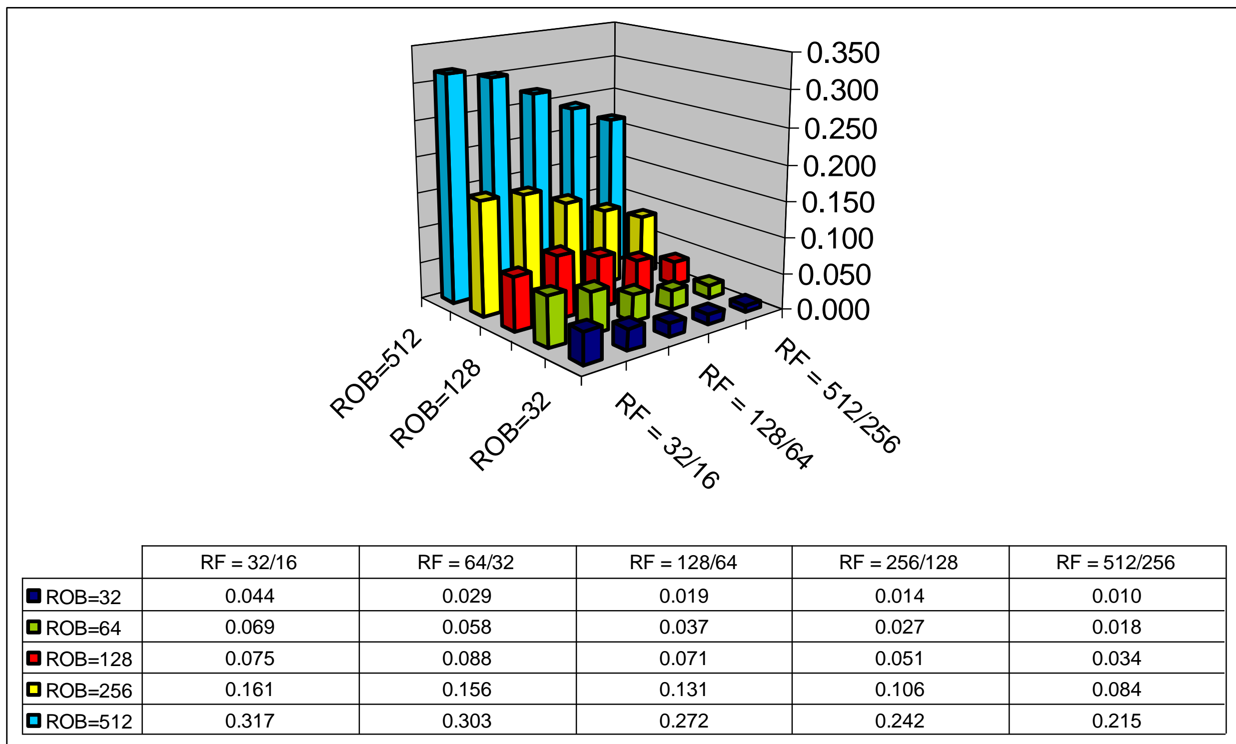
	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.056	0.037	0.024	0.018	0.012
ROB=64	0.078	0.074	0.047	0.034	0.023
ROB=128	0.083	0.098	0.088	0.063	0.042
ROB=256	0.183	0.182	0.160	0.132	0.102
ROB=512	0.352	0.344	0.315	0.278	0.242

Figure 21 H264 4-issue ideal case energy reduction



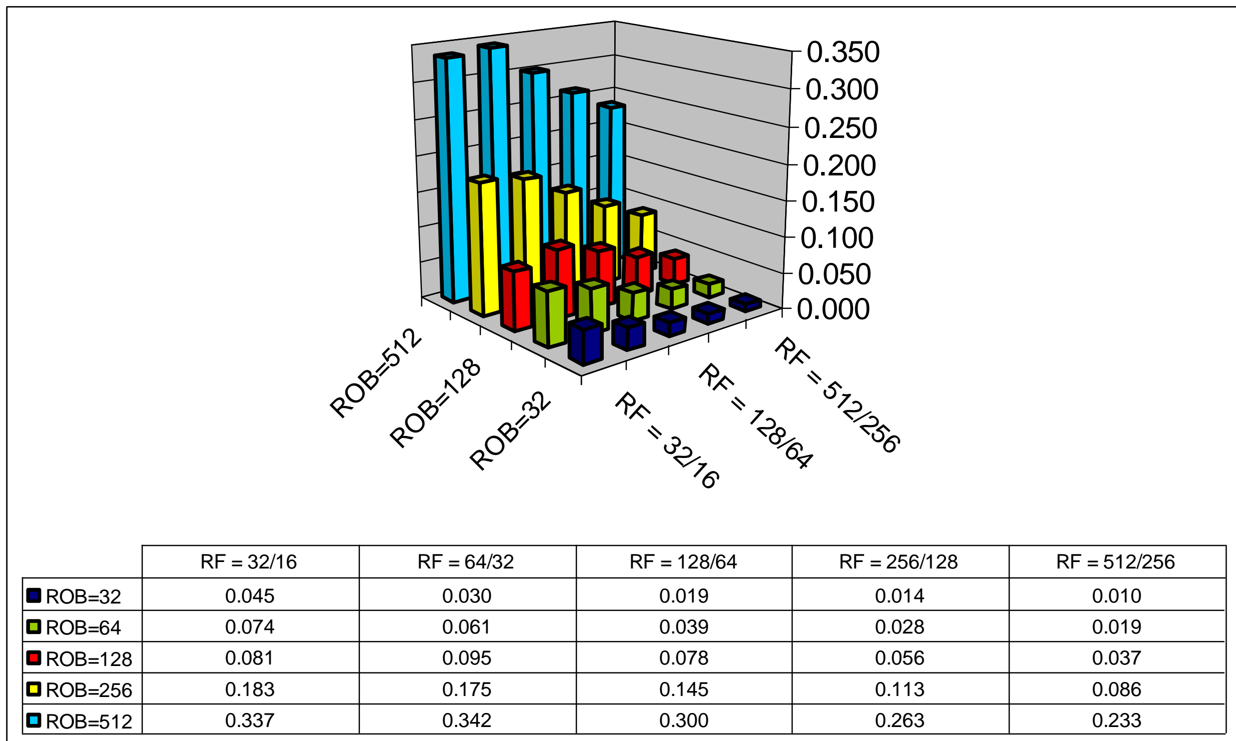
	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.069	0.036	0.029	0.020	0.014
ROB=64	0.101	0.091	0.058	0.040	0.026
ROB=128	0.108	0.130	0.112	0.077	0.050
ROB=256	0.225	0.223	0.192	0.147	0.106
ROB=512	0.412	0.401	0.359	0.269	0.254

Figure 22 MP3 4-issue ideal case energy reduction



	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.044	0.029	0.019	0.014	0.010
ROB=64	0.069	0.058	0.037	0.027	0.018
ROB=128	0.075	0.088	0.071	0.051	0.034
ROB=256	0.161	0.156	0.131	0.106	0.084
ROB=512	0.317	0.303	0.272	0.242	0.215

Figure 23 Bzip2 4-issue ideal case energy reduction



	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	0.045	0.030	0.019	0.014	0.010
ROB=64	0.074	0.061	0.039	0.028	0.019
ROB=128	0.081	0.095	0.078	0.056	0.037
ROB=256	0.183	0.175	0.145	0.113	0.086
ROB=512	0.337	0.342	0.300	0.263	0.233

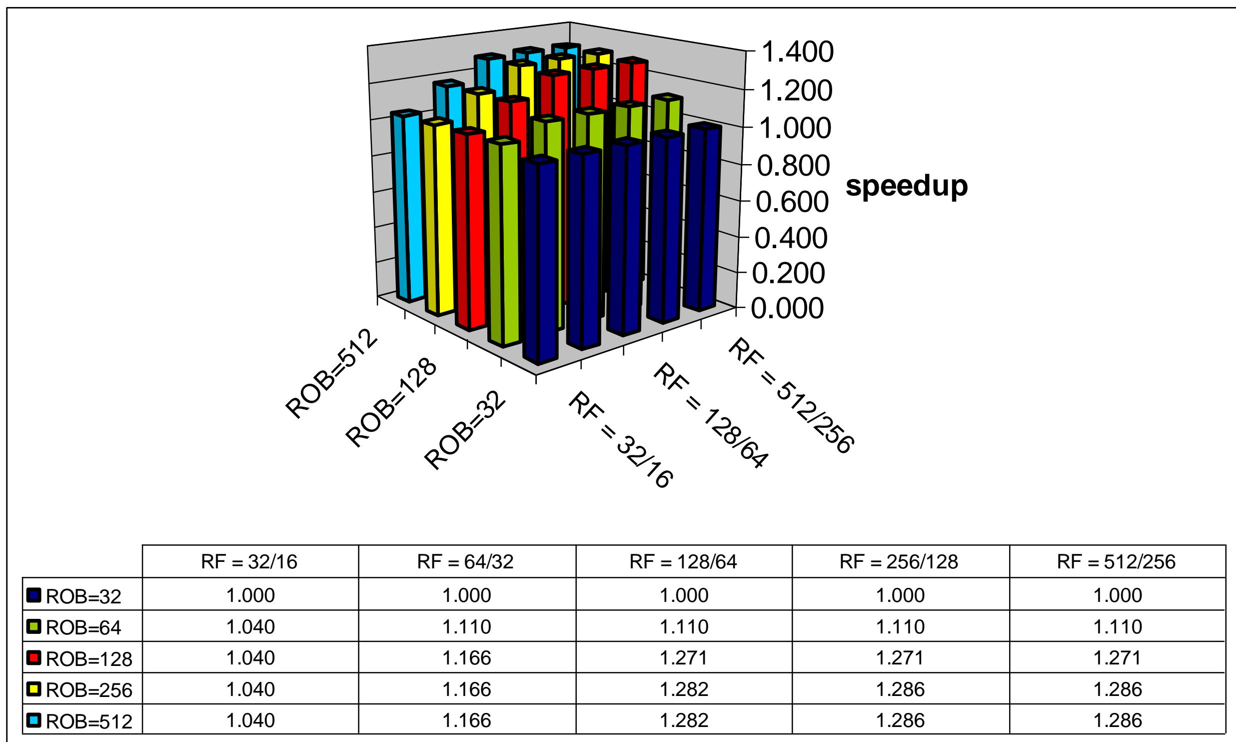
Figure 24 Namd 4-issue ideal case energy reduction

Ideal banking proves to have significant potential across a wide variety of benchmarks and processor configurations, achieving aggregate energy improvements ranging from 1% to 40% (2-21% over reasonable design configurations) with no performance degradation. It is interesting to note that the potential energy savings using banking appear similar in magnitude between benchmarks, suggesting that this technique works well with many different resource use patterns.

Presumably the very large energy reduction for an 8-issue, 512 ROB, 512/256 register machine is the result of many of those registers and ROB entries being unused. Similarly, the small energy reductions for small ROBs and register files indicate that most of the banks are always activated. In the first case, having a large number of frequently unused registers can still provide significant speedup because computationally intensive regions of code may be completed more rapidly. In the second case, for mobile environments where the workload is infrequent, banking can still provide energy savings during periods when there is no work to be done.

