LEAKAGE ENERGY OPTIMIZATIONS AND THEIR IMPLICATIONS

A Thesis in
Computer Science and Engineering
by
Wei Zhang

© 2003 Wei Zhang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2003
We approve the thesis of Wei Zhang.

Date of Signature

______________________________
Mahmut Taylan Kandemir
Assistant Professor of Computer Science and Engineering
Thesis Adviser
Chair of Committee

______________________________
Mary Jane Irwin
Distinguished Professor of Computer Science and Engineering

______________________________
Anand Sivasubramaniam
Associate Professor of Computer Science

______________________________
Vijaykrishnan Narayanan
Assistant Professor of Computer Science and Engineering

______________________________
Richard R. Brooks
Research Associate of Applied Research Laboratory

______________________________
Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering
Abstract

Power and energy consumption has become a significant constraint in modern microprocessor design. While energy conscious design is obviously crucial for battery driven mobile and embedded systems, it has also become important for desktops and servers due to packaging and cooling requirements where power consumption has grown from a few watts per chip to over 100 watts. Recent trends indicate that the architecture and software techniques, in addition to low-level circuit techniques, can play a major role in power-efficient computer system design.

VLIW architectures are increasingly used in systems where energy-efficiency is an important consideration. In a VLIW architecture, compiler plays an important role in achieving acceptable instruction level parallelism. The energy efficiency of VLIW processors is determined by the underlying hardware and compiler techniques. While many performance optimizations in VLIW architecture also bring energy benefits as a side effect, it is possible to achieve further benefits by explicitly focusing on energy. This thesis proposes three such compiler-directed energy optimizations for multiple-issue architectures.

The first optimization is based on the observation that “slacks” present in the CPU data-path in VLIW machines due to the lack of available independent instructions, data dependence and schedule-specific decisions. Therefore, an optimizing compiler can analyze the program to identify the slack for each instruction. To exploit slacks, we can scale down voltage/frequency or turn off functional units. Therefore, energy consumption can be reduced by exploiting these slacks with no impact on performance or under a performance constraint.
To find larger slacks to benefit leakage energy control, our second optimization is built upon a data-flow analysis to identify the idleness of a particular functional unit across basic blocks. Collecting this information from all basic blocks in the code, it then inserts explicit activate/deactivate instructions in the code to set/reset a sleep signal which controls leakage current for functional units.

Our third energy optimization targets at reducing leakage energy consumption in instruction caches. We propose and analyze two compiler-based strategies termed as “conservative” approach and “optimistic” approach. The conservative approach does not put an instruction cache line into a low leakage mode until it is certain that the current instruction in it is dead (i.e., it will not be accessed in the rest of the execution). In comparison, the optimistic approach places a cache line in low leakage mode if it detects that the next access to the instruction will occur only after a long gap. We evaluate different optimization alternatives by combining the compiler strategies with state-preserving and state-destroying leakage control mechanisms. Our results indicate that compiler-based cache leakage management can be very successful in practice.

With the scaling of technology and low supply voltage, the transient-error conscious system design is becoming a necessity for reliable functioning of hardware. While leakage energy can be reduced significantly by applying leakage control mechanisms, the aggressive use of such techniques can make the hardware circuits more susceptible to soft errors. Consequently, in the final portion of this thesis, we propose a novel solution to enhance the reliability for caches without compromising on performance. The idea is to replicate hot cache lines in rarely-used ones to provide a better error resilience.
## Table of Contents

List of Tables .......................................................... x

List of Figures .......................................................... xi

Acknowledgments ......................................................... xix

Chapter 1. Introduction and Motivation ............................. 1

Chapter 2. Major Contributions ....................................... 5

  2.1 Compiler-Directed Slack Scheduling for Dynamic and Leakage Energy Reduction ........................................... 5

  2.2 Compiler-Directed Leakage Energy Reduction for Functional Units .......................................................... 6

  2.3 Compiler-Directed Instruction Cache Leakage Optimization .......................................................... 6

  2.4 In-Cache Replication for Enhancing Data Cache Reliability .......................................................... 7

Chapter 3. Compiler-Directed Slack Scheduling for Dynamic and Leakage Energy Reduction ..................................... 8

  3.1 Support for Dynamic Energy Management ...................... 8

  3.2 Leakage Energy Management .................................... 9

    3.2.1 Input Vector Control ....................................... 9

    3.2.2 Power Supply Gating ..................................... 12

  3.3 Compiler Support ................................................ 13

    3.3.1 Algorithms for Dynamic Energy Reduction ............. 16
Chapter 3. Algorithms for Leakage Energy Reduction

3.3.1 Algorithm I

3.3.2 Algorithms for Leakage Energy Reduction

3.3.3 Combining Dynamic and Leakage Energy Reduction

3.4 Experimental Evaluation

3.4.1 Simulation Platform and Implementation

3.4.2 Benchmark Codes

3.4.3 Results and Discussion

3.4.3.1 Dynamic Energy Reduction

3.4.3.2 Comparing Voltage Scaling, Leakage Control and Unified Schemes

Chapter 4. Compiler-Directed Energy Reduction for Functional Units

4.1 Introduction and Motivation

4.2 Compiler Abstraction

4.3 Data Flow Analysis for Leakage Energy Reduction

4.3.1 Background on Control Flow Graph

4.3.2 Background on Data-Flow Analysis

4.3.3 Optimizing Leakage Energy Consumption

4.3.3.1 Data-Flow Analysis for Detecting Idleness

4.3.3.2 Inserting Activate/Deactivate Instructions

4.3.3.3 Discussion

4.4 Experimental Framework
4.4.1 Implementation ................................................. 54
4.4.2 Benchmarks and VLIW Configuration ..................... 55
4.4.3 Different Optimized Versions .............................. 56
4.5 Results ............................................................. 58

Chapter 5. Compiler-Directed Instruction Cache Leakage Energy Optimization .... 65
5.1 Introduction ....................................................... 65
5.2 Our Approach ..................................................... 67
5.2.1 Circuit Support ............................................. 67
5.2.2 Compiler Support ........................................... 69
5.2.3 Alternative Strategies ................................. 72
5.3 Experiments ...................................................... 73
5.3.1 Benchmarks and Simulation Platform .................. 73
5.3.2 Cache Life-Time Analysis .................. 77
5.3.3 Energy and Performance Results ........................ 78
5.4 Hybrid Strategy ................................................. 84
5.5 Impact of Compiler Optimizations ......................... 85

Chapter 6. In-Cache Replication for Enhancing Data Cache Reliability ......... 92
6.1 Introduction ....................................................... 92
6.2 Evicting Blocks Not in Use .................................. 95
6.3 Replication Design Space .................................. 96
6.3.1 Important Questions .................................... 96
6.3.2 Schemes Under Consideration ......................... 100
Chapter 6. Concluding Remarks and Directions for Future Research

6.4 Experimental Settings

6.4.1 Evaluation Metrics

6.5 Results

6.5.1 Replication Mechanisms

6.5.2 Comparing Performance — Aggressive Dead Block Prediction

6.5.3 Aggressiveness in Predicting Dead Blocks

6.5.4 Comparing Performance — Relaxed Dead Block Prediction

6.5.5 Comparing the Schemes with Error Injection

6.5.6 Performance Improvements

6.5.7 Sensitivity to Cache Parameters

6.5.8 Comparison with Write-Through Data Cache

6.5.9 Comparison with BaseECC with Speculative Loads

6.5.10 Performance, Energy, and Reliability Tradeoffs in Replicating Hot Cache Lines

Chapter 7. Concluding Remarks and Directions for Future Research

7.1 Summary

7.2 Future Research Directions

7.2.1 Integrating Software and Hardware Techniques for Optimizing Cache Energy

7.2.2 Compiler-Directed Hot Data Replication for Reliability

7.2.3 Compiler-Directed Functional Units Leakage Reduction for Super-scalar Architectures
6.1 Configuration parameters and their values in our base configuration for a super-scalar architecture. All caches are write-back. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
List of Figures

3.1 (a) VLIW architecture. (b) Version selection. . . . . . . . . . . . . . . . . . . . . . 9

3.2 (a) Block level implementation of input vector control scheme. (b) Latch design for setting input to one when sleep signal is activated. (c) Latch design for setting input to zero when sleep signal is activated. (d) Block diagram for gating supply voltage. When sleep signal is activated, the supply voltage to functional unit is gated. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.3 The effectiveness of the leakage control mechanisms. The leakage reduction column gives the average leakage current reduction due to the application of the control mechanism, and the initiation latency column indicates the time it takes for the control mechanism to take effect and recovery latency is the time required to return the functional unit to normal operational state. All the numbers presented above are extracted using Hspice circuit simulation of custom designed units in 0.25 micron technology using a threshold voltage of 0.48V. A cycle time of 1ns is used. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.4 (a) An example schedule where inslack and outslack of operation $d1$ are explicitly marked. (b) Different slacks for the schedule given in (a). (c) An example schedule that is the output of a performance-oriented schedule. (d) Optimizing the slack of operation $a1$ in (c) using Algorithm I. (e) Optimizing the slack of operation $a1$ in (c) using Algorithm II. (f) Optimizing the slack of operation $a1$ in (c) using the modified form of Algorithm II. (g) Conflicting slacks. (h) Impact of leakage control mechanism on energy consumption. (i) Comparison of dynamic energy control and leakage control. (j) Unified approach which uses both leakage control and dynamic energy control on the same slack. Note that dynamic energy consumption is associated with the first cycle of operation for convenience. It actually spans over the entire duration of the operation.

3.5 Algorithm I. .................................................. 19

3.6 Algorithm II. .................................................. 22

3.7 (a-c) Comparison of voltage scaling and input vector control. In (a), $r$ is 0.1 and the number of supply voltages is three (1V, 0.7V, 0.55V). In (b), $r$ is 0.1 and the number of supply voltages is two (1V and 0.7V). In (c), $r$ is 0.3 and the number of supply voltages is two (1V and 0.7V). (d) Unified approach with three supply voltages (1V, 0.7V, 0.55V) and $r$ being 0.3. In obtaining these results, energy values for 0.1 micron adder are used. ........................................ 25

3.8 Energy characteristics for the three 32-bit components using 0.25micron technology. The threshold voltage for these designs is 0.48V. .......................... 28

3.9 Simulation Framework ........................................... 29
3.10 Benchmark characteristics. The average length of slacks is the static length obtained for all IALU operations (including those without slack) in the basic block schedule. .......................................................... 30

3.11 Distribution of slacks across functional units. ........................................ 30

3.12 Percentage energy savings considering only IALU operations with slacks when using basic block and superblock scheduling (static results). ...................... 32

3.13 Overall runtime energy savings percentage for all IALU operations when voltage scaling is applied in conjunction with (a) basic-block scheduling (b) superblock scheduling. Algorithm II numbers are for three different performance degradation thresholds of 10%, 20% and 50%. ...................... 33

3.14 Energy savings for all IALU operations as compared to original case with no voltage scaling or leakage control (a) Static results with three supply voltages and basic block scheduling (b) Dynamic results with three supply voltages and basic block scheduling (c) Dynamic results with two supply voltages and basic block scheduling (d) Dynamic results with three supply voltages and superblock scheduling. .............................................................. 34

3.15 Overall runtime energy savings for all IALU operations for different number of IALUs in machine configuration. Basic block scheduling is used in conjunction with two supply voltages. ...................... 36

4.1 Data-flow (worklist) algorithm for computing \( OF_{i,j} \) and \( ON_{i,j} \) .......................... 44
4.2 (a) A control flow graph (CFG) annotated with $USE_{i,0}$. (b) $OFF_{i,0}$ (left part) and $ON_{i,0}$ (right part) sets for each basic block. (c) The ($ON_{i,0}$ or $OFF_{i,0}$) sets that will be used for inserting activate/deactivate instructions. (d) Final placement of activate/deactivate (ON/OFF) instructions. 48

4.3 (a) A multi-path execution pattern. (b) Pre-activating functional unit (early-ON). 53

4.4 Benchmark characteristics. 56

4.5 % energy improvements (with floating-point units). 58

4.6 % energy improvements (without floating-point units). 59

4.7 % distribution of energy savings (nbradar). 59

4.8 % energy savings with an reactivation latency of 45 cycles. 62

4.9 % increase in execution cycles. 63

4.10 % energy savings with different VLIW configurations (rawcaudio). 64

5.1 Leakage control circuitry. 69

5.2 A code fragment that contains three loops. 70

5.3 Four different implementation choices depending on the leakage control mechanism (mode) used and the compiler strategy employed. 73

5.4 Default parameters used in our simulations. 75

5.5 Benchmark codes used in our evaluations. The last column also contains the percentage contribution of leakage to overall instruction cache energy. Note that no leakage control mechanism is employed in obtaining this data. 76

5.6 Percentage of times where cache lines are in leakage control mode. 77
5.7 Normalized (w.r.t. Strategy I) energy consumption with different leakage saving strategies. ......................................................... 78
5.8 Energy breakdown for Strategy II .................................................. 80
5.9 Energy breakdown for Strategy III ................................................... 81
5.10 Extra dynamic energy consumption in off-chip L2 and memory. Only the strategies that employ state-destroying mode incur this penalty. ....... 82
5.11 Normalized (w.r.t. Strategy II) number of execution cycles with different leakage saving strategies. ..................................................... 82
5.12 Energy-delay (w.r.t. Strategy I) products with different leakage saving strategies. ........................................................................... 83
5.13 Normalized energy-delay product for the hybrid strategy. ................. 85
5.14 (a) A code fragment with a loop. (b) The distributed version of (a). (c) The instruction cache layout for (a). (d) The instruction cache layout for (d). ....... 87
5.15 Instruction cache leakage energy impact of optimizations on vpenta. All values are normalized w.r.t. to the case without power management. ................. 88
5.16 Performance impact of optimizations on vpenta. Y-axis is expressed in million cycles. ................................................................. 89
5.17 Instruction cache leakage energy impact of optimizations on adi. All values are normalized w.r.t. to the case without power management. ................. 90
5.18 Performance impact of optimizations on adi. .................................... 90
6.1 State transitions for cache decay. .................................................. 95
6.2 (a) Vertical replication. (b) Horizontal replication. ............................ 98
6.3 Replication ability for single and multiple attempt cases for ICR-P-PS (S).
In either case, we are trying to create a single replica for the primary block.
Multiple attempt case tries Distance-N/2 and Distance-N/4. .......................... 106

6.4 Loads with replica for single and multiple attempt cases for ICR-P-PS (S).
In either case, we are trying to create a single replica for the primary block.
Multiple attempt case tries Distance-N/2 and Distance-N/4. Only the bench-
marks that exhibit some difference are shown. ........................................... 106

6.5 Replication ability for single replica and two replicas for ICR-P-PS (S). For
the single replica case, we use Distance-N/2. For the multiple replica case, the
first replica tries Distance-N/2, and the second one tries Distance-N/4. .......... 107

6.6 Miss rates when single replica and multiple (two) replicas are created ICR-P-
PS (S). .......................................................... 108

6.7 Loads with replica for vertical and horizontal replications for ICR-P-PS (S).
The vertical replication uses Distance-N/2, while the horizontal replication uses
Distance-0. .................................................. 109

6.8 Replication ability for ICS-*(LS) and ICS-*(S) schemes. The results show that
ICS-*(LS) replicates more data than ICS-*(S). ............................................. 110

6.9 Loads with replica for ICS-*(LS) and ICS-*(S) schemes. ......................... 110

6.10 Miss rates for Base*, ICR-*(LS), and ICR-*(S) schemes. Both ICR-*(LS) and
ICR-*(S) increase the number of dL1 misses. ......................................... 111
6.11 Normalized execution cycles for all schemes. Compared to the increases in the number of misses, the increase in execution cycles is not excessive in some of our schemes. This is because superscalar execution hides many cache misses. The average increases over BaseP due to ICR-P-PS(S) and ICR-ECC-PS(S) are 3.6% and 21.0%, respectively.

6.12 Replication ability and loads with replica for ICR-P-PS (S) with different decay window sizes (vpr).

6.13 Normalized increased execution cycles with different decay window sizes (vpr). The increase in execution cycles due to ICR-P-PS (S) over BaseP is less than 4% for 1000 cycle window size and around 1.7% for 10000 cycle window size.

6.14 Normalized execution cycles with a decay window size of 1000 cycles. The average increases in execution cycles (over BaseP) due to BaseECC, ICR-P-PS (S), and ICR-ECC-PS (S) are 30.9%, 2.4%, and 10.2%, respectively.

6.15 Replication ability and loads with decay window sizes of 1000 cycles and 0 cycles.

6.16 Percentage of unrecoverable loads for vortex.

6.17 Normalized execution cycles when replicas are used for improving performance.

6.18 Performance and energy consumption (of L1 and L2 caches) with write-through dL1 for BaseP normalized with respect to ICR-P-PS(S) that uses a write-back dL1.
6.19 Performance and energy consumption (of L1 and L2 caches) for BaseECC assum- ing speculative (1-cycle) loads normalized with respect to ICR-P-PS(S), where replicas are left in place when primary copies are evicted. Representative results for Parity and ECC computations taking (b) 15% and 30% and (c) 10% and 30% respectively of the L1 access energy are given. 123

6.20 Baseline results. x-axis represents hot-block threshold (in cycles). The dead- block threshold is fixed at 1000 cycles. 127

6.21 Breakdown of extra dynamic energy. 130

6.22 PER (Performance-Energy-Reliability) metric. A higher value indicates a bet- ter operating point. 131

6.23 Breakdown of reads with no replicas. 132
Acknowledgments

First and foremost, I would like to thank my advisor, Professor Mahmut Kandemir, for his guidance, insight, and encouragement throughout my Ph.D study. I also thank Professor Vijaykrishnan Narayanan and Professor Anand Sivasubramaniam who helped to shape this work and provided me with their valuable advice. As a member of MDL at Penn State, I had the honor to work with Professor Mary Jane Irwin and I would like to thank her for her keen insights on the trend of future circuit and architecture techniques. I would also like to thank Dr. Richard Brooks for being a member of my research committee.

I would like to thank the members of the MDL research group for their friendly assistance and interesting discussions, of whom only a partial list is: Hyun Suk Kim, Lin Li, Jie S. Hu, Guangyu Chen, Ismail Kadayif, Victor De La Luz, Soontae Kim, Sudhanva Gurumurthi, YuhFang Tsai, Greg Link, Priya Unnikrishnan, and Ananth Hegde. And, special thanks to Vicki Keller for her professional assistance during my Ph.D study.

I am most grateful and indebted to my parents, for their love and support. Finally, I would like to thank my wife Minglu. Without her love, patience, and emotional support, this thesis would never have been done.
Chapter 1

Introduction and Motivation

Power and energy consumption has become a significant constraint in modern microprocessor design [105]. While energy-aware design is obviously crucial for battery driven mobile and embedded systems, it has also become important for desktops and servers due to packaging and cooling requirements where power consumption has grown from a few watts per chip to over 100 watts [57]. Voltage scaling and specialized circuit-level techniques have been the main strategies for low-power design. Unfortunately, these techniques alone are not sufficient; higher-level strategies for reducing power consumption are increasingly crucial [12]. Recent trends indicate that the architecture and software techniques, in addition to low-level circuit techniques, can play a major role in power-efficient computer system design [4, 12, 66, 63, 35, 59, 111].

Software has a significant impact on the overall energy consumption and is the main determinant factor for activities on the processor core, interconnect and memory system, which are, collectively, responsible for significant percentage of total power dissipation [57]. Based on this observation, there is a growing body of research in investigating energy-oriented compilation techniques and their interaction and integration with performance-oriented compiler optimizations [57, 128, 127, 142, 124, 71, 18, 150, 55, 56, 146, 60, 54]. While, to some extent, compiler optimizations for locality and performance can translate to optimizations for power [8], it is also important to develop innovative energy-oriented optimizations with no impact on performance or under pre-set performance constraints. In fact, energy-oriented optimizations under performance
constraints can be applied to reduce energy consumption or to make better tradeoffs between energy and performance [57, 131, 141, 30, 52, 25, 65, 91, 90].

Exploiting idle execution cycles provides such an opportunity to benefit energy while having a minimal impact on performance. Idleness widely exists in modern high-performance processors. For example, in a processor with multiple functional units, the functional units may not be fully utilized most of the time; many instructions are stored in the instruction cache for a long time after their last usage, wasting energy; data stored in the data cache may not be accessed for a certain period of time or not used at all; etc. The idleness of a particular hardware component can be exploited to reduce dynamic energy by scaling down voltage/frequency. The idleness can also be used to reduce leakage energy if we can apply the appropriate leakage energy management techniques.

Both hardware and software methods can be used to exploit idleness. We believe a compiler based approach has the advantage of analyzing the whole program behavior to make a better power management decision. A pure hardware-based approach has to rely on a monitoring mechanism at runtime, which itself consumes energy and increases area. Furthermore, there is a limitation for the number of activities it can monitor; therefore the pure hardware-based approach is limited in its ability to make correct decisions of power management; and an incorrect decision can potentially result in both the performance and energy overheads. Therefore, in this thesis, we focus mainly on compiler-directed techniques to reduce energy consumption.

VLIW (Very Long Instruction Word) architectures are being increasingly used in systems where energy/power-efficiency is an important consideration. For instance, TI TMS320C6x [129] and Philips R.E.A.L [103] are based on VLIW architecture. The Crusoe processor, which was designed to achieve low energy consumption while keeping a moderate performance, is also
based on a VLIW CPU engine [70]. VLIW architectures rely on the compiler to achieve high levels of instruction-level parallelism (ILP) with reduced hardware complexity, where the compiler plays the task of specifying when each operation will be executed, using which functional units, and with which registers as its operands. Thus, we mainly focus on VLIW architecture to study the energy-oriented compiler optimizations.

The energy efficiency of a VLIW architecture is determined by the underlying hardware and compiler technologies. There have been various approaches to reduce VLIW energy. In [1], the authors show how register assignment can be performed to reduce the number of spills to reduce the energy consumed by a VLIW ASIP. In [110], a data forwarding mechanism that reduces the number of register accesses for short-lived variables is presented. The experimental results reported indicate significant energy savings. In [82], the authors focus on the interaction of power-optimizing code transformations with the special performance improving sub-word instructions present in modern VLIW multi-media processors. All these optimizations improve energy as a “side effect” of the improvements in performance. In contrast, the optimizations considered in this thesis are essentially “energy-oriented”, and they either have no impact on performance or their impact can be controlled to enable interesting tradeoffs between energy consumption and performance. Therefore, our energy-oriented optimizations are different from previously-proposed performance-oriented optimizations, and they can be utilized to reduce the energy consumption, taking performance into consideration. In [113], operations within a VLIW instruction are rearranged to reduce the power consumed in the instruction-fetch datapath. While their work explicitly focused on reducing dynamic energy by minimizing number of bit transitions during instruction fetches, our optimization can reduce both dynamic and leakage energy.
This thesis proposes three energy-oriented optimizations. The first optimization is based on the observation that “slacks” present in the CPU datapath in VLIW machines due to the lack of available independent instructions, data dependences, and schedule-specific decisions. An optimizing compiler can analyze the program to identify the slack for each instruction, and energy consumption can be reduced by exploiting these slacks with no impact on performance or under a performance constraint [151]. Since leakage control mechanism benefits more for long slacks, our second optimization is built upon a data-flow analysis to identify idleness of a particular functional unit across basic blocks; and leakage energy can be reduced by exploiting the idleness detected [149]. Our third energy optimization is based on the generational behavior of the memory system [63]. We propose a compiler-directed scheme to exploit this behavior for reducing leakage energy of instruction caches with minimal impact on performance [147]. Our experimental results clearly show that all these three energy-saving techniques can be very successful in practice.

While energy can be reduced significantly by applying energy control mechanisms, the aggressive use of such techniques can make the hardware circuits more susceptible to soft errors [31]. Therefore, we also propose a novel solution to enhance the reliability for caches without compromising on their performance [148]. Our proposed solution replicates data that is in active use within the cache itself while evicting those that may not be needed in the near future. In addition, we study the cache-specific trade-offs between leakage saving and reliability.
Chapter 2

Major Contributions

This dissertation proposes three novel compiler-directed energy optimizations for VLIW architectures, and an approach for improving reliability for data caches.

2.1 Compiler-Directed Slack Scheduling for Dynamic and Leakage Energy Reduction

Scheduling slacks exist in today’s high performance microprocessor due to the limitation of parallelism in different applications and the limitation of the ability of compiler and/or processor to extract the parallelism. Casmira and Grunwald [17] found that 76-92% of the execution cycles have a "slackful" instructions. But, how one can quickly identify the scheduling slacks and effectively exploit them to reduce energy consumption is a challenging problem. Hardware-based slack detection implementation is proposed in [38], which includes a slack indicator table, re-order buffer, register file extension and scheduler modifications. This mechanism increases hardware complexity and requires dynamic scheduling support, which is not suitable for VLIW processors. This thesis proposes a compiler-directed scheme to detect and exploit the scheduling slacks without increasing hardware complexity.