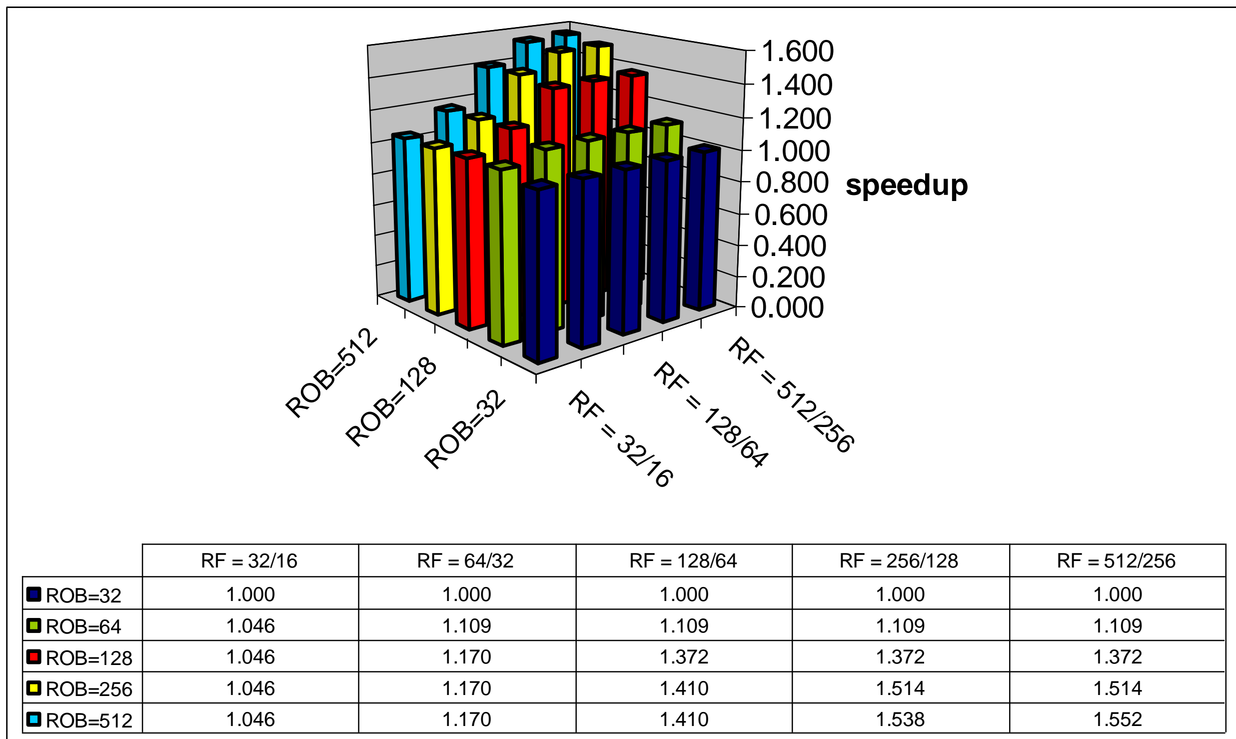
In the data above a somewhat counterintuitive trend is observed: the highest energy savings are achieved when we use the most conservative register file configurations (e.g., an ROB=512, RF=32/16 setup). In this case the RF becomes a severe bottleneck to performance allowing a large portion of the ROB to be gated off resulting in significant power savings.

To test that the ROB/RF configurations above are sensible and that we aren't simply increasing the ROB or RF beyond a size that provides some performance improvement, we examine the speedup normalized to a 32/16 integer/floating point register RF and a 32 entry ROB configuration. This data is plotted below for 2 and 8-issue machines on the Crafty benchmark. Similar speedups were observed for the other benchmarks (not shown).



	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	1.000	1.000	1.000	1.000	1.000
ROB=64	1.040	1.110	1.110	1.110	1.110
ROB=128	1.040	1.166	1.271	1.271	1.271
ROB=256	1.040	1.166	1.282	1.286	1.286
ROB=512	1.040	1.166	1.282	1.286	1.286

Figure 25 ROB and RF size impact on speedup for a 2-issue machine (Crafty)

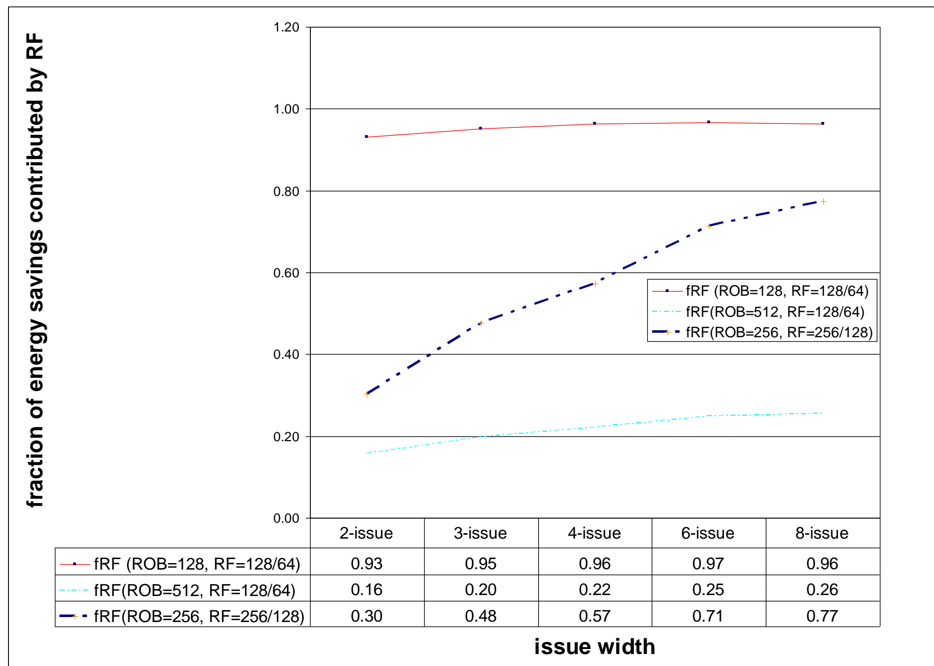


	RF = 32/16	RF = 64/32	RF = 128/64	RF = 256/128	RF = 512/256
ROB=32	1.000	1.000	1.000	1.000	1.000
ROB=64	1.046	1.109	1.109	1.109	1.109
ROB=128	1.046	1.170	1.372	1.372	1.372
ROB=256	1.046	1.170	1.410	1.514	1.514
ROB=512	1.046	1.170	1.410	1.538	1.552

Figure 26 ROB and RF size impact on speedup for an 8-issue machine (Crafty)

We observe a diminishing return trend for speedup beyond an ROB size of 256 and a RF size of 256/128 integer/floating point registers at the 8-issue design point. It is interesting to note that moving along the ROB and RF axes alone there are very minor speedups. It is only by increasing both the RF and ROB size that we achieve significant speedups. This makes sense because increasing a single resource will only make the other become a bottleneck as observed earlier. These results indicate that our processor design points are not artificially inflating the energy savings numbers by setting either the ROB or RF to an absurdly large value.

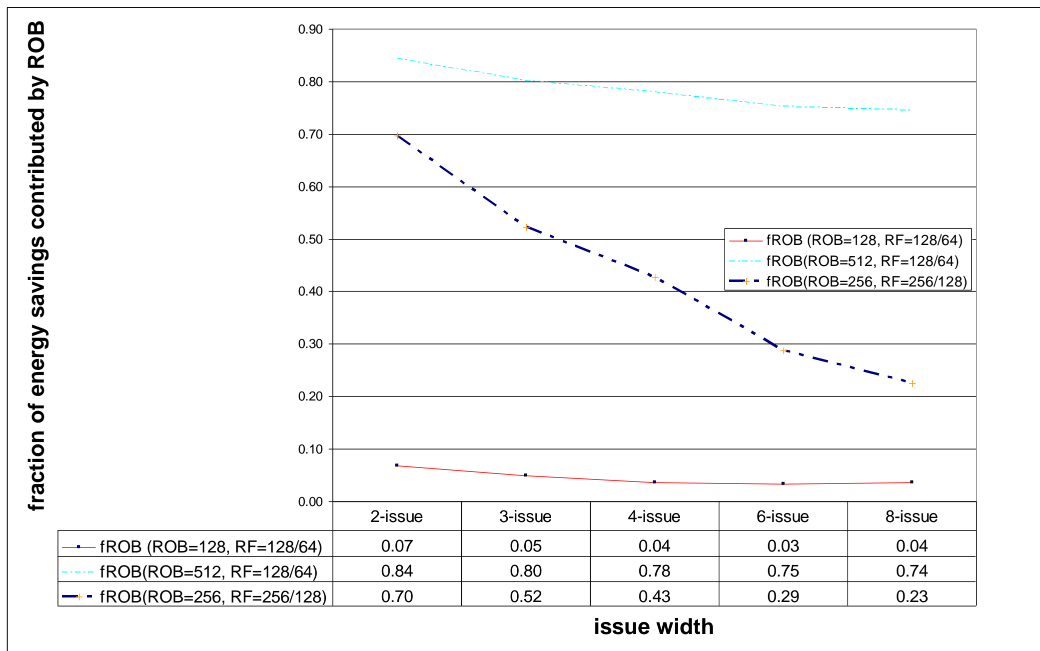
Since the above energy savings include both the reduction from the banked ROB and the banked RF, it is important to distinguish which fraction of the energy savings comes from each component. To determine this we run the Crafty benchmark at various issue widths with various ROB and RF sizes. This data is shown in the figure below.



	2-issue	3-issue	4-issue	6-issue	8-issue
fRF (ROB=128, RF=128/64)	0.93	0.95	0.96	0.97	0.96
fRF(ROB=512, RF=128/64)	0.16	0.20	0.22	0.25	0.26
fRF(ROB=256, RF=256/128)	0.30	0.48	0.57	0.71	0.77

Figure 27 Contribution of RF banking to energy savings

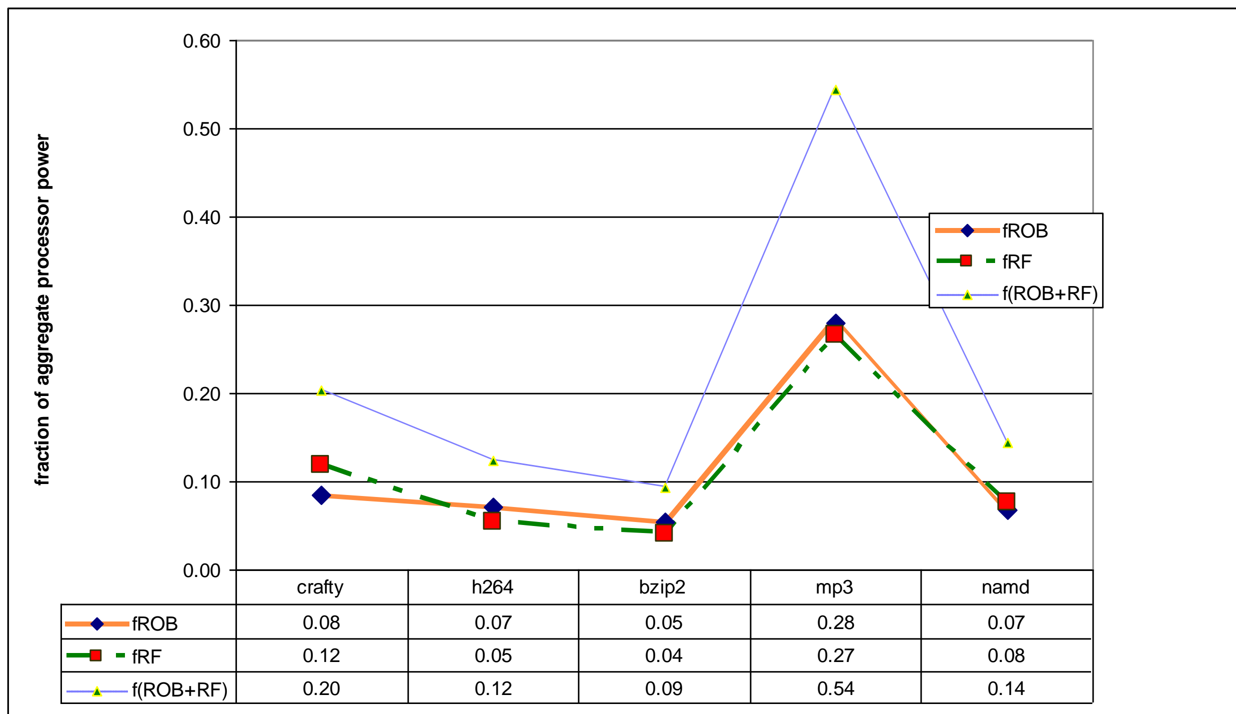
The ROB=128, RF=128/64 configuration's power savings are dominated by the energy reduction in the register file at all of the issue widths tested. This indicates the 128 entry ROB is under-provisioned to support a 128/64 register configuration. The ROB=512, RF=256/128 configuration's power savings are dominated by the energy reduction in the ROB at all issue widths. This indicates that the increase in ROB entries from 128 to 512 only results in a minor increase in usage of the ROB. Interestingly, for the ROB=256, RF=256/128 configuration, the energy reduction contribution of RF banking increases with issue width. This makes sense because in a many issue machine the ROB will be more heavily utilized and thus the benefit of banking it will be reduced. Similar data is shown below in Figure 28 for the contribution of banked ROB energy reduction. In subsequent sections in this chapter we will use a 3-issue ROB=256, RF=256/128 configuration because it splits the energy reduction contributed by the banked ROB and RF nearly in half.