We found plenty of static slacks based on our experimentation with VLIW architectures. As part of this thesis, two different algorithms are designed to exploit these slacks to reduce dynamic energy consumption of functional units. The first scheme reduces dynamic energy
consumption without sacrificing performance. The second scheme can reduce dynamic energy more aggressively under a user-specified parameter, which indicates the degree of performance loss that can be tolerated. We also propose a scheme to reduce leakage energy and a unified scheme to save both dynamic and leakage energy by exploiting compile-time slacks.

2.2 Compiler-Directed Leakage Energy Reduction for Functional Units

With the increasing number of transistors employed in current microprocessors and the continued reduction in threshold voltages of these transistors, leakage energy reduction has become an important issue [14]. Since longer idleness benefits leakage energy control, we propose a compiler-directed scheme to detect the idleness of the functional units and apply leakage control mechanism to reduce energy consumption of the functional units for VLIW machines.

Our experimental results show that the proposed compiler-based strategy is very effective in reducing leakage energy of functional units.

2.3 Compiler-Directed Instruction Cache Leakage Optimization

Cache lines typically see a flurry of use when first brought in, and then a period of idle time between their last access for the current data item and the point where a new data item is brought into that cache location [92]. Kaxiras et al [63] exploited this generational behavior to reduce cache leakage energy by using saturating counters to monitor the periods of inactivity in cache lines. While this approach is effective to reduce the cache leakage, it incurs an hardware overhead for each cache line and the energy penalty for decaying to the low-leakage mode only after fixed periods. In contrast to this hardware-centric approach, this thesis proposes two compiler-directed algorithms, termed as “conservative“ approach and “optimistic“ approach,
to turn off instruction cache lines based on this generational behavior of cache line usage [92] to reduce leakage energy consumption. Our compiler-directed approaches have the advantage to adapt to the different cache line usage pattern of different programs (or of different phases of the same program). These approaches identify the last use of the instructions and place the corresponding cache lines into a low leakage mode. The conservative approach does not put a cache line into a low leakage mode until it is certain that the current instruction in it is dead (i.e., it will not be used in the rest of execution). On the other hand, the optimistic approach places a cache line in low leakage mode if it detects that the next access to the instruction will occur only after a long gap. The experimental results show that the compiler-directed approaches we proposed are very competitive in terms of energy or energy-delay product with one of the recently proposed pure hardware-based leakage control schemes.

### 2.4 In-Cache Replication for Enhancing Data Cache Reliability

To address the susceptibility to soft errors due to aggressive use of leakage control mechanisms, we propose a novel solution to enhance the reliability for data caches without compromising on their performance. The proposed solution replicates data that is in active use within the cache itself while evicting those that may not be needed in the near future. In addition, we study the trade-offs between leakage saving and reliability by modulating the hot-block threshold, which determines whether a cache block is active enough to be replicated or not. Our experimental analysis reveals that a large fraction of the data read from the cache have replicas available with this optimization. Our results also indicate that having a hot-block threshold in the range of 10-1000 cycles can provide good reliability characteristics, without compromising on performance or power.
Chapter 3

Compiler-Directed Slack Scheduling for Dynamic and Leakage Energy Reduction

3.1 Support for Dynamic Energy Management

We assume a VLIW architecture (see Figure 3.1(a)) composed of integer ALUs (IALUs), floating point ALUs (FPALUs), one load/store (LD/ST) unit, and one branch (BR) unit. Integer ALUs have different versions that possess different performance (latency) and energy consumption characteristics. As shown in Figure 3.1(b), in the instruction word, a few control bits are associated with each functional unit to select the appropriate low energy version of the IALU. These control bits are used to route the control signals/data to the appropriate versions. Our algorithms take an already scheduled code and reschedules it. In doing so, it sets the control bits to appropriate values and modifies start time (taking into account data dependencies) and execution length (latency) of operations. The compiler ensures that the rescheduling does not change the issue width. Multiple versions of functional units also involve some circuit overheads. First, when the supply voltages to the different versions are different, a level converter circuit is required to interface the circuits operating at different voltages [106]. Second, as the multiple versions of the functional units operate at different frequencies, there is also a need for multiple clock domains. We limit the overhead of the multiple clock distribution circuitry by using local clock dividers wherever possible. In addition to multiple clock domains, we also require multiple supply rails for supporting the different versions of the units.
3.2 Leakage Energy Management

While dynamic energy consumption arises due to signal transitions, leakage current flows from every transistor that is powered on. With the increasing number of transistors employed in current microprocessors and the continued reduction in threshold voltages of these transistors, leakage energy consumption has become a major concern [14]. There are several leakage energy reduction techniques.

3.2.1 Input Vector Control

Many researchers have used models to estimate leakage and algorithms to find the minimum and maximum leakage of a given circuit [50]. It has been found that the leakage, as in the case of dynamic power, depends on the input pattern. This is a consequence of the transistor stacking effect, where a simple two-transistor stack can reduce leakage by a factor of up to 10. Additional stacking can only provide incrementally more savings [145]. The state of transistors...
Fig. 3.2. (a) Block level implementation of input vector control scheme. (b) Latch design for setting input to one when sleep signal is activated. (c) Latch design for setting input to zero when sleep signal is activated. (d) Block diagram for gating supply voltage. When sleep signal is activated, the supply voltage to functional unit is gated.
in the stack, however, is determined by the inputs. The objective is to find the input pattern that maximizes the number of disabled transistors in a stack. Clearly, the more complex the circuit (i.e., the larger the number of gates), the harder this task becomes. In [40], the authors use probabilistic theory to reduce the number of trials (simulations) as a function of how close the final result vector is to the absolute minimum leakage vector in terms of leakage energy. Obviously, to find the input vector closest to the minimum leakage vector, a more time-consuming search needs to be performed. Once this vector has been found, the input latches of the units can be designed such that a sleep signal sets the value of the unit’s inputs to the desired state (See Figure 3.2(a)). The implementation of the input control technique requires minimal architectural support. The overhead of the input latches is quite small. For example, the area overhead for setting the inputs of the multipliers is less than 10%. A sleep signal is activated whenever the unit is idle. This signal is set in our approach by the compiler as described in Section 3.3.2. Figures 3.2(b), and (c) show how values of one and zero are set at the inputs when the sleep signal is activated (set to one). When the sleep signal is low, the normal inputs are fed into the circuit. Note that if the switching incurred in setting the input to the desired sleep pattern causes a dynamic energy consumption larger than that produced by the reduction in leakage energy, this technique can increase overall energy consumption. Based on the relative values of the dynamic and leakage energy consumption and the duration of the idleness, the sleep signal needs to be activated intelligently.

We have quantified the leakage energy reduction for different integer ALU components using circuit-level simulation for 0.25 micron, 3.3V technology. Random input patterns were generated for each unit to provide a 95% confidence of finding the input vector that provides the least leakage current [40] and simulations were done for each of them. The second, third and
fourth columns of Figure 3.3 show the leakage power savings, initiation and recovery latencies due to the input control technique for three of the IALU components considered this work.

<table>
<thead>
<tr>
<th>Component</th>
<th>Input Vector Control</th>
<th>Power Supply Gating</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% Leakage Reduction</td>
<td>% Leakage Reduction</td>
</tr>
<tr>
<td></td>
<td>Initiation Latency</td>
<td>Initiation Latency</td>
</tr>
<tr>
<td></td>
<td>Recovery Latency</td>
<td>Recovery Latency</td>
</tr>
<tr>
<td>Adder</td>
<td>66</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1 cycle</td>
<td>480 cycles</td>
</tr>
<tr>
<td></td>
<td>0 cycle</td>
<td>480 cycles</td>
</tr>
<tr>
<td>Multiplier</td>
<td>18</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1 cycle</td>
<td>800 cycles</td>
</tr>
<tr>
<td></td>
<td>0 cycle</td>
<td>800 cycles</td>
</tr>
<tr>
<td>Shifter</td>
<td>86</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1 cycle</td>
<td>396 cycles</td>
</tr>
<tr>
<td></td>
<td>0 cycle</td>
<td>396 cycles</td>
</tr>
</tbody>
</table>

Fig. 3.3. The effectiveness of the leakage control mechanisms. The leakage reduction column gives the average leakage current reduction due to the application of the control mechanism, and the initiation latency column indicates the time it takes for the control mechanism to take effect and recovery latency is the time required to return the functional unit to normal operational state. All the numbers presented above are extracted using Hspice circuit simulation of custom designed units in 0.25 micron technology using a threshold voltage of 0.48V. A cycle time of 1ns is used.

3.2.2 Power Supply Gating

There are many ways in which power supply gating approach can be implemented [19] but the basic idea is to disconnect the power supply from the unit so that idle units do not consume any leakage energy. Supply gating can be implemented using a sleep transistor that serves as a pass transistor. The supply line $V_{DD}$ passes through this pass transistor to provide a gated supply voltage to the functional unit. The supply voltage to the functional unit is shut down when the sleep signal is activated to turn off the sleep transistor. The sleep transistor is built using a higher threshold voltage than the transistors in the functional unit. Thus, its leakage current in
the off state is negligible. In our implementation, we use a sleep transistor per functional unit as shown in Figure 3.2(d). Implementation of the power supply gating needs careful consideration of sleep mode transistor sizing to consider performance and noise immunity issues. The initiation/recovery latency for this technique is influenced by the diffusion capacitances of the sized sleep transistors. It must also be noted that frequent switching of large sleep transistors has dynamic energy overheads. Thus, we utilize this technique only to shutdown units for longer durations. The fifth, sixth and seventh columns of Figure 3.3 show the leakage power savings, initiation and recovery latencies due to power supply gating, respectively.

3.3 Compiler Support

The objective of the compiler support we explored is to exploit the idleness of functional units. The compiler can decide whether to employ dynamic or leakage energy control mechanisms depending on the scope for optimization and the relative magnitudes of leakage and dynamic energy. In reducing dynamic and leakage energy, the compiler exploits the hardware support discussed in the previous section. The algorithms considered in this section start with an already available schedule. In our implementation, this schedule is obtained using basic block-level [86] or superblock-level [23, 22, 21, 24, 48] scheduling. We start with these performance-oriented schedules so as to have a minimal impact on performance while exploiting the slacks for energy reduction. Note that the performance-oriented scheduling uses only the fastest version of each functional unit type.

Figure 3.4(a) presents an example input schedule for the energy optimization phase. All schedule figures are given as two-dimensional grids. In these schedule figures, each column denotes a functional unit and each row denotes a cycle. For example, in Figure 3.4(a), a schedule
Fig. 3.4.  (a) An example schedule where inslack and outslack of operation α1 are explicitly marked. (b) Different slacks for the schedule given in (a). (c) An example schedule that is the output of a performance-oriented schedule. (d) Optimizing the slack of operation α1 in (c) using Algorithm I. (e) Optimizing the slack of operation α1 in (c) using Algorithm II. (f) Optimizing the slack of operation α1 in (c) using the modified form of Algorithm II. (g) Conflicting slacks. (h) Impact of leakage control mechanism on energy consumption. (i) Comparison of dynamic energy control and leakage control. (j) Unified approach which uses both leakage control and dynamic energy control on the same slack. Note that dynamic energy consumption is associated with the first cycle of operation for convenience. It actually spans over the entire duration of the operation.
fragment for five functional units: four integer ALUs (IALU-0 thru IALU-3), and a load/store unit (LD/ST) is shown. Operation $a_0$ is scheduled to be executed in the first cycle in IALU-0 whereas operation $e_1$ is scheduled to be executed in IALU-2 in the second cycle.

For each operation, we define an inslack and an outslack. The outslack of an operation is the number of cycles between the end of the current operation and the start of the next (earliest) dependent operation. Similarly, the inslack of an operation is the number of cycles between the latest end time of all operations on which the current operation is dependent and the start of the current operation. We use the term slack refers to the sum of the number of cycles in both inslack and outslack. Note that this slack definition is with respect to data dependences between instructions and is independent of the underlying architecture and the types, numbers, and versions (e.g., low-energy versus high-energy) of functional units. However, whether a given slack can be exploited or not depends not only on data dependences but also on the architecture in question. We define a slack as exploitable if the corresponding operation can be executed in a slower, low-energy version of a functional unit without violating any data dependencies. This definition implies that there should be an available low-energy version to execute the operation and that the low-energy version’s latency must be smaller than or equal to the sum of the slack of the operation and the latency for executing the operation using the fastest version.