	2-issue	3-issue	4-issue	6-issue	8-issue
fROB (ROB=128, RF=128/64)	0.07	0.05	0.04	0.03	0.04
fROB(ROB=512, RF=128/64)	0.84	0.80	0.78	0.75	0.74
fROB(ROB=256, RF=256/128)	0.70	0.52	0.43	0.29	0.23

Figure 28 Contribution of ROB banking to energy savings

Another concern is what fraction of the total processor energy is actually being used in the RF and ROB. This data is shown below for each of the benchmarks. We see that the ROB and RF power consumption varies dramatically depending upon the application's cache and branch predictor behavior. Because the mp3 benchmark consumes so much energy in the RF and ROB (over 50% for both combined) we would expect that a mechanism to reduce the power in those components would have the greatest impact on that benchmark. This is indeed exactly what we observe later.



	crafty	h264	bzip2	mp3	namd
fROB	0.08	0.07	0.05	0.28	0.07
fRF	0.12	0.05	0.04	0.27	0.08
f(ROB+RF)	0.20	0.12	0.09	0.54	0.14

Figure 29 Fraction of total processor power consumed in the (unbanked) RF and ROB for various benchmarks

The f(ROB+RF) line in the figure above demonstrates the absolute upper limit of the energy savings achievable using banking.

4.3. Bank Policy Impact

In this section we characterize the impact of bank turn-on, turn-off, and insertion policies on energy consumption and other metrics.

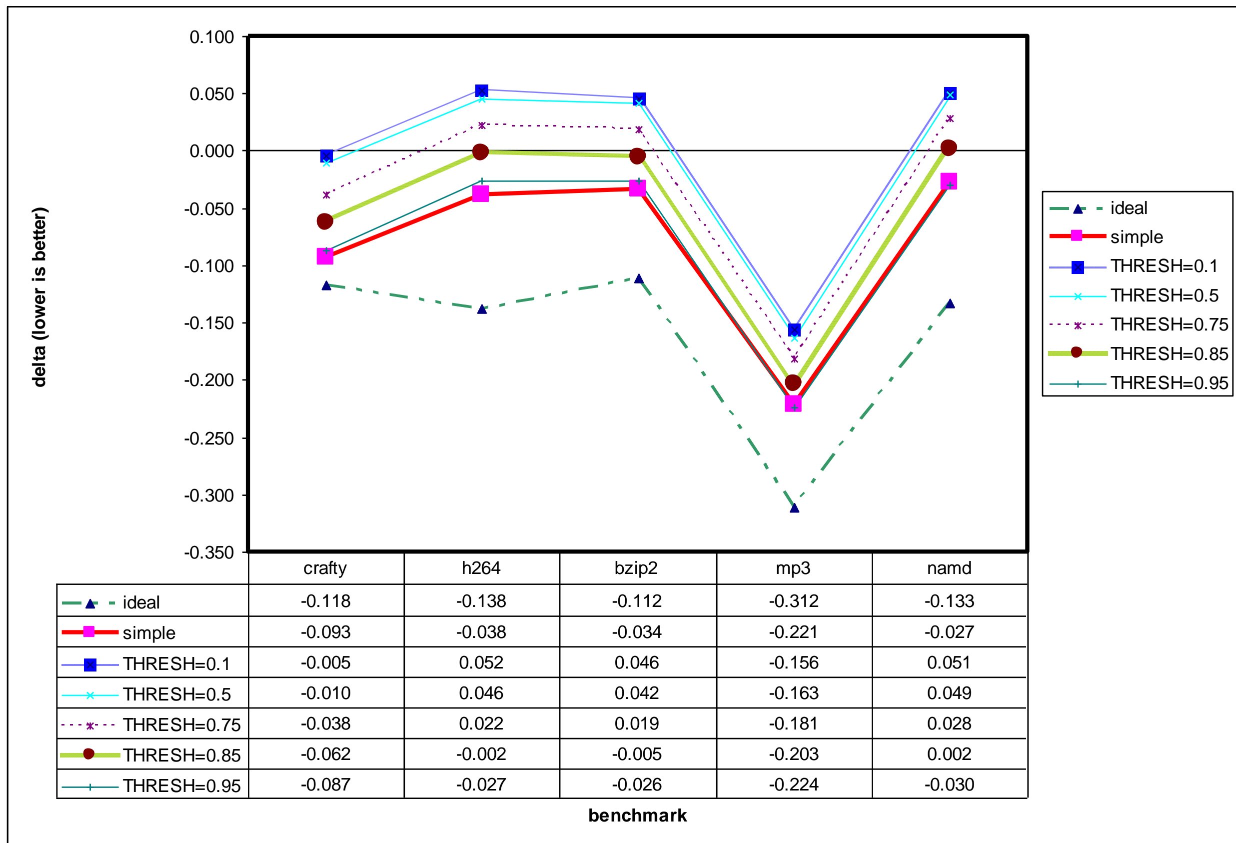
4.3.1 Bank Activation Policies

First we examine the impact of the proposed bank activation policies. To do so, we establish an optimistic configuration for the bank activation mechanism. This configuration is shown below.

Table 7 Optimistic banking configuration

Number of ROB, RF banks	8
ROB, RF bank activation policy	{simple, thresholded (α={0.1, 0.5, 0.75, 0.85, 0.95})}
ROB, RF bank deactivation policy	thresholded (γ=200)
RF insertion policy	FFE
Bank turn-on latency	8 cycles
Bank turn-off latency	2 cycles
Bank manager energy overhead	None

A thresholded deactivation policy with a large γ value was selected for the activation policy tests to maximize the impact of incorrectly turning on a bank. In the figure below we run the benchmarks on a 3-issue 256ROB 256/128RF configuration and plot the aggregate energy deltas compared to an unbanked configuration. It is important to note that because we are considering aggregate processor power in these figures the relevant comparison is relative to other variable settings.



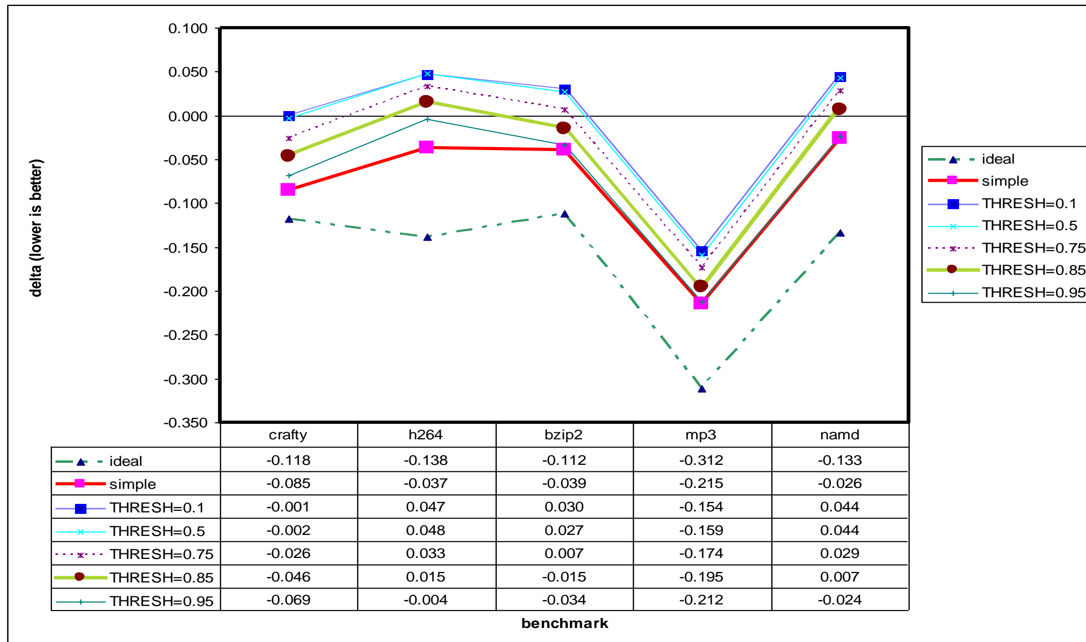
	crafty	h264	bzip2	mp3	namd
ideal	-0.118	-0.138	-0.112	-0.312	-0.133
simple	-0.093	-0.038	-0.034	-0.221	-0.027
THRESH=0.1	-0.005	0.052	0.046	-0.156	0.051
THRESH=0.5	-0.010	0.046	0.042	-0.163	0.049
THRESH=0.75	-0.038	0.022	0.019	-0.181	0.028
THRESH=0.85	-0.062	-0.002	-0.005	-0.203	0.002
THRESH=0.95	-0.087	-0.027	-0.026	-0.224	-0.030

Figure 30 Energy deltas at various activation policies compared to non-banked ROB/RF

As expected, the energy consumption increases as the activation threshold α decreases. This is because we wastefully precharge banks at low α values. Additionally, the behavior of the thresholded activation policy approaches the simple policy as α approaches 1, as it should since at that value there is no precharging.

It is important to note that pure energy consumption is no longer the whole story. Because banking can introduce stalls if there are no turned-on banks with free entries available, time must now be factored into the results presented earlier. Below we present similar data in the same format for delta energy-delay product (which accounts for the fact that doing something more slowly can be considered the same as doing it with less power) and energy-delay squared

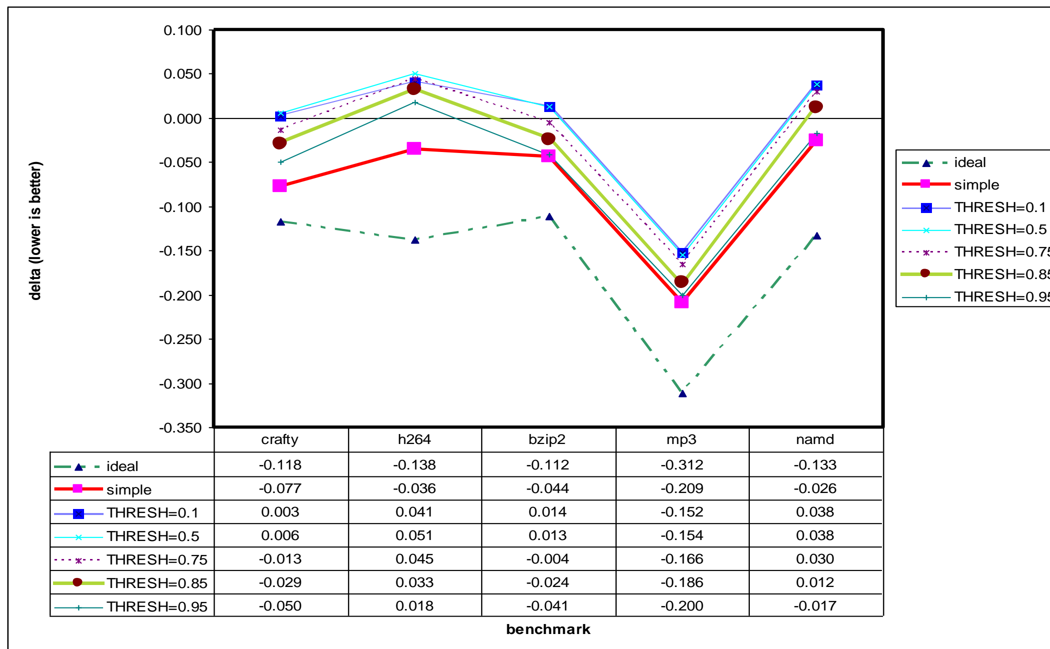
product (which accounts for the fact that doing something at a lower voltage reduces both power and performance) metrics. In the two figures below, lower values indicate better performance in terms of energy efficiency.



	crafty	h264	bzip2	mp3	namd
ideal	-0.118	-0.138	-0.112	-0.312	-0.133
simple	-0.085	-0.037	-0.039	-0.215	-0.026
THRESH=0.1	-0.001	0.047	0.030	-0.154	0.044
THRESH=0.5	-0.002	0.048	0.027	-0.159	0.044
THRESH=0.75	-0.026	0.033	0.007	-0.174	0.029
THRESH=0.85	-0.046	0.015	-0.015	-0.195	0.007
THRESH=0.95	-0.069	-0.004	-0.034	-0.212	-0.024

Figure 31 Energy-delay product deltas at various activation policies compared to non-banked ROB/RF

For each benchmark, the simple policy (which does not precharge banks) achieves significant improvement in terms of energy delay product. We see that reducing the precharging threshold (hence increasing precharge aggressiveness), however, can significantly degrade energy efficiency to the extent that the banked processor actually performs more poorly than an unbanked one. This indicates that with small bank turn-on delays precharging should be employed conservatively. We see a similar trend with the energy delay squared metric performance, shown below.



	crafty	h264	bzip2	mp3	namd
ideal	-0.118	-0.138	-0.112	-0.312	-0.133
simple	-0.077	-0.036	-0.044	-0.209	-0.026
THRESH=0.1	0.003	0.041	0.014	-0.152	0.038
THRESH=0.5	0.006	0.051	0.013	-0.154	0.038
THRESH=0.75	-0.013	0.045	-0.004	-0.166	0.030
THRESH=0.85	-0.029	0.033	-0.024	-0.186	0.012
THRESH=0.95	-0.050	0.018	-0.041	-0.200	-0.017

Figure 32 Energy, delay-squared product deltas at various activation policies compared to non-banked ROB/RF

Because we use small bank turn-on and turn-off latencies in this experiment, the slowdown caused by banking is similarly small. Thus the ET and ET^2 metrics track closely with the energy consumption data shown earlier.

In these experiments threshold activation doesn't seem to have much benefit compared to the simple activation policy. While precharging (threshold activation) did reduce execution time in all cases (for example, by up to 6% for the mp3 benchmark), it does so at a significant power cost which increases the overall energy consumption.

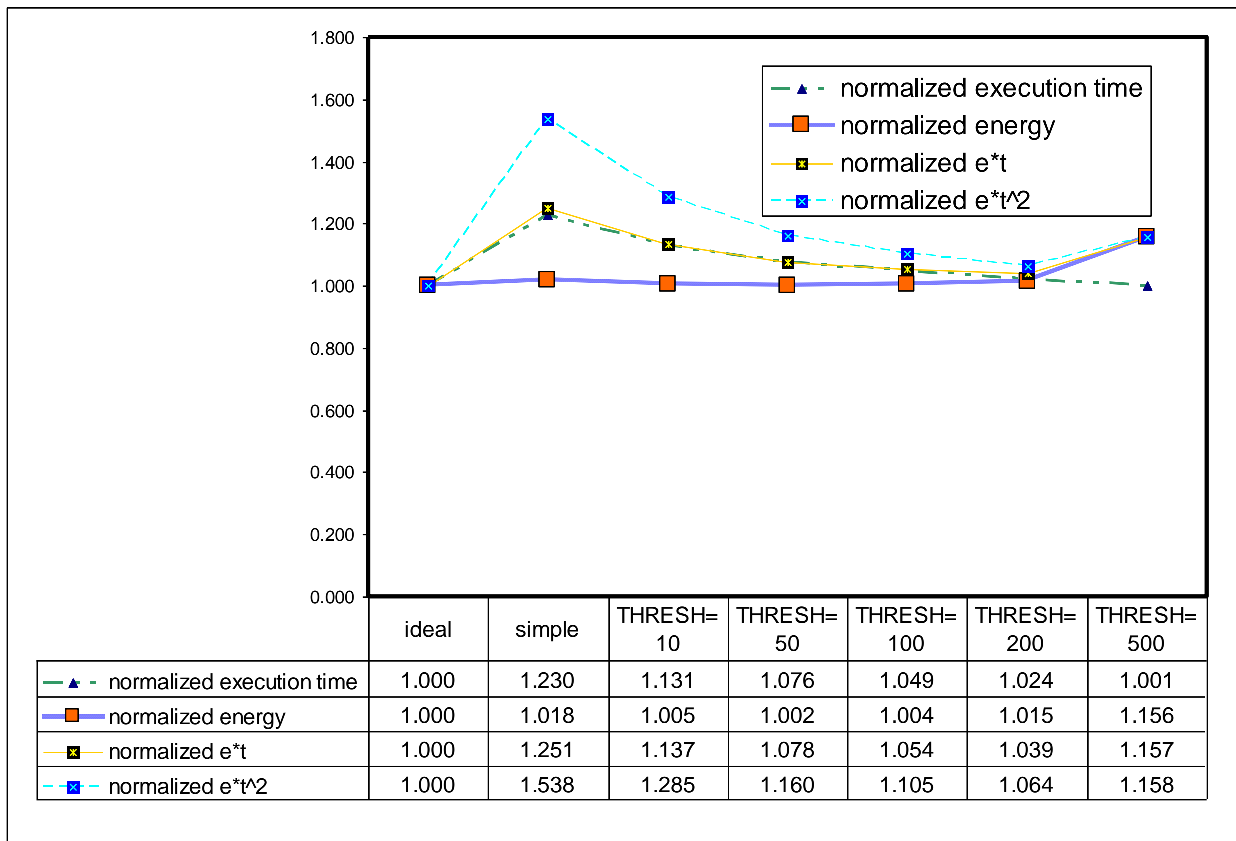
4.3.2 Bank Deactivation Policies

In this section we examine the impact of various bank turn-off policies. To extract meaningful data, we use the configuration shown below.

Table 8 Realistic banking configuration

Number of ROB, RF banks	8
ROB, RF bank activation policy	Simple
ROB, RF bank deactivation policy	{simple, thresholded (γ={10, 50, 100, 200, 500})}
RF insertion policy	FFE
Bank turn-on latency	100 cycles
Bank turn-off latency	2 cycles
Bank manager energy overhead	None

To maximize the impact of the deactivation policy we set the turn-on latency to be quite large: 100 cycles. Note that this is over five times the already conservative 20-cycle latency used in [19]. We then examine the impact of the deactivation threshold for the h264 benchmark which we noted to be one of the more difficult benchmarks for banking. This data is shown below normalized to the ideal case.



	ideal	simple	THRESH=10	THRESH=50	THRESH=100	THRESH=200	THRESH=500
normalized execution time	1.000	1.230	1.131	1.076	1.049	1.024	1.001
normalized energy	1.000	1.018	1.005	1.002	1.004	1.015	1.156
normalized e*t	1.000	1.251	1.137	1.078	1.054	1.039	1.157
normalized e*t^2	1.000	1.538	1.285	1.160	1.105	1.064	1.158

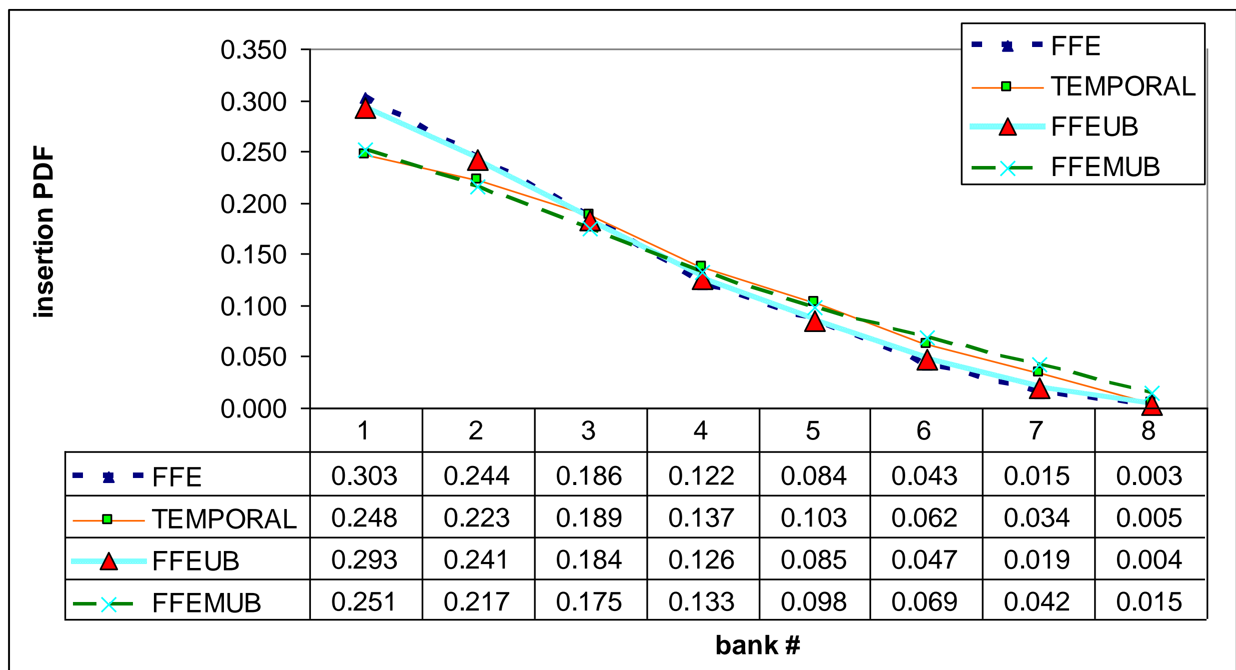
Figure 33 Deactivation policy energy and execution time impact

As expected, increasing the deactivation threshold monotonically decreases execution time because it reduces the number of banks that turn off prematurely and must then wait at least 100 cycles before they can be turned on again (since D_ON=100). Similarly, the energy consumed increases with the threshold value because banks spend more time powered on but unused before they can be turned off to save energy. As the threshold value increases, the energy consumption and execution time approach those of the unbanked configuration asymptotically.

4.3.3 Bank Insertion Policies

Perhaps the most interesting job of the bank controller is choosing the bank into which it should allocate (insert) demanded resources. In this section we examine the impact of various banking policies on the energy savings achievable using banking. Additionally we study the bank use history to determine how factors such as hotspotting might come into play.

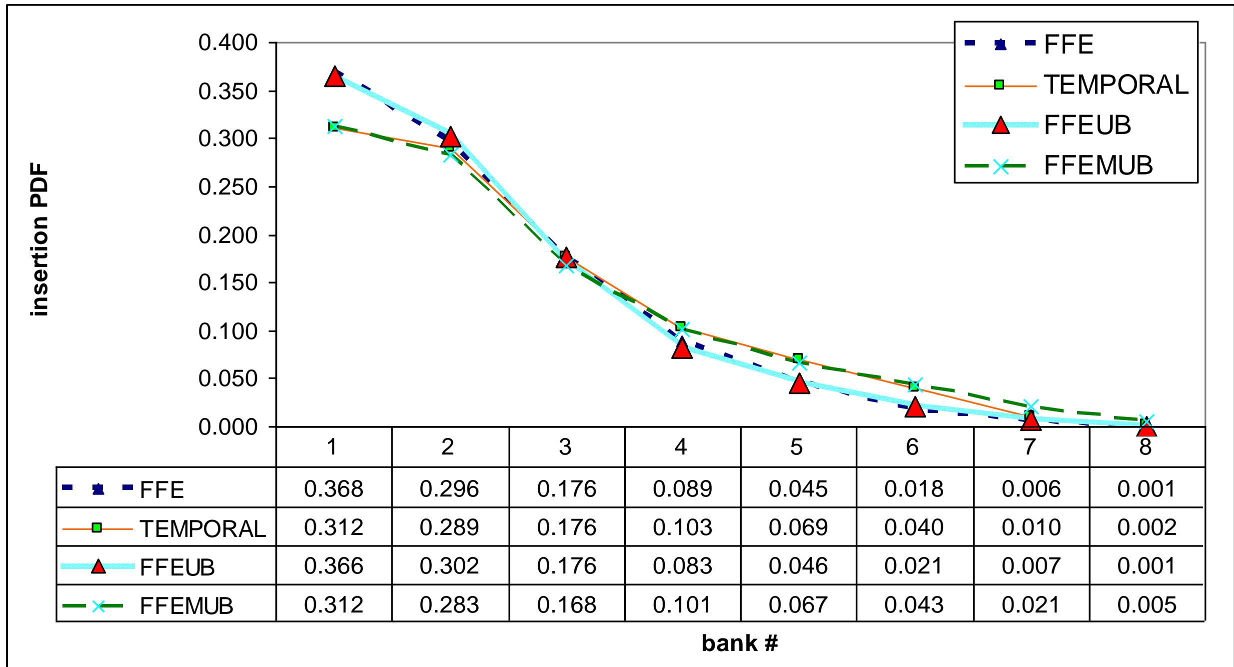
First we examine the distribution of insertions for the crafty benchmark. This data is plotted below. In each of the figures we plot the historical probability of inserting into a given bank over the entire execution of the benchmark for the various insertion policies, first with a small bank turn-on delay and then with a large bank turn-on delay to increase the differentiation between policies.