As an example, the inslack and outslack of operation $d_1$ are explicitly illustrated in Figure 3.4(a). Here, we assume that $d_1$ is only dependent on $d_0$ and that $d_2$ is the only operation dependent on $d_1$. To illustrate the difference between slack and exploitable slack, we consider the schedule fragment shown in Figure 3.4(c) which contains four IALUs. We focus on operation $a_1$ which has four idle cycles before it and five idle cycles after it. Assuming that $a_1$ is dependent on only $b_0$ and that $c_0$ is dependent on $a_1$, we cannot exploit all of these nine slack
cycles for reducing energy even if we have a low-energy unit with nine cycles of latency. This is because data dependences with \( b_0 \) and \( c_0 \) put a restriction on the number of cycles that can be used to execute \( a_1 \). Consequently, the inslack and outslack for \( a_1 \) are 2 and 3, respectively. Suppose now that we have two low-energy functional units that can execute \( a_1 \) in 4 and 7 cycles, respectively. Since the operation \( a_1 \) needs to complete in six cycles (latency of the fastest version + slack duration), we can only use the first functional unit. Therefore, the exploitable slack for this operation is 3 (note this is the difference between the latency on the low-energy version on which operation is scheduled and the latency of the original high-performance version).

Note that schedule-specific decisions impact the availability of low-energy functional units for exploiting slacks. Consider, for instance, operation \( c_1 \) scheduled on IALU-2 in Figure 3.4(b). Assuming that this operation is not involved in any data dependence relation, it should normally be possible to prolong its execution and save energy. However, if we have only one low-energy IALU and decide to use it for \( c_0 \), it will not be possible to exploit \( c_1 \)'s slack. Similarly, let us assume that \( a_0, a_1, \) and \( b_0 \) are independent operations and can all be executed in low-energy functional units. Depending on whether the performance-oriented scheduling assigns \( a_0 \) to IALU-0 (as shown in the figure) or IALU-1, we may or may not be able to exploit the slacks due to operations \( a_1 \) and \( b_0 \).

### 3.3.1 Algorithms for Dynamic Energy Reduction

Figure 3.4(c) shows how a given slack can be exploited to reduce dynamic energy consumption. As mentioned earlier, the inslack and outslack for \( a_1 \) are 2 and 3, respectively. Below we present two algorithms that take advantage of these slacks by scheduling the operation in question in a slower, less energy-consuming version of the corresponding functional unit. Both
algorithms assume that the VLIW architecture in question has multiple (low-energy) versions of each functional unit type. The first algorithm works without increasing the schedule length of the original performance-oriented scheduling. The second algorithm, on the other hand, allows a user-specified performance degradation if doing so leads to larger energy savings (as compared to the first algorithm). Both algorithms first order the operations to be exploited for energy optimization. Then, for each operation (in order), considering its exploitable inslack and taking into account number of IALU operations that can be issued in a cycle, the start time of the operation is set to the earliest possible time. After that, considering the outslack of the operation and the available low-energy versions of the corresponding unit, the most suitable low-energy version of the unit is selected and the operation is scheduled in that unit.

3.3.1.1 Algorithm I

The first algorithm is based on the idea that the low-energy units are used only if this does not increase the length of the performance-oriented schedule. This can be achieved by not prolonging the (compile-time estimated) execution of the operation beyond its outslack. The idea can be best explained using an example. Figure 3.4(d) shows how the operation $a_1$ in Figure 3.4(c) is scheduled in a low-energy unit with a latency of 6. Note that the operation is scheduled to start at the fourth cycle and finish at the ninth cycle. It should also be noted that if the latency of the unit was 5, the operation would finish at the eight cycle (but, it would still start at the fourth cycle). However, if the latency of a candidate low-energy version was 7, it would not be possible to use this (version of the) unit for this operation (as it causes the finish time to go beyond the ninth cycle, which is end of the outslack).
A sketch of this algorithm is given in Figure 3.5. The algorithm takes a region of code to schedule (region) and a table (table) that gives energy consumption and latency for each IALU component. `compute_slack()` computes the inslack and outslack for each operation in the region and `build_slack_list()` builds a list of operations with slacks. In the for-loop, we employ a selection heuristic to determine the most beneficial operation candidate for slack exploitation. Note that it is very important to determine a suitable order of processing for operations as exploiting one slack might prevent another slack from being exploited. This is illustrated in Figure 3.4(g) where fully exploiting the slack for operation $a_1$ prevents the outslack for operation $a_2$ from being exploited if $a_1$ depends on $a_2$. Consequently, our selection heuristic evaluates each and every operation with a slack and calculates the potential energy gain if the associated slack is exploited. The potential gain is the difference in energy consumption between the fastest (and most energy-consuming) version of the unit and the most-energy saving version that does not distort any data dependence. Our approach, using the `heuristic_energy()` function, selects the operation with the largest potential gain (`curr_max` keeps the maximum potential energy gain found so far). After selecting an operation, the scheduler updates the code region, and calls itself with the updated region. `compute_latency()` return the largest latency value (from table) which is less than or equal to the sum of slack of the operation $next_op$ and the minimum latency for executing the operation using the fastest version. In Figure 3.5, `compute_energy()` returns the corresponding energy value. Variables `latency` and `energy` keep the latency and energy consumption of the version on which $next_op$ is scheduled. Finally, `stime` and `stime_old` are the updated and original start times of $next_op$. 
INPUT: A sequence of operations ("region") scheduled using a performance-oriented scheduler; A table that gives energy consumption and latency for each IALU component ("table")

OUTPUT: A scheduled set of operations where slacks have been exploited

Algorithm I(region, table, ...)

begin
    compute_slack(region);
    list = build_slack_list(region);
    curr_max = -1;
    next_op = NULL;
    for each operation op in list do
        if (heuristic_energy(op) > curr_max) then
            next_op = op;
            curr_max = heuristic_energy(op);
        endif
    endfor
    if (next_op ! = NULL) then
        stime = stime_old - inslack;
        latency = compute_latency(next_op);
        energy = compute_energy(next_op);
        update_region(region, next_op, stime, energy, latency);
        Algorithm I(region, table, ...);
    endif
end

Fig. 3.5. Algorithm I.
3.3.1.2 Algorithm II

This algorithm attempts to increase dynamic energy savings further by allowing a user-specified increase in schedule length. Note that this might be a reasonable approach in many embedded/portable environments where energy consumption holds a first-class status. Informally, this algorithm checks whether, for a given operation with slack, using a more energy-saving version than the one that would normally be selected by Algorithm I is possible without exceeding a performance degradation threshold (PDT), a user-specified parameter. If so, it employs this more energy-saving option and considers the next operation (and its slack). An example application of this approach is illustrated in Figure 3.4(e), which shows the optimized version of Figure 3.4(c). As discussed in the previous subsection, the first algorithm would exploit the slack of operation \( a_1 \) by scheduling its execution over six cycles. Note that considering the in-slack and outslack, it is not possible to achieve a better result using the first algorithm even if we have a more energy-saving version (of the same unit) with a latency of 7 cycles. However, if we are allowed to increase the schedule length by one cycle, we can extend the execution of this operation to 7 cycles as shown in Figure 3.4(e). This can be achieved by inserting one empty cycle to the schedule. There are two important points here that should be mentioned. First, in general, we might need to add more than one cycle to the schedule (depending on the latency of the low-energy version being considered). Second, the added cycles can also allow the operations scheduled in other functional units to extend their slacks, and potentially, exploit them by selecting more energy-savings versions.
Figure 3.6 gives a sketch of this algorithm. Since the algorithm needs to check whether the PDT has been exceeded or not, before calling this algorithm, a `compute_total_cycles()` function, which computes the schedule length of the original schedule, is called and the result (`total_cycles`) is passed as parameter to the algorithm. The first part of the algorithm proceeds the same way as that of the first algorithm, and builds a slack list. However, in selecting the most beneficial operation, it uses a different heuristic, implemented in `heuristic_pdd()`, which tries to select the most energy-efficient version of the unit without taking into account the exploitable slack. Then, within the second if statement, a slack difference (`slack_diff`) between the latency of the unit and the available slack is calculated. Subsequently, if the operation can be executed in this most energy-efficient unit without exceeding the PDT, the operation is scheduled and the region is updated. Otherwise, we schedule this operation (using the `schedule_slack_without_extending()` function) in the same way it would be scheduled by the first algorithm (i.e., without extending the schedule length). Finally, the algorithm calls itself recursively. In this algorithm, `get_latency()` returns the largest latency with which `next_op` can be exploited and `get_energy()` gives the corresponding energy value. `update_dfg()` is the function used to insert empty cycles.

### 3.3.2 Algorithms for Leakage Energy Reduction

As leakage energy is becoming a significant portion of the overall energy consumption [14], it is also important to study software techniques to reduce leakage energy. The algorithms for leakage energy reduction exploit the slacks in the schedule by activating the appropriate leakage control mechanism. The compiler is provided with the information on the latency to invoke the leakage control mechanism, the latency to restore the unit to normal mode, the potential
INPUT: A sequence of operations ("region") scheduled using a performance-oriented scheduler;
A table that gives energy consumption and latency for each IALU component ("table");
A performance degradation threshold (PDT)

OUTPUT: A scheduled set of operations where slacks have been exploited

Algorithm II\(\text{II}(\text{region, table, PDT, total cycles, ...})\)
begin
    compute\_slack(\text{region});
    list = build\_slack\_list(\text{region});
    curr\_max = -1;
    next\_op = NULL;
    for each operation \text{op} in list do
        if (heuristic\_pdd(\text{op}) > curr\_max) then
            next\_op = \text{op};
            curr\_max = heuristic\_pdd(\text{op});
        endif
    endfor
    if (next\_op \neq NULL) then
        latency = get\_latency(next\_op);
        slack = get\_slack(next\_op);
        slack\_diff = latency - slack;
        total\_extended\_cycles = total\_extended\_cycles + slack\_diff;
        performance\_degradation = total\_extended\_cycles / total\_cycles;
        if (performance\_degradation \leq PDT) then
            stime = stime\_old - ins\_slack;
            energy = get\_energy(next\_op);
            update\_region(\text{region, next\_op, stime, energy, latency});
            if (slack\_diff > 0) then
                update\_dfg(\text{region, next\_op, slack\_diff});
            endif
        endif
    else
        total\_extended\_cycles = total\_extended\_cycles - slack\_diff;
        performance\_degradation = total\_extended\_cycles / total\_cycles;
        schedule\_slack\_without\_extending();
    endif
end\(\text{Algorithm II}())

Fig. 3.6. Algorithm II.
leakage energy reduction, and any additional overhead energy associated with the application of the leakage control mechanism.

It must be noted that the definition of the slack is slightly different when considering leakage control. Since the application of leakage control techniques is not impacted by data dependencies, we define the slack for leakage control as the duration between two successive accesses to the unit in question. Note that this implies leakage control can exploit larger slacks than voltage scaling, which is restricted by the data dependences.

It should be noted that when the leakage control mechanism is employed, we incur extra dynamic energy consumption in the second cycle. The scheduling algorithm determines for each slack whether to activate the input control mechanism or not. It utilizes the following expression to determine whether the slack can be exploited:

\[
E_d + (k + 1)E_l \geq 2E_d + E_l + r'E_l + r(k - 1)E_l
\]  \hspace{1cm} (3.1)

In this equation, \(E_l\) is the leakage energy per cycle and \(E_d\) is the dynamic energy per operation, \(r\) is leakage energy reduction factor (i.e., if the original leakage energy is \(E_l\), the optimized energy is \(rE_l\)), \(k\) is the slack duration in cycles, \(r'\) is the leakage energy reduction factor during the current-settling time. Note that assuming \(r = r'\) and a single-cycle current-settling time, we see that leakage energy is beneficial when \(pk(1 - r) - 1 > 0\), where \(E_l = pE_d\).

3.3.3 Combining Dynamic and Leakage Energy Reduction

In the previous sections, dynamic and leakage energy reduction have been explored individually. As the relative magnitudes of dynamic and leakage energy become comparable, it
will become important to reduce both in an integrated fashion. In this section, two alternate
approaches for combining voltage scaling and leakage reduction by the compiler are presented.
Since our current voltage scaling scheme for dynamic energy reduction works on instruction
granularity, we consider here only input vector control for the leakage reduction.

In the first approach, the compiler determines whether it is better to exploit dynamic
or leakage energy reduction for each slack duration independently. To achieve this, it uses the
following expression to determine which technique to employ (see Figure 3.4(i)):

$$E_{dk} + (k + 1)E_{lk} < 2E_d + (1 + r' + (k - 1)r)E_l$$  (3.2)

where $E_{dk}$ is the dynamic energy consumed when voltage scaling is employed to increase the
latency of the operation to exploit the slack of $k$ cycles, and $E_{lk}$ is the corresponding leakage en-
ergy per cycle when voltage scaling is employed. Note that leakage energy also scales down with
supply voltage scaling. If the condition specified by the above expression is satisfied, voltage
scaling is employed to exploit the slack; otherwise, the leakage control mechanism is activated.