	1	2	3	4	5	6	7	8
FFE	0.303	0.244	0.186	0.122	0.084	0.043	0.015	0.003
TEMPORAL	0.248	0.223	0.189	0.137	0.103	0.062	0.034	0.005
FFEUB	0.293	0.241	0.184	0.126	0.085	0.047	0.019	0.004
FFEMUB	0.251	0.217	0.175	0.133	0.098	0.069	0.042	0.015

Figure 34 Bank insertion probability density functions for various policies (small bank latency)

With a small bank turn-on delay the policies appear similar in their placement patterns. The FFE and FFEUB policies are more likely to place elements into earlier banks because their search criteria are less restrictive than FFEMUB and TEMPORAL. FFEMUB and TEMPORAL insertion are far more likely than the others to place entries in later banks because they are less likely to consistently find banks satisfying their search criteria in earlier banks. To determine how bank latency affects the distribution of inserted entries we do the same experiment with a significantly larger bank turn-on delay. These results are shown below.



	1	2	3	4	5	6	7	8
FFE	0.368	0.296	0.176	0.089	0.045	0.018	0.006	0.001
TEMPORAL	0.312	0.289	0.176	0.103	0.069	0.040	0.010	0.002
FFEUB	0.366	0.302	0.176	0.083	0.046	0.021	0.007	0.001
FFEMUB	0.312	0.283	0.168	0.101	0.067	0.043	0.021	0.005

Figure 35 Bank insertion probability density functions for various policies (large bank latency)

With larger bank turn-on delays we observe a significantly higher probability of placement in the first several banks for all insertion policies. This is because later banks will begin warming up and subsequently shut down before they can be fully turned on due to the large D_ON delay. Another trend less obvious with small bank turn-on delays emerges: The FFE and FFEUB policies appear very similar in terms of their allocation distributions and FFEMUB and TEMPORAL appear to have very similar behavior. This makes intuitive sense because FFEUB is a fairly simple extension of the FFE policy. Similarly, TEMPORAL insertion is at its heart a mechanism for placing entries in frequently used banks. It appears that the added hardware complexity of implementing the TEMPORAL and FFEUB policies is not useful compared to the FFEMUB and FFE policies, respectively since their placement patterns are quite similar.

With a very large latency we also examine the behavior of FFELUB. These results are shown below.

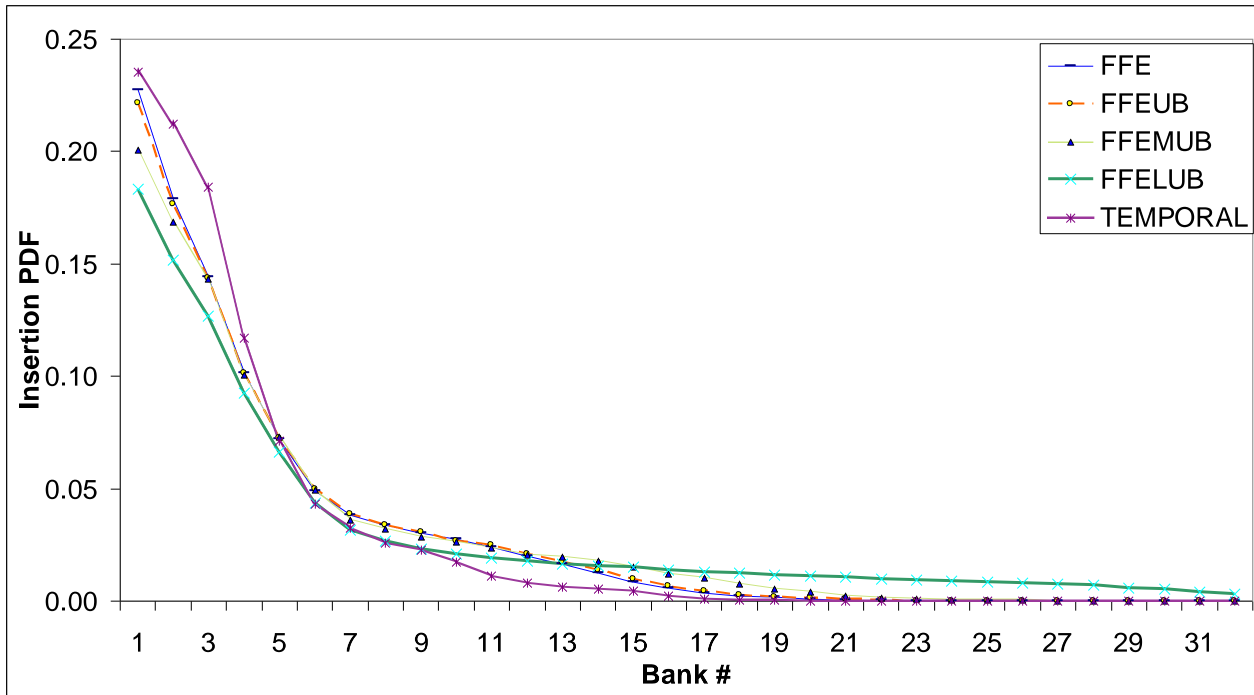
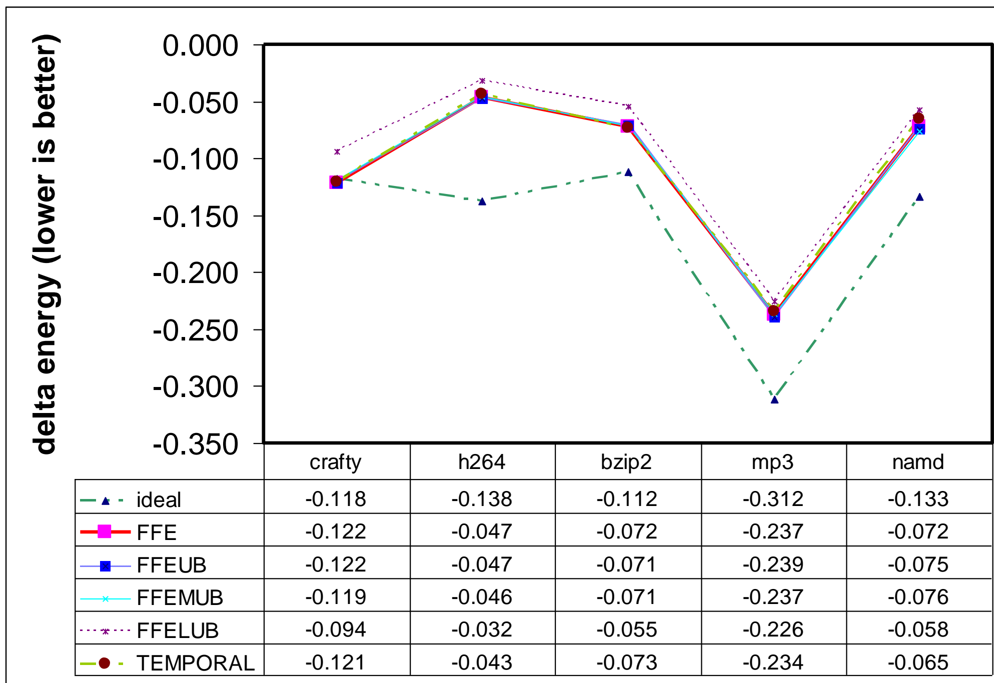


Figure 36 Bank insertion probability density functions for various policies (very large bank turn-on delay)

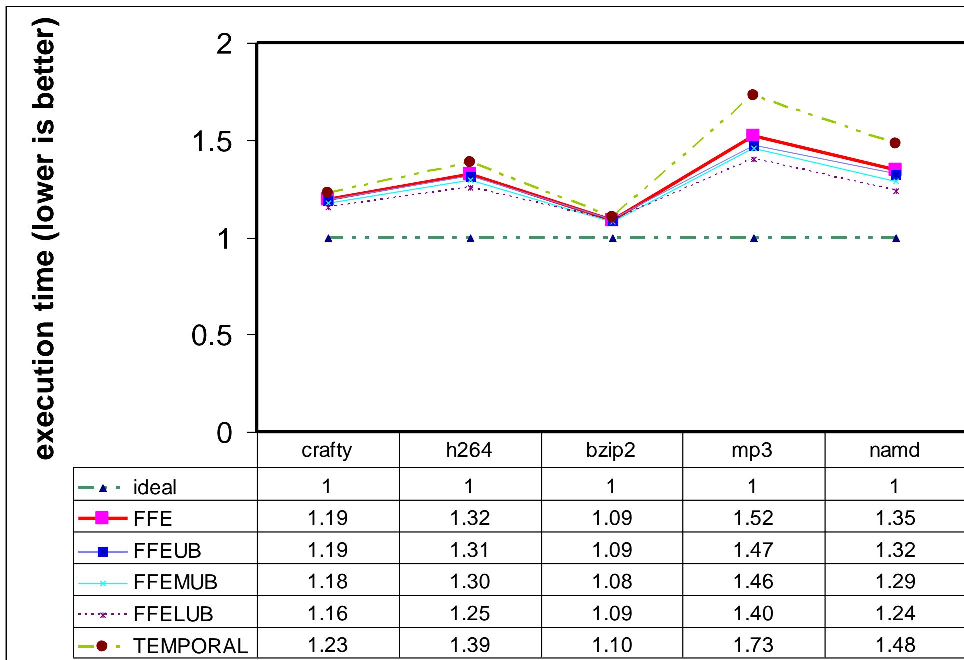
We see that FFELUB is much more likely than the other policies to place entries in later banks. This is because it distributes entries as much as possible, making it less likely to allow any single bank to be unused and subsequently turned off.

Next we examine the impact of insertion policy on aggregate processor energy consumption. This data is plotted below for the various benchmarks.



	crafty	h264	bzip2	mp3	namd
ideal	-0.118	-0.138	-0.112	-0.312	-0.133
FFE	-0.122	-0.047	-0.072	-0.237	-0.072
FFEUB	-0.122	-0.047	-0.071	-0.239	-0.075
FFEMUB	-0.119	-0.046	-0.071	-0.237	-0.076
FFELUB	-0.094	-0.032	-0.055	-0.226	-0.058
TEMPORAL	-0.121	-0.043	-0.073	-0.234	-0.065

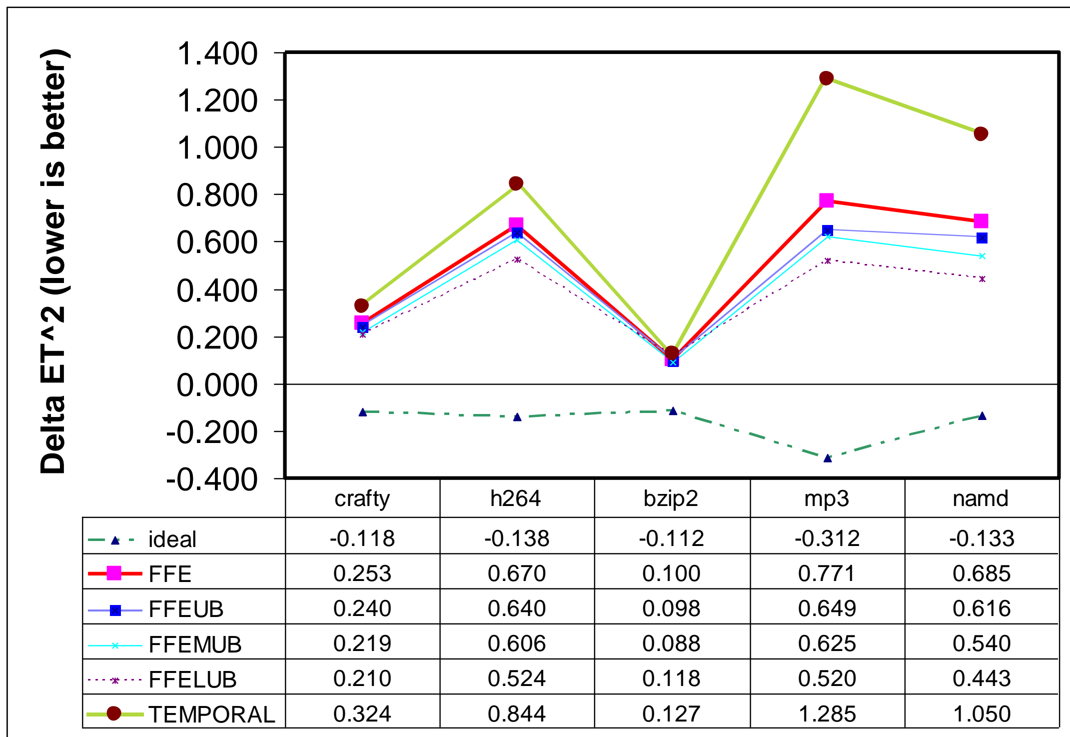
Figure 37 Delta energy by benchmark for various insertion policies



	crafty	h264	bzip2	mp3	namd
ideal	1	1	1	1	1
FFE	1.19	1.32	1.09	1.52	1.35
FFEUB	1.19	1.31	1.09	1.47	1.32
FFEMUB	1.18	1.30	1.08	1.46	1.29
FFELUB	1.16	1.25	1.09	1.40	1.24
TEMPORAL	1.23	1.39	1.10	1.73	1.48