Figure 3.7 shows for each slack duration, the energy consumed when no optimization
is performed, when only voltage scaling is applied, and when only the input vector control
is applied. It is interesting to note that, based on slack duration, either voltage scaling or in-
put vector control mechanism generates the best result, motivating an integrated approach that
employs different energy reduction mechanisms depending on slack size. In Figure 3.7(a), volt-
age scaling is the preferred technique for slacks of duration less than 10 cycles. Beyond this
crossover point, the leakage reduction mechanism becomes favorable. This crossover point is
influenced by the number of available functional units with different supply voltages (which is
Fig. 3.7. (a-c) Comparison of voltage scaling and input vector control. In (a), $r$ is 0.1 and the number of supply voltages is three (1V, 0.7V, 0.55V). In (b), $r$ is 0.1 and the number of supply voltages is two (1V and 0.7V). In (c), $r$ is 0.3 and the number of supply voltages is two (1V and 0.7V). (d) Unified approach with three supply voltages (1V, 0.7V, 0.55V) and $r$ being 0.3. In obtaining these results, energy values for 0.1 micron adder are used.
three in Figure 3.7(a)). When we reduce the number of available supply voltages to two (e.g., in an attempt to reduce unit replication and multiple supply line overheads), the crossover point shifts to a slack duration of 5 cycles as can be observed from Figure 3.7(b). This shift is caused by the inability to exploit larger slacks due to the non-availability of functional units using the third supply voltage. In contrast, the input vector control is able to amortize the initial dynamic energy consumption overhead better with larger slacks. The leakage energy reduction factor, \( r \), (when the leakage control mechanism is used) also influences the location of the crossover point. Figure 3.7(c) shows that the crossover point increases to 10 when the leakage reduction is less, even in the presence of just two supply voltages. These results emphasize the need for the compiler designers to be aware of parameters that can influence their technique of choice for slack exploitation.

In the second approach, which we call \textit{unified}, the goal is to investigate whether both the leakage and dynamic energy management schemes can be exploited at different intervals of \textit{the same slack}. The idea is illustrated in Figure 3.4(j). Basically, the compiler selects an optimum value \( f (< k) \) up to which dynamic energy is reduced and beyond which leakage energy is reduced. This scheme is in contrast to the previous approach that determines a single technique to employ for each slack. Figure 3.7(d) shows how the unified approach compares to using the best of the dynamic and leakage control mechanisms individually for each slack duration explained in the previous paragraph. We see that the unified approach performs best as it combines the best of both the techniques. Since the individual slack exploitation for leakage or dynamic energy control (discussed in the previous paragraph) is subsumed by this technique, we only consider the unified scheme in the rest of the dissertation.
3.4 Experimental Evaluation

In this section, we discuss our implementation and simulation environment (Section 3.4.1), introduce our benchmark codes (Section 3.4.2), and present our results (Section 3.4.3).

3.4.1 Simulation Platform and Implementation

Trimaran is a compiler infrastructure to provide a vehicle for implementation and experimentation in state-of-the-art research in compiler techniques for Instruction Level Parallelism (ILP) [130]. As can be seen in Figure 3.9, a program flows through IMPACT, Elcor, and the cycle-level simulator. IMPACT applies machine-independent classical optimizations and transformations to the source program, whereas Elcor is responsible for machine-dependent optimizations and scheduling. The proposed algorithms are implemented in Elcor. The increase in compilation time due to our algorithms was around 20% on average. The cycle-level simulator was modified to record the activity of the IALU units. Further, the simulator was augmented to support a cache hierarchy. This recorded information was used along with energy parameters to evaluate the energy consumption. The energy estimation is activity-based in which energy consumption is based on number of accesses to the components. The dynamic energy parameters and leakage reduction factors used are based on actual circuit-level simulation of the components. Figure 3.8 shows the energy parameters of three IALU components (adder/subtractor, shifter and multiplier) for three supply voltages. These numbers are based on actual layouts performed in 0.25 micron technology. Scaling factors [83] are applied to these values to obtain corresponding values for 0.10 micron with a 1V supply voltage and 0.2V threshold voltage. The
leakage reduction numbers are extracted from the Figure 3.3. The default configuration for our experiments uses four IALU, two FPALU, one LD/ST unit and one branch unit.

<table>
<thead>
<tr>
<th>IALU Component</th>
<th>Supply Voltage</th>
<th>Dynamic Energy (pJ)</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder</td>
<td>3.3V</td>
<td>66.6</td>
<td>1 cycle</td>
</tr>
<tr>
<td></td>
<td>2.1V</td>
<td>26.9</td>
<td>2 cycles</td>
</tr>
<tr>
<td></td>
<td>1.7V</td>
<td>17.6</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Multiplier</td>
<td>3.3V</td>
<td>258.0</td>
<td>8 cycles</td>
</tr>
<tr>
<td></td>
<td>2.1V</td>
<td>104.5</td>
<td>14 cycles</td>
</tr>
<tr>
<td></td>
<td>1.7V</td>
<td>68.4</td>
<td>27 cycles</td>
</tr>
<tr>
<td>Shifter</td>
<td>3.3V</td>
<td>66.1</td>
<td>4 cycles</td>
</tr>
<tr>
<td></td>
<td>2.1V</td>
<td>26.8</td>
<td>8 cycles</td>
</tr>
<tr>
<td></td>
<td>1.7V</td>
<td>17.5</td>
<td>16 cycles</td>
</tr>
</tbody>
</table>

Fig. 3.8. Energy characteristics for the three 32-bit components using 0.25micron technology. The threshold voltage for these designs is 0.48V.

We present results for the different optimizations both using compile time metrics and runtime metrics. The energy savings estimated at the compile time are provided by analyzing the slacks using Elcor (without taking into account conditionals and loop bounds) and are called static results. The energy savings at run time are estimated using the cycle-accurate simulator and are called dynamic results. It must be observed that the dynamic results depend on the number of times each portion of the schedule is executed. All energy saving numbers are reported with respect to an architecture that uses a performance oriented schedule with no support for voltage scaling or leakage control. As we apply our techniques to the IALU, we consider the energy consumed only by the IALU operations.
3.4.2 Benchmark Codes

To evaluate the effectiveness of our algorithms, we used a suite of fifteen programs from different benchmark sets. The important characteristics of these codes are given in Figure 3.10. The third column in this figure gives the number of slacks in each code and the fourth column gives the average slack length (in cycles). The fifth column shows the percentage of exploitable slacks. On average, 51.4% of the slacks are exploitable and the average slack length is 3.21. Figure 3.11 shows the distribution of slacks across different types of operations. It can be observed that, in these codes, more than 88% of the slacks, on the average, occur with integer ALU operations using our default configuration. This provides a strong motivation for us to focus on these operations for exploiting slacks.
<table>
<thead>
<tr>
<th>Program</th>
<th>Source</th>
<th>Number of Slacks</th>
<th>Average Slack Length</th>
<th>% Exploitable Slacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>SpecInt95</td>
<td>2,741</td>
<td>3.99</td>
<td>34%</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>SpecInt95</td>
<td>1,809</td>
<td>3.78</td>
<td>32%</td>
</tr>
<tr>
<td>129.compress</td>
<td>SpecInt95</td>
<td>632</td>
<td>3.47</td>
<td>44%</td>
</tr>
<tr>
<td>130.li</td>
<td>SpecInt95</td>
<td>1,077</td>
<td>4.22</td>
<td>49%</td>
</tr>
<tr>
<td>132.jpeg</td>
<td>SpecInt95</td>
<td>1,360</td>
<td>4.27</td>
<td>49%</td>
</tr>
<tr>
<td>convolution</td>
<td>DSPstone</td>
<td>33</td>
<td>1.50</td>
<td>50%</td>
</tr>
<tr>
<td>dot_product</td>
<td>DSPstone</td>
<td>29</td>
<td>1.75</td>
<td>25%</td>
</tr>
<tr>
<td>fir</td>
<td>DSPstone</td>
<td>59</td>
<td>2.00</td>
<td>75%</td>
</tr>
<tr>
<td>n_complex_updates</td>
<td>DSPstone</td>
<td>61</td>
<td>1.53</td>
<td>90%</td>
</tr>
<tr>
<td>n_real_updates</td>
<td>DSPstone</td>
<td>64</td>
<td>1.56</td>
<td>67%</td>
</tr>
<tr>
<td>cordic</td>
<td>Mediabench</td>
<td>456</td>
<td>5.73</td>
<td>14%</td>
</tr>
<tr>
<td>idea</td>
<td>Mediabench</td>
<td>668</td>
<td>3.76</td>
<td>75%</td>
</tr>
<tr>
<td>nbradar</td>
<td>Mediabench</td>
<td>441</td>
<td>6.60</td>
<td>61%</td>
</tr>
<tr>
<td>paraffins</td>
<td>Trimaran</td>
<td>383</td>
<td>2.34</td>
<td>52%</td>
</tr>
<tr>
<td>rawcaudio</td>
<td>Mediabench</td>
<td>83</td>
<td>1.74</td>
<td>54%</td>
</tr>
</tbody>
</table>

Fig. 3.10. Benchmark characteristics. The average length of slacks is the static length obtained for all IALU operations (including those without slack) in the basic block schedule.

![Slack Distribution](image)

Fig. 3.11. Distribution of slacks across functional units.
3.4.3 Results and Discussion

3.4.3.1 Dynamic Energy Reduction

In this subsection, we assume that there are three versions for each IALU (with each version operating with a different supply voltage) in the default configuration that can be used simultaneously (that is a total of 12 IALU versions) The energy value corresponding to each version is shown in Figure 3.8. It should be emphasized that the leakage energy contribution for this technology (0.25 micron) is not significant as compared to dynamic energy consumption (around 3% of overall energy for a junction temperature of 110°C).

Figure 3.12 shows that a 70.8% energy saving is possible when only considering energy consumption of operations with exploitable slacks (i.e., the operations that can be optimized by Algorithm I). This is obtained starting with a basic-block oriented performance schedule. The corresponding number when superblock scheduling is used is 71.6%. These static results show that the proposed approach can cover the slacks successfully across applications from different benchmark suites.

In order to evaluate how this translates to actual energy savings, Figure 3.13(a) provides the overall energy savings obtained by running the benchmarks through the simulator (dynamic results). The average energy savings by using Algorithm I which employs voltage scaling is 32.3%. Further energy savings can be obtained at the cost of performance when Algorithm II with three different PDTs is used as observed from the figure. The average energy savings across the different benchmarks for these three PDTs are 50.1%, 63.7% and 71.3%, respectively. It can be observed that the additional energy savings are small when moving from 20% to 50% performance degradation. This is because of the limited number of supply voltages that imposes an
3.4.3.2 Comparing Voltage Scaling, Leakage Control and Unified Schemes

In this subsection, we use the 0.1 micron, 1V supply voltage technology energy parameters as leakage and dynamic energy become comparable in this technology. The leakage energy reduction factor (r) for the adder, shifter and multiplier used were 0.34, 0.14, and 0.72, respectively. Further, we use a value of p = 1, that is leakage energy per cycle is equal to the dynamic

Fig. 3.12. Percentage energy savings considering only IALU operations with slacks when using basic block and superblock scheduling (static results).
Fig. 3.13. Overall runtime energy savings percentage for all IALU operations when voltage scaling is applied in conjunction with (a) basic-block scheduling (b) super-block scheduling. Algorithm II numbers are for three different performance degradation thresholds of 10%, 20% and 50%.

energy per operation executed on that unit. The voltage scaling technique is the same as that used for Algorithm I in the previous section. When optimizing for only leakage energy (using input vector control), it is assumed that only the highest performance version of each IALU is available. All low-energy versions are supply-gated to completely eliminate leakage energy. Finally, the unified scheme employs a combination of leakage control and voltage scaling as explained earlier. In the unified scheme, input vector control is applied whenever a functional unit becomes idle.