Figure 38 Insertion policy impact on execution time

Surprisingly we observe that all of the insertion policies tested result in nearly identical energy consumption with the exception of FFELUB, which consumes the most energy as expected. There are minor variations in energy consumption between the others, but none significant. The execution times of the same runs are also shown above. Again, with very large bank turn-on delays (in this case we use 512 cycles) FFELUB performs significantly better than the other policies. We used a large 512-cycle turn-on latency in this section to maximize differentiation between the policies since with small bank turn-on delays we observed little differentiation in terms of energy efficiency, but is important to note that this is quite an unrealistic design point since with 512 cycles of turn-on delay the energy efficiency in terms of ET^2 is significantly worse than an unbanked processor as shown below.



	crafty	h264	bzip2	mp3	namd
ideal	-0.118	-0.138	-0.112	-0.312	-0.133
FFE	0.253	0.670	0.100	0.771	0.685
FFEUB	0.240	0.640	0.098	0.649	0.616
FFEMUB	0.219	0.606	0.088	0.625	0.540
FFELUB	0.210	0.524	0.118	0.520	0.443
TEMPORAL	0.324	0.844	0.127	1.285	1.050

Figure 39 Impact of insertion policy on energy efficiency (ET^2) with very large bank turn-on delay (512 cycles)

While the results above are important in that they verify our intuition that more complex policies such as FFELUB can significantly outperform simpler policies like FFE when bank turn-on latencies are very large, we observed no significant difference (less than 1% between the best and worse performing policy across all benchmarks) between the FFE, FFEUB, FFEMUB, and FFELUB insertion policies with delays smaller than 128 cycles. In all cases, however, TEMPORAL was the worst performer in terms of energy consumed, energy delay product, and energy delay squared product.

In subsequent tests we will use FFE since it is the simplest insertion policy and achieves energy efficiencies similar to more complex policies such as FFEUB, FFEMUB, and FFELUB for bank delays less than 128 cycles.

4.4. Impact of Bank Latencies

In this section we examine the impact of the bank turn-off and bank turn-on latencies. As we have discussed, bank latencies are a critical design parameter and the D_ON and D_OFF delays determine the effectiveness of banking arguably more than any other design constraint. In [9], bank latencies of 6-20 cycles are examined. In this work we consider turn-on latencies ranging from 0 to 128 cycles (up to 128ns at 1GHz) as shown below. During testing we use the 3-issue 256ROB 256/128RF processor configuration and the crafty and h264 benchmarks because of the difficulty they present for achieving good banking performance.

Table 9 Bank latency test configuration

Parameter	Value
Number of ROB, RF banks	8
ROB, RF bank activation policy	Simple
ROB, RF bank deactivation policy	Simple
RF insertion policy	FFE
Bank turn-on latency	{0-128} cycles

Bank turn-off latency	FLOOR([D_ON/4])
Bank manager energy overhead	None

Below are the results of these runs for h264 and crafty. We see that crafty is considerably less sensitive to bank latencies than h264. Interestingly, bank latency has very little impact on the total energy consumed during execution for either benchmark. This indicates that increasing the bank latencies simply causes the processor to consume less power for more time. However, if we examine the ET^N metrics which account for the speed of execution we observe a significant degradation beyond a turn-on latency of even 8 cycles (8 ns). Overcoming this degradation for large latencies is a significant challenge. Note that in the figures below the full configuration corresponds to an unbanked processor.

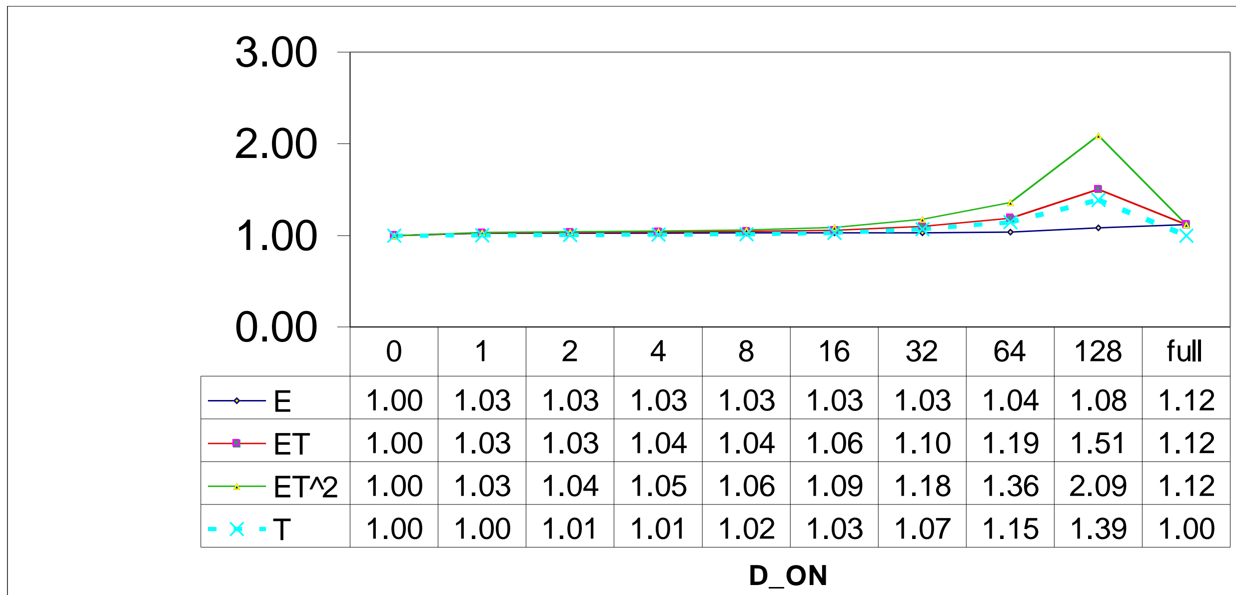
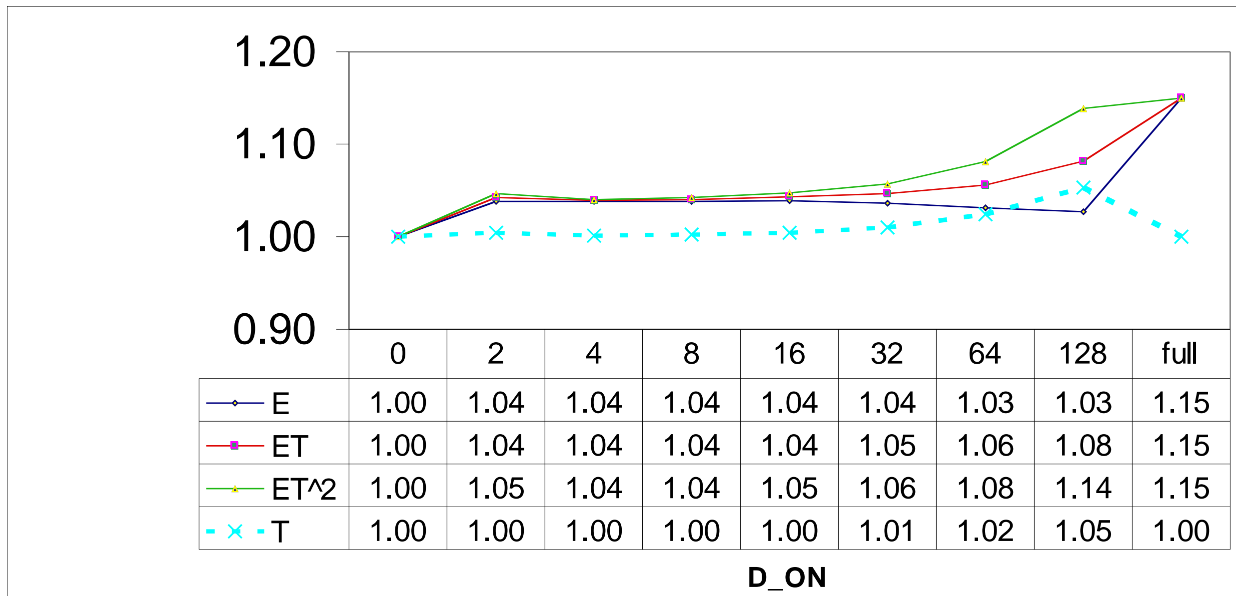


Figure 40 Impact of the D_ON (and D_OFF) latencies on banking performance in terms of E, ET, and ET^2 for the h264 benchmark normalized to the banked, zero-delay case (h264)



	0	2	4	8	16	32	64	128	full
E	1.00	1.04	1.04	1.04	1.04	1.04	1.03	1.03	1.15
ET	1.00	1.04	1.04	1.04	1.04	1.05	1.06	1.08	1.15
ET^2	1.00	1.05	1.04	1.04	1.05	1.06	1.08	1.14	1.15
T	1.00	1.00	1.00	1.00	1.00	1.01	1.02	1.05	1.00

Figure 41 Impact of the D_ON (and D_OFF) latencies on banking performance in terms of E, ET, and ET^2 for the crafty benchmark normalized to the banked zero-delay case (crafty)

4.5. Impact of Bank Manager Energy Consumption

In this section we factor in the energy consumed by the bank manager for the RF and ROB structures. We simulate three points for the bank manager energy consumption: conservative, which is an upper bound; optimistic, which is a middle-of-the-road estimate, and ideal, which assumes the bank manager consumes no energy. For the conservative estimate we use the area overheads provided in Figure 15. For the optimistic estimate we use half the overhead of the conservative estimate. For the ideal case we simply assume the bank manager consumes zero power at all times. We ran simulations with the ROB=256, RF=256/128 configuration for a 3-issue machine on the crafty benchmark with various numbers of banks and present the results of these runs below.

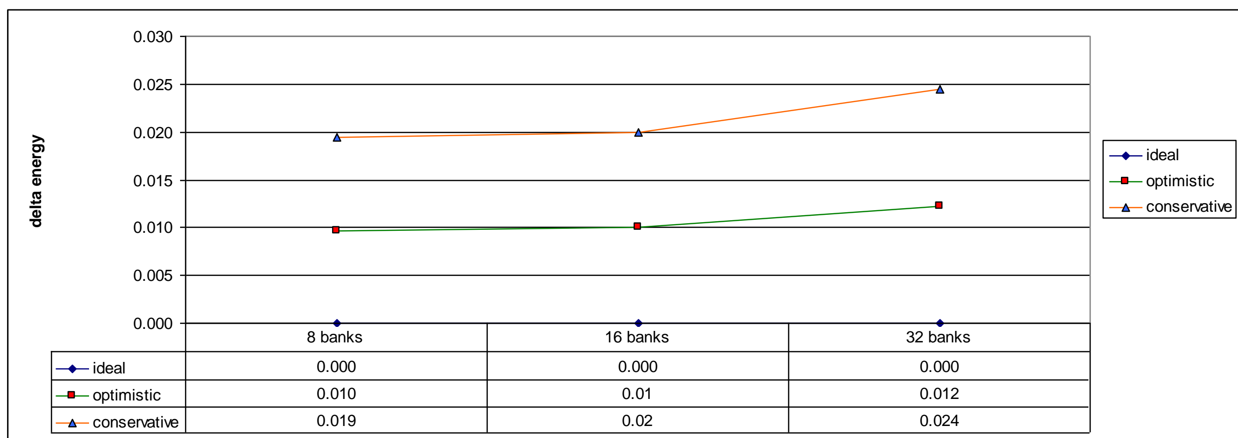


Figure 42 bank energy manager power consumption impact

The energy consumed by the bank manager ranges from 1 to nearly 2.5% depending upon which estimate is used and how many banks are utilized. Based upon the results obtained earlier there is not much benefit in moving from 8 to 16 or even 32 banks so we will use 8 banks in subsequent tests. Additionally, we have seen that there is not a significant performance gain from using a more power hungry banking policy like TEMPORAL compared to even the simplest policy, FFE, so we will use FFE in subsequent tests. At these design points we would incur roughly a 1% aggregate processor energy overhead in the bank manager for crafty.