Figure 3.14(a) gives the energy savings of the three schemes obtained from compiler (static results) when using basic block scheduling. Note that these are cumulative energy savings across all basic blocks. In eleven out of fifteen cases voltage scaling is estimated to perform better than leakage reduction at compile time. The unified approach is the best in all cases. The corresponding dynamic results are shown in Figure 3.14(b). The effectiveness of each scheme
Fig. 3.14. Energy savings for all IALU operations as compared to original case with no voltage scaling or leakage control (a) Static results with three supply voltages and basic block scheduling (b) Dynamic results with three supply voltages and basic block scheduling (c) Dynamic results with two supply voltages and basic block scheduling (d) Dynamic results with three supply voltages and superblock scheduling.
depends on the exploitable slack durations in the schedule. It can be observed that for nine of the benchmarks, the leakage control mechanism outperforms voltage scaling. The reason that there are larger energy gains in favor of leakage control when we move from static to dynamic results is two-fold. First, in some benchmarks, basic blocks with slacks larger than the average are executed more frequently. Second, the compiler visible slacks are prolonged during execution due to unexpected delays such as cache miss stalls. We also note that the unified scheme brings an average of 22.2% and 20.9% improvement over the voltage scaling and input vector control mechanisms, respectively. Figure 3.14(c) gives the dynamic results when only two supply voltages (1V and 0.7V) are employed. Due to the limited number of supply voltages, the average benefits from voltage scaling reduce from 32.7% to 25.4% on the average. Consequently, the gains due to unified scheme also reduce from 54.9% to 50.7% on the average. Figure 3.14(d) shows the dynamic results when superblock scheduling is employed with three supply voltages. It can be observed that as compared to basic block scheduling, the voltage scaling performs better than leakage control. This is due to the reduction in the slack length duration. Specifically, when moving from basic block to superblock, average slack length reduces by 12%.

Figure 3.15 shows the dynamic results when the number of IALUs is varied for two representative benchmarks. As the number of IALUs increases, initially, voltage scaling is able to exploit the additional slacks in the schedule. However, beyond a certain number of IALUs, the data dependencies and increased leakage energy of the unused units cause a decrease in energy benefits. Another trend, we observe, is the increasing effectiveness of leakage control mechanisms as the number of functional units increases. As the unified scheme combines the best of dynamic and leakage control mechanisms, it is quite effective across all configurations.
Fig. 3.15. Overall runtime energy savings for all IALU operations for different number of IALUs in machine configuration. Basic block scheduling is used in conjunction with two supply voltages.
Chapter 4

Compiler-Directed Energy Reduction for Functional Units

4.1 Introduction and Motivation

While dynamic energy is the dominant energy component in today’s CMOS circuits, the trends show that leakage energy consumption will play a much larger role in upcoming circuit generations [19]. As opposed to dynamic energy which is spent only when there is a bit transition (switching) activity, leakage consumption occurs as long as the circuit is powered on, irrespective of whether it is accessed or not. As a result, idle (unused) functional units consume leakage energy when power supply is maintained to them.

Previous research on leakage optimization focused on circuit-level [19] and architectural-level [63, 35] techniques to reduce leakage energy. We believe that an optimizing compiler can be in a good position to exploit circuit-level and architectural mechanisms, if a suitable abstraction can be provided to it. This is because the compiler can analyze the code, identify which functional units will be idle during execution, and estimate their duration of idleness. It can then exploit this information by activating and deactivating leakage control mechanism(s) provided by the underlying architecture. In chapter3, we found that leakage control mechanism benefits more to large slacks. In chapter3, the compiler exploit idleness of functional units at the operation granularity to reduce leakage energy; however, compiler can detect longer idleness by analyze the entire program. Therefore, compiler has the potential to reduce the leakage energy of functional units further by obtaining global information about the idleness.
We propose a compiler-based leakage energy optimization strategy. Our strategy first analyzes the control flow graph (CFG) representation of the program and, for each functional unit, determines the paths along which that functional unit is idle (unused). It then selects a suitable leakage control mechanism and inserts activate/deactivate instructions in the code to enable/disable the functional unit. Since reactivating a functional unit from the leakage control mode takes some extra execution cycles, we also consider a circuit pre-activation strategy which tries to bring the circuit to the normal operation mode before it is actually needed. In this work, two leakage control mechanisms are considered to exploit the idleness of functional units for reducing leakage energy. The first leakage control mechanism, called input vector control, exploits the state dependence of the leakage current and sets the inputs to values that have the minimum leakage current when the units are idle [40]. The second mechanism, called supply gating, eliminates the leakage energy consumption by cutting the power supply to the units [88].

To evaluate the effectiveness of our strategy, we implemented different versions of it using an experimental compiler and simulation environment and conducted experiments using a VLIW (very long instruction word) architecture and several media applications and array-intensive codes from different domains. It should be emphasized that VLIW architectures are increasingly being used in media applications and are perfect candidates for compiler optimizations as, in these architectures, the compiler determines the order in which the instructions will be executed. Our experimental results show that the proposed compiler-based strategy is very effective in reducing leakage energy. Our results also indicate that employing two leakage optimization mechanisms simultaneously improves upon using a single mechanism. In addition, we find that in determining the idleness of functional units, using profile data is more effective than using compiler estimation.
In [17], a limit study of exploitable slacks using a compiler-directed voltage scaling technique is presented for a dynamic instruction schedule processor. In contrast, our target is a VLIW architecture and we focus on reducing leakage energy consumption. In [151], a combined strategy that exploits voltage scaling and leakage control is presented for an architecture with multiple versions of functional units (each operating with its own voltage). While the compiler algorithm in [151] works on a basic block granularity, our approach focuses on the entire control flow graph and determines paths along which leakage control can be used. This global scope provides more opportunities for applying more aggressive leakage control mechanisms and for longer durations. Also, instead of just integer ALU components, we measure the potential benefits of leakage control for all functional units.

4.2 Compiler Abstraction

Irrespective of the specific leakage control mechanism employed, we can abstract the following important phases when using leakage control:

- **Initiation latency, IL** — the time required for the sleep signal to propagate and logic state of the block to change.

- **Settling time, TS** — the time for the internal node voltages of the functional unit to move such that the steady state (the low leakage current state) is reached — this time is in the 10-100 ns range for both input vector control and power supply gating [145], and it depends strongly on the temperature and the amount of transistor leakage in the technology.

- **Low leakage time, T** — the time in idle state (between the time when steady state is reached and the time when re-activation occurs) when leakage energy is actually saved. It
is necessary for this duration to be large enough such that the leakage reduction achieved
during this period can amortize the additional energy-overhead for initiation and deactivation. In this work, the compiler is responsible for determining when and whether to apply leakage control based on anticipated savings.

- Reactivation latency, RL — the time required to propagate the reactivation signal and for the logic state of the functional unit to be restored. If reactivation is performed predictively (pre-activated) before an unit is required, we can eliminate the adverse impact on performance. Since leakage energy increases with operation time, we can also save energy due to pre-activation.

The values for IL, TS and RL depend on the specific mechanism. The IL and RL values for input vector control and supply gating mechanisms are given in Figure 3.3. We use a TS of 10ns in our experiments for both the leakage control mechanisms considered.

In the rest of the section, irrespective of the leakage control mechanism employed, we assume the existence of a sleep signal which can be activated and deactivated by the compiler (e.g., by setting/resetting a bit in one of the control registers). Since our default configuration has nine functional units, we need a nine bit space (in a control register) to implement this scheme.

When the sleep signal is set, the corresponding functional unit is placed into the leakage control mode. The magnitude of energy saving in this mode depends on the leakage control mechanism used. When the sleep signal is reset, the corresponding functional unit exits the leakage control mode and returns the normal operating mode. However, as explained above, it takes some amount of time and energy to transition from the leakage control mode to the normal operating mode, the magnitude of which again depends on the specific mechanism employed.
When our compiler analyzes the idle duration of functional units, it takes a conservative approach. What this means is that the compiler always turns on a functional unit when it is needed. It achieves this by placing activate instructions at each path that can be taken from the current point in the CFG. However, as we will see later, when profile data is used, it might be possible that a path which was not taken at all during profiling can be entered when the real input is fed to the application. In this case, a functional unit may need to be accessed before an activate instruction is executed. Consequently, we assume that a functional unit gets reactivated either by an explicit activate instruction or by any instruction that accesses it. In either case, we incur the corresponding reactivation latency and energy. For completeness, we also assume that an activate instruction on an already active functional unit or a deactivate instruction on an already deactivated unit does not have any effect. In our framework, such scenarios never occur. In the remainder of this section, we use the terms activating (deactivating) and turning on (turning off) interchangeably.

4.3 Data Flow Analysis for Leakage Energy Reduction

4.3.1 Background on Control Flow Graph

A basic block is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching except may be at the end [86]. A control flow graph (CFG) is a directed graph constructed by basic blocks and represents the flow-of-control information of the program.

For our purposes, the CFG can be thought of as a directed graph \( G = (V, E) \), where each \( v \in V \) represents a basic block, and each \( e \in E \) represents an edge between blocks. In
this section, we use the terms node and basic block interchangeably. Two unique nodes \( s \) and \( t \) denote the start and terminal nodes, respectively, of a CFG. One might think of these nodes as dummy statements. It is assumed that every node \( n \in V \) lies on a path from \( s \) to \( t \). We define the sets of all successors and predecessors of a node \( n \) as \( \text{succ}(n) = \{ m \mid (n, m) \in E \} \) and \( \text{pred}(n) = \{ m \mid (m, n) \in E \} \), respectively.

### 4.3.2 Background on Data-Flow Analysis

The objective of data-flow analysis is to provide global information about how a procedure (or a larger segment of a program) manipulates its data [86]. For example, constant-propagation analysis tries to determine whether at some particular point in the program all previous assignments to a particular variable along all possible paths produce a constant value for that variable. If this is the case, a use of the variable can be replaced by the constant at that point.

Traditionally, the most important data flow analysis problems include reaching definitions, available expressions, live variables, upwards exposed uses, copy propagation analysis, constant propagation analysis, and partial redundancy elimination [86]. Worklist algorithm is the most frequently used algorithm to solve these data flow analysis problems. In this algorithm, a set of equations are written for each basic block in the code and they are solved iteratively until a fixed point is reached. During the solution process, all unprocessed basic blocks are kept in a worklist. In general, data-flow algorithms are divided into two categories, namely, forward and backward analyses, depending on whether data-flow moves from top to bottom or from bottom to top, respectively.
4.3.3 Optimizing Leakage Energy Consumption

Our compiler-based leakage optimization strategy consists of the following two steps:

- Detecting the idleness of functional units for a given VLIW code. In this step, the compiler employs a data-flow analysis and delineates the paths along which a functional unit is not used.

- Based on the results of the data-flow analysis performed in the first step, the compiler inserts the sleep signal activation/deactivation instructions in the code. In doing this, the compiler must consider two important issues: (i) which leakage control mechanism should be selected? and (ii) where should pre-activation instructions be inserted (if pre-activation is required)?

4.3.3.1 Data-Flow Analysis for Detecting Idleness

Since a VLIW processor provides multiple functional units, which cannot be fully exploited except at the peak performance time with high instruction-level parallelism, in many execution cycles, we can find some idle functional units. We use backward data-flow analysis to collect the global (program-wide) information about the functional unit usage. If the compiler detects that a particular functional unit will not be used for some period of time, it can activate the sleep signal (of a leakage control mechanism) to turn off the unit. We formulate the idleness detection problem as a backward data-flow problem and solve it using a worklist algorithm.

Our data-flow analysis is, however, different from the traditional data-flow analysis in two aspects. First, in contrast to many data-flow optimizations, it is run after instruction scheduling has been performed. This is important as, in order to detect idleness accurately, we need to know
init(input: N ,output: USE)  
N: set of nodes in the CFG  
{}  
B: a node in the CFG  
for each B ∈ N do  
  for each functional unit j do  
    i=index(B);  
    compute USEi,j;  
    OFFi,j = 0;  
    ONi,j = 0;  
  end for  
end for  
}

Worklist-Algorithm(input: N,exit,USE, output: OFF,ON)  
N: set of nodes in the CFG  
extit: terminal node in the CFG  
{}  
B,S: nodes in the CFG  
Worklist: set of nodes  
effect_ON, total_effect_ON: set of indices  
effect_OFF, total_effect_OFF: set of indices  
call init(N,USEi,j);  
for the node exit  
  compute OFFi,j and ONi,j  
  using formulations (4.2) and (4.3)  
Worklist=N-exit;  
repeat  
  select a B from Worklist;  
  Worklist = Worklist - {B};  
  for each S ∈ succ(B) do  
    for each functional unit j do  
      total_effect_ON = ONi,j;  
      total_effect_OFF = OFFi,j;  
      compute OFFi,j and ONi,j  
      using formulations (4.4) and (4.5)  
      effect_ON = ONi,j;  
      effect_OFF = OFFi,j;  
      if ((effect_ON <> total_effect_ON)  
        or (effect_OFF <> total_effect_OFF)) then  
        total_effect_ON = effect_ON;  
        total_effect_OFF = effect_OFF;  
        Worklist = Worklist ∪ {B};  
      end if  
    end for  
  end for  
until Worklist is empty;  
}

Fig. 4.1. Data-flow (worklist) algorithm for computing OFFi,j and ONi,j.
the execution order of instructions. Second, as against many data-flow based optimizations, we aim at reducing energy consumption rather than reducing execution cycles. The input of our data flow algorithm is a control flow graph (CFG) which is obtained after a control-flow analysis [86]. Identifying basic blocks and building a CFG are standard compiler techniques and can be found in [86].

The formulation of our data-flow analysis can be described as the follows. Suppose that we have \( n \) functional units and that the functional unit usage is defined at the basic block level, instead of at the operation level. Obtaining the basic block level functional usage information is easy (in a VLIW architecture) as the scheduled code associates a functional unit with each operation in the basic block. Our compiler identifies each basic block using a number or id.

To build our data-flow equations, we use three different variables: \( USE_{i,j} \), \( OFF_{i,j} \), and \( ON_{i,j} \). Informally, \( USE_{i,j} \) tells us whether functional unit \( j \) is used by basic block \( i \). \( OFF_{i,j} \) and \( ON_{i,j} \), on the other hand, are two sets and keep the numbers (ids) of basic blocks that will contain deactivate and activate instructions, respectively.

More formally:

\[
USE_{i,j} = \begin{cases} 
0, & \text{if no operation in basic block } i \text{ uses functional unit } j \\
1, & \text{if at least one operation in basic block } i \text{ uses functional unit } j 
\end{cases}
\] (4.1)

Note that \( USE_{i,j} \) represents local (basic block level) functional unit usage information, and can be collected by analyzing each basic block separately and recording the functional units used (or not used) in that block. In a sense, \( USE_{i,j} \)'s are input to our data-flow system.
Formal definitions for $OFF_{i,j}$ and $ON_{i,j}$ depend on the position of the basic block $i$ in the CFG. If $i$ is the last basic block (the terminal basic block), then we have:

$$OFF_{i,j} = \begin{cases} \{i\}, & \text{if } USE_{i,j} = 0 \\ \emptyset, & \text{if } USE_{i,j} = 1 \end{cases} \quad (4.2)$$

and

$$ON_{i,j} = \begin{cases} \{i\}, & \text{if } USE_{i,j} = 1 \\ \emptyset, & \text{if } USE_{i,j} = 0 \end{cases} \quad (4.3)$$

On the other hand, for the remaining blocks in the CFG, we have:

$$OFF_{i,j} = \begin{cases} \{i\}, & \text{if } USE_{i,j} = 0 \\ \cup_{k \in \text{succ}(i)} OFF_{k,j}, & \text{if } USE_{i,j} = 1 \end{cases} \quad (4.4)$$

and

$$ON_{i,j} = \begin{cases} \{i\}, & \text{if } USE_{i,j} = 1 \\ \cup_{k \in \text{succ}(i)} ON_{k,j}, & \text{if } USE_{i,j} = 0 \end{cases} \quad (4.5)$$

In these formulations, $\cup$ denotes set union and $\cup_{k \in \text{succ}(i)} OFF_{k,j}$ ($\cup_{k \in \text{succ}(i)} ON_{k,j}$) indicates the union of all $OFF_{k,j}$ ($ON_{k,j}$) sets, where $k \in \text{succ}(i)$. Please see Section 4.3.1 for a formal description of $\text{succ}(\cdot)$.

Note that $OFF_{i,j}$ is a set of basic block number, in which we need to turn off functional unit $j$. Similarly, $ON_{i,j}$ is a set of basic block numbers, in which we need to turn the functional
Consider a non-terminal basic block \( i \). What we are trying to capture is whether the basic blocks that will be executed after \( i \) are using functional unit \( j \) or not. This information is important, because if there exists some basic block that uses \( j \) but basic block \( i \) itself does not use it, the basic block \( i \) should keep track of where (in which block) \( j \) should be reactivated if \( i \) turns it off.

The formulations (4.2) and (4.3) define the initial values for our data-flow system. Then, using a worklist algorithm, we work our way up in the control flow graph and compute \( OFF_i,j \) and \( ON_i,j \) for each basic block \( i \) and functional unit \( j \) using the formulations (4.4) and (4.5). Our data-flow algorithm that implements these formulations is given in Figure 4.1. Note that this algorithm computes \( OFF_i,j \) and \( ON_i,j \) for all \( i \) and \( j \) in a single invocation. In this algorithm, the Init() subroutine computes \( USE_i,j \) and initializes \( OFF_i,j \) and \( ON_i,j \) for each basic block \( B_i \) and functional unit \( j \). The Worklist-Algorithm() routine is where \( OFF_i,j \) and \( ON_i,j \) are computed iteratively. The repeat-until loop in this routine iterates as long as the sets \( OFF_i,j \) and \( ON_i,j \) are being updated.

To illustrate working of this algorithm, let us consider the control flow graph (CFG) shown in Figure 4.2(a). In this CFG, \( B_i \) represents basic block \( i \), where \( 1 \leq i \leq 8 \) and \( B_8 \) is the terminal block. To make illustration simple, we focus only on functional unit 0. In Figure 4.2(a), the value associated with basic block \( i \) represents \( USE_i,0 \). Recall that \( USE_i,0 = 0 \) means that the functional unit 0 is not used for any operation in the basic block \( i \), whereas \( USE_i,0 = 1 \) indicates that at least one operation in the basic block \( i \) uses the functional unit 0.

First, using the formulations (4.2) and (4.3), we find that

\[
OFF_{8,0} = \emptyset \text{ and } ON_{8,0} = \{8\}.
\]
Fig. 4.2. (a) A control flow graph (CFG) annotated with $USE_i,0$. (b) $OFF_i,0$ (left part) and $ON_i,0$ (right part) sets for each basic block. (c) The $(ON_i,0$ or $OFF_i,0$) sets that will be used for inserting activate/deactivate instructions. (d) Final placement of activate/deactivate (ON/OFF) instructions.
Before starting the iterative process for solving data-flow equations, $OFF_{i,0}$ $ON_{i,0}$ are set to $\emptyset$ for all $i \neq 8$. Then, using the formulations (4.4) and (4.5), in the first iteration of our worklist algorithm, we determine:

\[
\begin{align*}
OFF_{7,0} &= OFF_{8,0} = \emptyset \\
ON_{7,0} &= \{7\} \\
OFF_{5,0} &= \{5\} \\
ON_{5,0} &= ON_{7,0} \cup ON_{6,0} = \{7\} \\
OFF_{6,0} &= \{6\} \\
ON_{6,0} &= ON_{5,0} = \{7\} \\
OFF_{4,0} &= \{4\} \\
ON_{4,0} &= ON_{5,0} = \{7\} \\
OFF_{3,0} &= OFF_{8,0} = \emptyset \\
ON_{3,0} &= \{3\} \\
OFF_{2,0} &= OFF_{3,0} \cup OFF_{4,0} = \{4\} \\
ON_{2,0} &= \{2\} \\
OFF_{1,0} &= \{1\} \\
ON_{1,0} &= ON_{2,0} = \{2\}
\end{align*}
\]
In the second iteration of the algorithm, the $OFF_{i,0}$ and $ON_{i,0}$ sets do not change; so, we obtain the final values of these sets as shown in Figure 4.2(b). It should be noted that a similar analysis needs to be performed for other functional units as well.

4.3.3.2 Inserting Activate/Deactivate Instructions

In the second step of our approach, the compiler inserts activate/deactivate instructions in the code using the $OFF_{i,j}$ and $ON_{i,j}$ found in the first step.

We start by observing that, for each basic block $i$, if the functional unit $j$ is not used by $i$, it can be put in the leakage control mode at the beginning of $i$. However, the compiler needs to know at which basic blocks this functional unit should be reactivated. Similarly, if the functional unit $j$ is used by basic block $i$, the compiler needs to know at which basic blocks the functional unit can be turned off. More formally, if $USE_{i,j} = 0$, then $ON_{i,j}$ is used to find the basic blocks where the activate instructions for functional unit $j$ should be inserted. In a similar fashion, if $USE_{i,j} = 1$, then $OFF_{i,j}$ is used to find the basic blocks where the deactivate instructions for functional unit $(j)$ should be inserted.

It should be noted that the $ON_{i,j}$ and $OFF_{i,j}$ sets of a given basic block $i$ are not used to insert activate/deactivate instructions for that basic block itself. Instead, they are used to insert activate/deactivate instructions for other basic blocks that are control-dependent on that basic block. Returning to our example, the CFG in Figure 4.2(c) is the same as that in Figure 4.2(b) except that, for a given basic block $i$, it shows only the $OFF_{i,0}$ set if $USE_{i,j} = 1$. Similarly, if $USE_{i,j} = 0$, it only shows the $ON_{i,j}$ set. These are the sets that are actually used to insert activate/deactivate instructions. The only exception to this rule is the start block $(s)$ of the CFG. In this block, the content of the $USE_{s,j}$ is used to place activate or deactivate instruction.
Figure 4.2(d) shows the final placement of activate/deactivate instructions for our example. Since $USE_{1,j} = 0$, a deactivate instruction is inserted at the beginning of $B_1$. Similarly, since $ON_{1,0}$ is $\{2\}$, an activate instruction is inserted at the beginning of $B_2$. A deactivate instruction is inserted at the beginning of $B_4$ as $OFF_{2,0}$ is $\{4\}$. Since $ON_{4,0}$ is $\{7\}$, an activate instruction is inserted at the beginning of $B_7$. Other basic blocks are processed in a similar fashion. It should be mentioned, however, that once an activate (or deactivate) instruction is inserted at the beginning of a basic block, a second one is not inserted as it would serve no purpose.

We see from Figure 4.2(d) that the functional unit 0 can be kept deactivated during executions of basic blocks $B_1$, $B_4$, $B_5$, and $B_6$. In particular, in the loop composed of $B_5$ and $B_6$, large energy savings can be achieved if the iteration count of this loop is large.

4.3.3.3 Discussion

It should be noted, however, that this activate/deactivate instruction placement strategy can be more fine-granular than necessary. For example, in our example in Figure 4.2(d), if the execution time of basic block 1 is not much larger than the sum of initiation, reactivation latencies and settling time, very little energy saving can be achieved at the expense of some performance loss. This is because $B_2$, the only successor of $B_1$, needs to activate the functional unit immediately after $B_1$ has completed its execution. Consequently, it might be important to also take into account how much time a functional unit can remain in the idle state. For instance, in our example CFG, if the execution time of basic block 1 is not sufficiently large, we might want to prefer not to deactivate the functional unit 0 at the beginning of $B_1$.

To make such decisions, our compiler-based approach needs to be augmented with appropriate data about execution times of basic blocks in the CFG. This can be achieved using
either profiling or compiler analysis. In the profiling strategy, the code is instrumented in such a way that when this instrumented code is executed, we gather information about execution time of each basic block in the code. The second approach uses compiler to estimate the number of execution cycles each basic block would take when executed. This is possible as there is no conditional control flow between the instructions in a basic block. Assuming conservatively that each memory access will be a cache hit, we can statically estimate how much time a scheduled basic block would take. However, we also need to consider the flow of control between basic blocks. Consider the CFG fragment shown in Figure 4.3(a). In this fragment, we assume that a functional unit will be turned off in \( B_1 \) and will be turned on at basic blocks \( B_2, B_3, \) and \( B_4 \) that are not necessarily immediate successors to \( B_1 \). That is, there might be several basic blocks between \( B_1 \) and the other basic blocks shown in the figure.

For each of these three execution paths, we can take initiation and reactivation latencies, settling time and the path length (in cycles) into account and decide whether turning off the unit makes sense. If the results of these three decisions are the same, there is no problem; that is, we can easily decide whether to turn off the unit in \( B_1 \) or not. However, if some of the paths favors turning off while others do not, we need a conflict resolution scheme. One possible (conservative) solution in this case would be just not turning off the unit. A better solution, however, is to focus on the path which favors turning off and to insert a new basic block on this path (just after \( B_1 \)) with a deactivate instruction.\(^1\) This guarantees that the functional unit in question will be turned off only if the said path is taken during execution. Obviously, this strategy can be applied to each such path independently. It should be noted, however, in either

\(^1\)It should be noted that, in this case, it is not always correct to insert the deactivation instruction to the basic block that immediately follows \( B_1 \) on the path in question. This is because if this successor is reached from a basic block (other than \( B_1 \)), the deactivation command would be wrongly executed.
case, estimating the path length may not be trivial when the path contains a cycle (due to a loop). In such a case, if the loop bounds are known at compile-time, the compiler takes them into account and computes path length. Otherwise, it conservatively assumes that each loop on a path will iterate only once (if it can prove that the loop will be entered) or will not iterate at all (if cannot prove that the loop will be entered).