4.6. Testing a realistic low-power banked processor

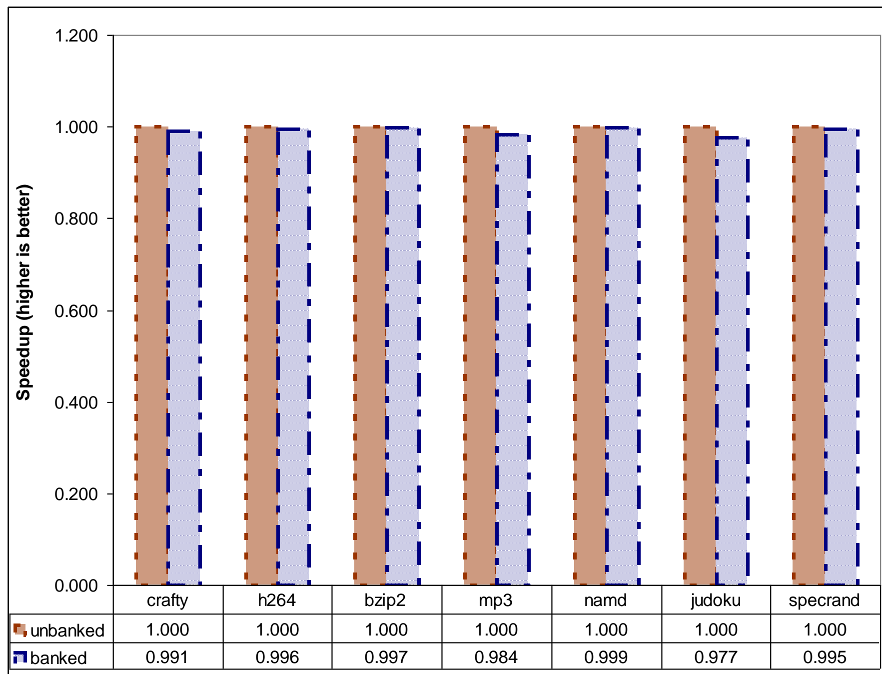
In previous sections we have characterized the impact of various policies on energy overhead, bank utilization and other metrics. In this section we will use these observations to select policies for a banked processor which is potentially capable of achieving significant energy savings. We test this processor at a feasible design point (the 3-issue, 256ROB 256/128RF configuration described earlier) and include the effects of stabilization latencies and bank manager energy consumption as shown in the table below.

Table 10 Realistic banking configuration

Number of ROB, RF banks	8
ROB, RF bank activation policy	Simple
ROB, RF bank deactivation policy	thresholded (γ=200)
RF insertion policy	FFE
Bank turn-on latency	64 cycles (64ns @1GHz)
Bank turn-off latency	4 cycles (4ns @1GHz)
Bank manager energy overhead	optimistic (0.035 @ 256ROB, 256/128RF)

FFE was selected because it is effective, simple, and has a small implementation overhead. We model the RF and ROB to consume 3.5% additional energy to simulate the energy required to enforce the FFE policy. We select a rather large deactivation threshold (γ =200) to reduce bank state oscillations and increase the likelihood that when a bank is turned off it will not be turned on shortly thereafter. We select rather large turn bank latencies to be as conservative as possible (indeed, we are significantly more conservative than [19]). The simple activation policy was selected because threshold activation was observed to too frequently precharge banks without providing much performance gain at bank latencies below two hundred cycles.

First we examine the banking performance cost in terms of execution time. Plotted below are the speedups for each of the benchmarks normalized to the unbanked case. Note that we include two previously untested benchmarks, judoku (int) and specrand (floating point), to ensure that any banking improvements are not artifacts of benchmark-specific optimization.

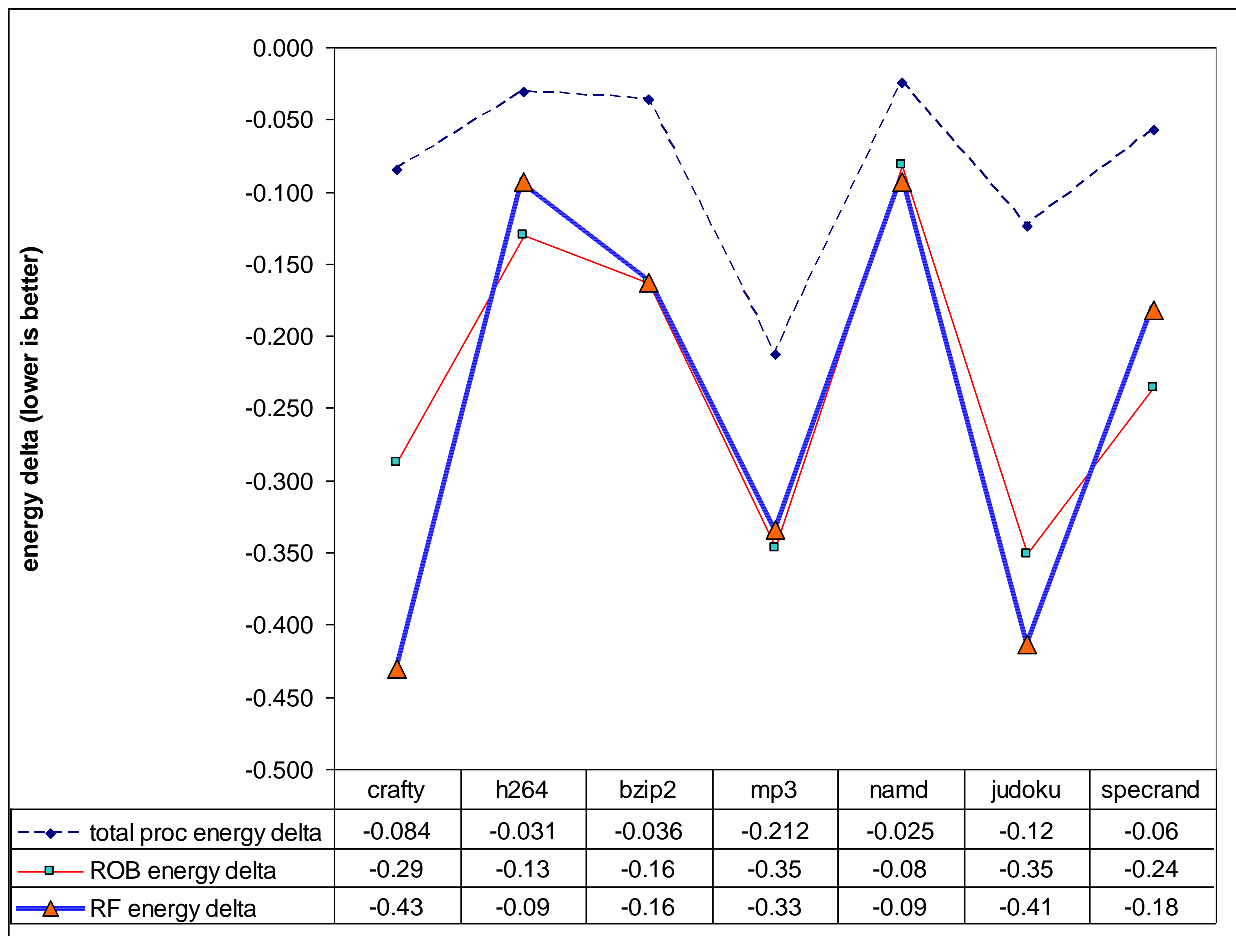


	crafty	h264	bzip2	mp3	namd	judoku	specrand
unbanked	1.000	1.000	1.000	1.000	1.000	1.000	1.000
banked	0.991	0.996	0.997	0.984	0.999	0.977	0.995

Figure 43 Banking performance cost (execution time)

We see that banking causes a very minor degradation in performance (0.1% to 2.3%) for the benchmarks studied. In all cases this reduction in speed is more than offset by the reduction in energy, as indicated by the energy-delay product metrics shown in Figure 45.

Next we examine the effect of banking on processor energy consumption. Plotted below for each of the benchmarks is the impact of banking on energy consumption for the entire processor as well as the ROB and RF components individually.



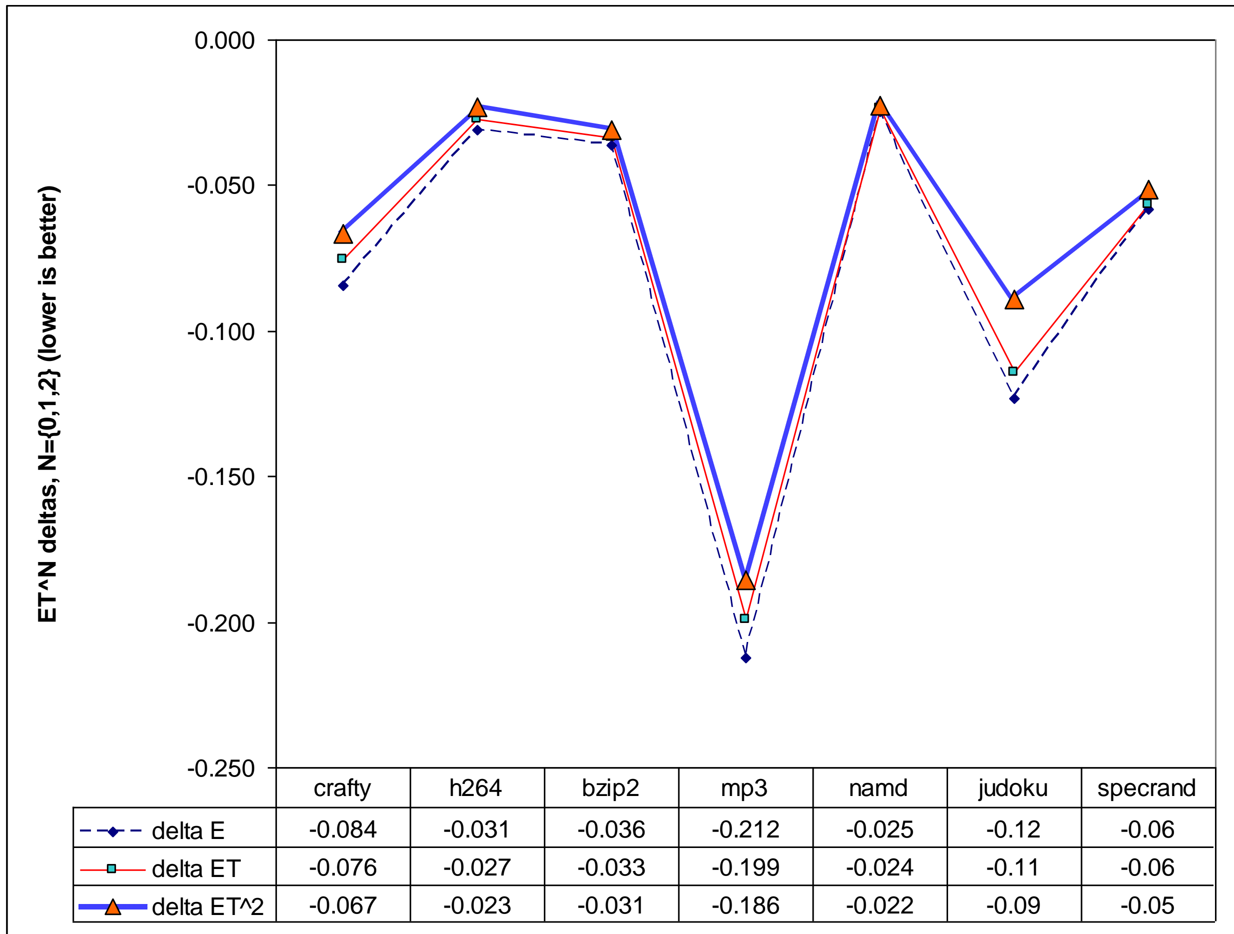
	crafty	h264	bzip2	mp3	namd	judoku	specrand
total proc energy delta	-0.084	-0.031	-0.036	-0.212	-0.025	-0.12	-0.06
ROB energy delta	-0.29	-0.13	-0.16	-0.35	-0.08	-0.35	-0.24
RF energy delta	-0.43	-0.09	-0.16	-0.33	-0.09	-0.41	-0.18

Figure 44 Banking performance with a realistic configuration accounting for bank latencies and bank manager power overhead

Banking reduces the energy consumption of the RF and ROB structures across all benchmarks tested, though the actual energy reduction varies significantly between benchmarks. MP3 is the best performing in terms of the energy reduction achieved for the entire processor, with a decrease of nearly 21% in total energy consumed. The crafty, judoku, and specrand benchmarks also achieve significant total processor energy savings (8.4%, 12% and 6%, respectively) with similar reductions in ROB and RF energy consumption. We see less of an overall processor energy reduction in a benchmark like crafty compared to mp3 because for

crafty the fraction of total energy consumed by the RF and ROB structures combined is 20% whereas in mp3 these structures consume 54% of the aggregate processor energy.

Plotted below are the deltas in the aggregate energy, energy delay product, and energy, delay-squared product metrics for the same configuration.



	crafty	h264	bzip2	mp3	namd	judoku	specrand
delta E	-0.084	-0.031	-0.036	-0.212	-0.025	-0.12	-0.06
delta ET	-0.076	-0.027	-0.033	-0.199	-0.024	-0.11	-0.06
delta ET^2	-0.067	-0.023	-0.031	-0.186	-0.022	-0.09	-0.05

Figure 45 Impact of banking on aggregate processor E, ET, and ET^2 metrics

4.7. Testing a hypothetical high-power banked processor

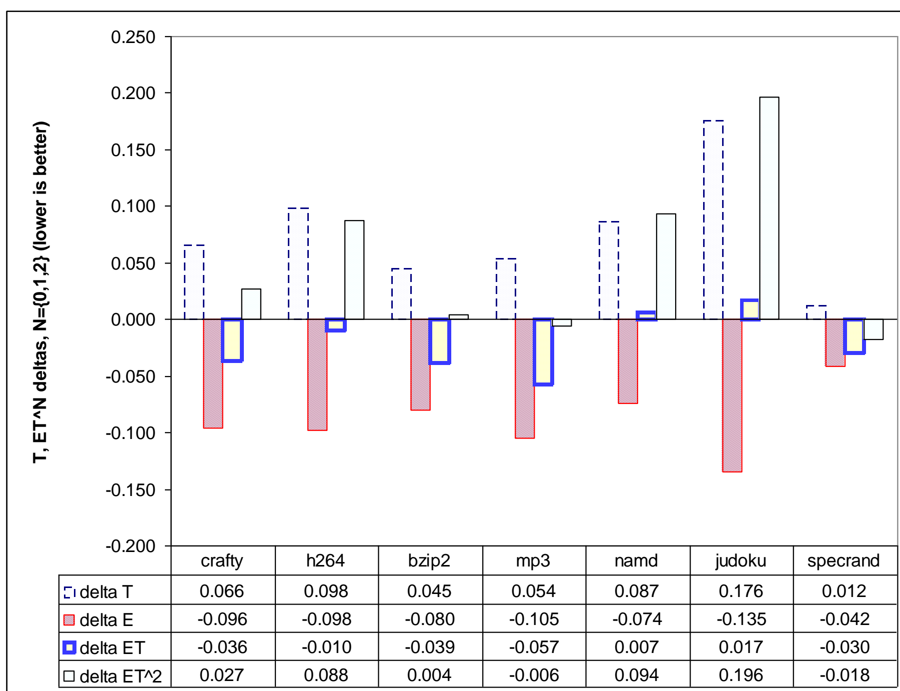
In this section we attempt to determine how effective banking would be for the considerably more aggressive processor configuration shown below.

Table 11 Aggressive core configuration

Issue width, pipeline depth, retire width	8
Fetch width	([issue]/6+1)*6
IQ size	2*[fetch width]
ROB entries	256
Integer registers	256
Floating point registers	[iregs]/2
Clock frequency	4GHz

Note that increasing the clock frequency by a factor of 4 effectively quadruples the number of cycles in the D_ON and D_OFF bank delays to 256 and 16 cycles respectively. To reduce the impact of these large delays we use a threshold bank activation policy (α=0.95). Aside from these modifications and those described in the table above, we leave the remainder of the processor and memory parameters the same.

Plotted below for each of the benchmarks is the impact of banking on execution time, energy consumption, energy-delay product, and energy delay-squared product for the aggressive processor.



	crafty	h264	bzip2	mp3	namd	judoku	specrand
delta T	0.066	0.098	0.045	0.054	0.087	0.176	0.012
delta E	-0.096	-0.098	-0.080	-0.105	-0.074	-0.135	-0.042
delta ET	-0.036	-0.010	-0.039	-0.057	0.007	0.017	-0.030
delta ET^2	0.027	0.088	0.004	-0.006	0.094	0.196	-0.018

Figure 46 Banking impact on execution speed, energy, energy-delay product and energy delay-squared product for an aggressive processor design

Interestingly, we observe better energy reductions for the aggressive processor case. This reduction in energy comes at the high cost of significantly reduced performance, however, and the energy delay product and energy delay squared product metrics suffer because of it. The significant reduction in performance observed is a result of the dramatic increase in bank latencies caused by the four-fold increase in clock speed. Banking is a technique which is clearly limited by these bank latencies and is thus a technique more applicable to the lower end of the processor aggressiveness design spectrum, at least if implemented in the manner proposed in this work.

4.8. Future Work

The experiments in this thesis lay the groundwork for a wide body of future research. Perhaps the most obvious extension of this work would be the continuing search for better insertion and bank turn-on/off policies, particularly those that reduce bank port conflicts to improve the effectiveness of a port reduction scheme like [21]. We found that the bank insertion policies, while theoretically interesting and capable of shaping the bank insertion affinity curves, were not significantly different in terms of performance. In the context of an optimistically ported banking scheme, however, the insertion policies would play a critical role and more exotic policies (such as anti-temporal) might achieve significantly better results than FFE.

Another avenue of research this work inspires concerns the concept of bank defragmentation. During execution the banks inevitably begin to fill with holes (entries which are not used) as individual entries in a bank retire before other entries in the bank. The increasingly complex insertion policies studied were an attempt to minimize this effect, but a simpler and possibly more energy efficient way of doing this would be to periodically re-rename entries (i.e., defragment them) to maximize the utilization of banks which are turned on at any given time.

One problem with the banking policies we have studied is that individual banks tend to be used significantly more than others, introducing the problem of hotspotting. Finding a policy that minimizes hotspotting without reducing performance noticeably would be an interesting task.

CHAPTER 5. CONCLUSIONS

We have demonstrated that voltage gating banks of entries in the RF and ROB structures in a modern out-of-order processor is an effective method of reducing the energy it consumes during execution provided its bank turn-on latencies are less than 64 clock cycles. Across a wide range of benchmarks we observed significant reductions in RF energy consumption (6 to 40%), ROB energy consumption (5 to 35%) and aggregate processor energy consumption (2 to 21%) due to banking. These energy improvements came at such a small cost in terms of execution time overhead (0.1 to 2.3%) that the energy-delay product and energy, delay-squared product for all of the benchmarks tested was improved by 2% to 19%.

Of the bank deactivation policies tested, we found threshold deactivation to be the most useful because it reduces the likelihood that a turned-off bank will be needed in the immediate future. Of the bank activation policies tested, we found simple activation to work well with small bank turn-on delays and threshold activation to work better with long bank turn-on delays because it preemptively turns on banks when most of the resources in currently turned-on banks are utilized. We found that of the insertion policies tested, FFE provided the most significant energy savings and FFELUB worked best for large bank latencies.

Banking and voltage gating banks of resources as described in this work is a promising technique in low-power processor design. It has the benefit of low complexity in terms of implementation, small area overhead and does not prohibitively affect performance. Additionally, banking has the potential to provide additive energy savings with existing energy efficient techniques such as conservative porting.

REFERENCES

[1] C. Meenderinck and B. Juurlink, "(When) Will CMPs Hit the Power Wall?" in *Proc. Workshop on Highly Parallel Processing on a Chip*, 2008.

[2] G. Venkatesh et al, "Conservation Cores: Reducing the Energy of Mature Computations," in *ASPLOS*, 2010.

[3] G. Moore, "Lithography and the Future of Moore's Law," in the *Proceedings of SPIE*, vol. 2437, pp. 2-17, 1995.

[4] R. R. Schaller, "Moore's Law: Past, Present and Future," in *IEEE Spectrum*, June 1997.

[5] .F. Fallah and M. Pedram, "Standby and Active Leakage Current Control and Minimization in CMOS VLSI Circuits," in *IEICE Transactions on Electronics*, Vol. E88-C, No. 4, 2005, pp. 509-519.

[6] N. Kim, T. Austin et al., "Leakage Current: Moore's Law Meets Static Power," in *IEEE Transactions on Computers*, vol. 36, no. 12, pp. 68-75, 2003.

[7] Battery life ref

[8] "The ARM Cortex-A9 Processors." Retrieved 06.12.11 from http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf

[9] "Mobile Intel Atom Processor N270 Single Core Datasheet," May 2008. Retrieved 06.12.11 from http://download.intel.com/design/processor/datashts/320032.pdf

[10] J. Cruz, A. Gonzalez, et al, "Multiple-Banked Register File Architectures," in *ISCA 27*, 2000.

[11] J. Tseng, K. Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors," in *ISCA 30*, 2003.

[12] R. Kessler, "The Alpha 21264 Microprocessor," in *IEEE Micro*, 1999.

[13] John L. Hennessy, David A. Patterson, Computer Architecture A Quantitative Approach, Elsevier, Inc, 2007.

[14] R. Sangireddy, "Reducing Rename Logic Complexity for High-Speed and Low-Power Front-End Architectures," in *IEEE Transactions on Computers*, 2006.

[15] G. Kucuk et al, "Distributed Reorder Buffer Schemes for Low Power," in *ISCA*, 2003.

[16] Q. Wu et al, "Clock Gating and Its Application to Low Power Design of Sequential Circuits," in *IEEE Custom IC Conference*, 1997.

[17] A. Chandrakasan, R. Brodersen, "Low Power Digital CMOS Design," Kluwer Academic Publishers, 1995.

[18] Z. Hu et al, "Microarchitectural Techniques for Power Gating of Execution Units," in the *Proceedings of ISLPED04*, pp. 32-37, 2004.

[19] H. Homayoun et al, "Functional Unit Power Gating in SMT Processors," in *PACRIM*, 2005.

[20] J. Martinez et al, "Dynamic Multicore Resource Management: A Machine Learning Approach," in *IEEE Micro*, 2009.

[21] J Tseng et al, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors," in *SIGARCH*, 2003.

[22] G. Kucuk et al, "Distributed Reorder Buffer Schemes for Low Power," in *ICCD03*, 2003.

[23] G. Kucuk et al, "Low Complexity Reorder Buffer Architecture," in *ICS*, 2002.

[24] D. Ponomarev et al, "Energy Efficient Design of the Reorder Buffer," in *Lecture Notes in Computer Science*, Volume 2451, pp. 219-227, 2002.

[25] P. Ortego, P. Sack, "SESC: Super ESCalar Simulator," in *Tech Report*, December 2004.

[26] D. Brooks et al, "Wattch: A Framework for Architectural Level Power Analysis and Optimizations," in *ISCA 27*, 2000.

[27] P. Shivakumar, N. Jouppi, "Cacti 3.0: An Integrated Cache Timing, Power and Area Model," Technical report, Compaq Western Research Laboratory, Feb. 2001.

[28] E. Perelman et al, "Using SimPoint for Accurate and Efficient Simulation," ACM Sigmetrics Performance Evaluation Review, 2003.