![Diagram of multi-path execution pattern and early-ON activation]

Fig. 4.3. (a) A multi-path execution pattern. (b) Pre-activating functional unit (early-ON).

Another issue that needs to be addressed is the impact of our strategy on performance (execution cycles). While deactivating otherwise idle units can lead to large energy savings, it may also cause some performance loss due to reactivation latency. While in some cases such increases in execution time can be tolerated, in many cases we would want to eliminate them if it is possible to do so. Let us consider the path from B1 to B2 shown in Figure 4.3(b), assuming that a functional unit will be turned off in B1 and will need to be reactivated in B2. To eliminate the associated reactivation latency, we propose a pre-activation strategy. In this strategy, instead of waiting till the execution thread comes to B2, we pre-activate the functional unit at an earlier
point in the path such that when the execution thread reaches B2, it finds the functional unit in
the normal operation mode. Selecting a suitable pre-activation (early-ON) point is important.
Our current strategy decides that by taking into account the estimated execution times of the
blocks on the paths. Specifically, assuming that the path from B1 to B2 in our current example
is \( B1 \rightarrow Bf_1 \rightarrow Bf_2 \ldots \rightarrow Bf_k \rightarrow B2 \), our approach selects a basic block \( Bf_r \) to pre-activate
the functional unit such that

\[
\sum_{i=r}^{k} T(Bf_i) \geq RL \quad \text{and} \quad \sum_{i=r+1}^{k} T(Bf_i) < RL
\]

are satisfied. In this formulation, \( RL \) is the reactivation latency and \( T(\cdot) \) gives the estimated
execution time of a basic block. That is, we try to pre-activate the functional unit ahead of time
\( (k - r + 1 \) blocks earlier to be specific) such that when B2 is reached, the functional unit is ready.

4.4 Experimental Framework

4.4.1 Implementation

We used Trimaran infrastructure [130] to implement our approach. Elcor is the back-end
compiler in Trimaran, which is responsible for machine-dependent optimizations and schedul-
ing. Our data-flow algorithm was implemented as an independent optimization module in Elcor
to be activated following the instruction scheduling pass. After performing instruction schedul-
ing, Elcor generates an output code sequence, which is represented as regions in Elcor. In this
code, for each operation, the functional unit to be used and execution start cycle (for that oper-
ation) are specified. This information is fed to our optimization module. Our module creates a
CFG based on the input region, and then performs our data-flow analysis as explained earlier.
Following this analysis, the activate/deactivate instructions are inserted in the code. The increase in static code size due to these extra (activate/deactivate) instructions were found to be negligible (less than 2%).

We also modified the interface between the Elcor and the simulator of Trimaran to pass the inserted activate/deactivate instructions from Elcor to the simulator. More specifically, a pseudo operation is inserted before each basic block in the generated code, which is then interpreted by the simulator. This pseudo operation contains detailed information for notifying the simulator when activation/deactivation occurs. The simulator in turn takes this information into account and generates output.

4.4.2 Benchmarks and VLIW Configuration

To test the effectiveness of our energy-saving optimization, we used four benchmarks from MediaBench suite (cordic, nbradar, raw caudio, and rawdaudio [76]) and two array-intensive applications (tomcatv and vpenta) from Spec benchmarks. For array-intensive benchmarks, we performed experiments with both integer arrays and double (precision) arrays to put different amounts of pressure on floating-point units. Important characteristics of our benchmarks are given in Figure 4.4.

Our VLIW configuration has a total of nine functional units: 4 integer ALUs, 2 floating-point ALUs, 2 load/store units, and 1 branch unit. All results have been obtained using a register file of 32 entries and a perfect cache configuration (that is, all cache accesses are assumed to be hits). The last two columns in Figure 4.4 give, for each benchmark, the number of execution cycles (a cycle time of 10ns was used) and functional unit leakage energy consumption (when no leakage optimization is applied). The energy numbers reported in this section are based on
0.1 micron technology, 1V supply voltage and 0.26V threshold voltage. Scaling factors [83] are applied from actual designs performed in 0.25 micron technology to obtain corresponding values for 0.10 micron. All energy saving numbers given in Section 4.5 are percentage improvements over the values in the last column of Figure 4.4.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Exec. Size (MB)</th>
<th>Number of Basic Blocks</th>
<th>Number of Cycles (Millions)</th>
<th>Energy (mJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cordic -</td>
<td>-</td>
<td>2.8</td>
<td>300</td>
<td>2.10</td>
<td>1.89</td>
</tr>
<tr>
<td>nbradar -</td>
<td>-</td>
<td>2.4</td>
<td>677</td>
<td>223.82</td>
<td>201.44</td>
</tr>
<tr>
<td>rawcaudio</td>
<td>clinton.pcm</td>
<td>2.6</td>
<td>130</td>
<td>7.43</td>
<td>6.68</td>
</tr>
<tr>
<td>rawdaudio</td>
<td>clinton.adpcm</td>
<td>2.5</td>
<td>123</td>
<td>5.71</td>
<td>5.14</td>
</tr>
<tr>
<td>tomcat v (int)</td>
<td>280KB</td>
<td>1.6</td>
<td>51</td>
<td>5.55</td>
<td>0.49</td>
</tr>
<tr>
<td>vpenta (int)</td>
<td>15.1MB</td>
<td>1.1</td>
<td>41</td>
<td>18.53</td>
<td>16.68</td>
</tr>
<tr>
<td>tomcat v (double)</td>
<td>560KB</td>
<td>1.7</td>
<td>51</td>
<td>10.75</td>
<td>14.67</td>
</tr>
<tr>
<td>vpenta (double)</td>
<td>30.2MB</td>
<td>1.3</td>
<td>41</td>
<td>25.26</td>
<td>22.74</td>
</tr>
</tbody>
</table>

Fig. 4.4. Benchmark characteristics.

4.4.3 Different Optimized Versions

For each benchmark code, we experimented with six different versions as explained below. The versions differ from each other with respect to the leakage control mechanism used, whether or not profile information is utilized (in determining idle paths), and whether or not pre-activation is employed. What we mean by profile data here is the profile data indicating the duration of idleness.
• **Input Vector Control:** This version uses only input vector control. It determines idleness duration using compiler estimation (i.e., it does not utilize profile data) and it does not employ pre-activation.

• **Supply Gating:** This version uses only power supply gating. As in the previous version, it determines idleness using compiler estimation (i.e., it does not utilize profile data) and it does not employ pre-activation.

• **Both (without pre-activation):** This version employs both input vector control and supply gating. Depending on the duration of idleness, it chooses one of these mechanisms. It does not use pre-activation or profile data.

• **Both (with pre-activation):** This is similar to the previous version except that it employs pre-activation.

• **Supply Gating (profile-based):** This version uses only power supply gating. It uses profile data for obtaining duration of idleness. It does not use pre-activation.

• **Both (profile-based):** This version uses both input vector control and supply gating. To determine duration of idleness, it exploits profile data. It does not utilize pre-activation.

In the third and fourth versions, when the compiler determines the duration of idleness, it compares that duration with the sum of activation, reactivation and settling times of input vector control and supply gating. It then chooses the method that would be most beneficial from the energy perspective by factoring in the leakage reduction factor. Consequently, it is possible that two different idleness periods (in the same functional unit) in different parts of the code can be optimized using different leakage control mechanisms. In the last two versions, the compiler first
identifies the paths where functional units are idle (using the algorithm in Figure 4.1). However, instead of trying to estimate the duration of idleness conservatively, it profiles the path and measures idleness duration for a typical execution. Based on this duration, it then decides whether to turn off the unit or not. While it might be possible to generate more versions, we believe that this combination enables us to observe general trends and make conclusive remarks. All these six versions have been implemented in the Trimaran infrastructure.

![Graph](image_url)

**Fig. 4.5.** % energy improvements (with floating-point units).

### 4.5 Results

Figure 4.5 gives the percentage leakage energy improvements for different optimized versions. It should be noted any increase in dynamic energy consumption and additional leakage consumed due to performance penalties resulting from our optimization strategies are also included in these results. We can make several observations from these results. First, the input
Fig. 4.6. % energy improvements (without floating-point units).

Fig. 4.7. % distribution of energy savings (nbradar).
vector control version performs very well. Specifically, it improves the functional unit leakage energy consumption by 45.4% on the average. The supply gating version, on the other hand, does not perform that well. In fact, in three applications (nbradar, rawcaudio, and rawdaudio), it hardly makes any difference in energy behavior. The main reason for this is the large re-activation time. Recall that in the supply gating version, the compiler detects the idleness and in computing the duration of idleness, the compiler is conservative. In many cases, the duration of idleness detected by the compiler is less than the sum of initiation and reactivation latencies and settling time; so, the compiler does not turn off the functional unit. When we combine these two leakage control mechanisms, we get an average improvement of 45.8%. This is only slightly better than the input vector control version. Again, the main reason for this is that in this combined version, input vector control dominates; that is, the compiler selects the input vector control in most cases due to the large re-activation time of the supply gating mechanism.

We also observe that including pre-activation did not bring too much benefit. This is again due to the fact that in most cases input vector control is used. Since the initiation and reactivation latencies of input vector control mechanism require a total of only 2 cycles (1 cycle latency for issuing the instruction to transition to leakage control mode and 1 cycle latency for issuing the instruction to transition to normal operating mode), pre-activation can only avoid additional leakage energy consumed during the reactivation cycle. As a result, the overall energy improvement is only slightly better than the case without pre-activation (45.8% versus 46.1%).

We also see that the profile-based supply gating generates much better results than the supply gating version. It generates an average energy reduction of 50.1%. This is because when we profile the code, we generally detect larger idle periods (e.g., we can detect the real execution time for loops instead of being conservative); as a result, the compiler uses power supply gating
more aggressively. Finally, the last version generates the best energy behavior (a 56.6% reduction in the functional unit energy consumption). It should be stressed, however, that a major part of this benefit comes from supply gating (thanks to longer idleness periods). This is in contrast to the combined case without profiling where input vector control was responsible for the bulk of energy improvements.

Not all of our benchmarks use the floating-point units. Therefore, including floating-point units might tend to exaggerate our energy savings. To address this, we performed another set of experiments where only the savings in integer ALUs, load/store units, and branch unit are considered. The results given in Figure 4.6 indicate that even in this case, energy savings are significant, the average (across all benchmarks) savings varying between 3.6% (the supply gating version) and 48.2% (the combined version with profiling).

To better understand where these energy savings are coming from, we present in Figure 4.7, the distribution of energy savings across different functional units in our VLIW. We focus only on nradar as the trends in other benchmarks are similar. We observe from these results that almost all functional units benefit from leakage control. The first integer ALU gets the least benefit as it is heavily utilized. The benefits for the remaining three integer ALUs are around 51-52% on the average (across all six optimized versions). The first floating-point (FP) ALU achieves an average saving of 11.8%, whereas the second one achieves 66.1%. The average savings for the load/store units, on the other hand, are 63.2% and 67.7%. Finally, the branch unit saves 54.3% energy on the average. These results clearly indicate that, while actual savings might depend on the specific optimized version used, it is important to apply leakage control to as many functional units as possible.
To evaluate the impact of pre-activation on energy savings and see whether it would be more useful with a larger initiation and re-activation latencies, we performed some experiments assuming per cycle energy savings similar to input vector control but with a hypothetical initiation (and re-activation) latency of 45 cycles (instead of 2 cycles). We see from the results given in Figure 4.8 that when the activation latency is larger, pre-activation starts to make a difference. The average energy savings with and without pre-activation are 18.7% and 16.3%, respectively.

![Figure 4.8](image_url)

Fig. 4.8. % energy savings with an reactivation latency of 45 cycles.

Figure 4.9 shows the percentage increase in execution time when different optimized versions are used. We observe that, except for cordic, the increase in execution cycles is bounded by 18%, and less than 10% in many cases. Since when we use pre-activation, the increase in execution time is eliminated, the Both (with pre-activation) version is not explicitly shown in this figure.
In our final set of experiments, we measured the sensitivity of our energy savings to the VLIW configuration. We focused on rawcaudio and conducted experiments with seven different VLIW configurations using the Both (without pre-activation) version.

In Figure 4.10, each configuration is denoted using the tuple:

(# of integer ALUs, # of floating-point ALUs, # of load/store units, # of branch units).

The first configuration on the x-axis (that is, (4,2,2,1)) corresponds to our default configuration. We see from these results that our optimization strategy is beneficial for all these VLIW configurations. The percentage energy improvements range from 27.2% to 45.8%, averaging on 36.8%. Based on these results, we encourage compiler writers to spend time in developing leakage energy optimization techniques for their VLIW architectures.
Fig. 4.10. % energy savings with different VLIW configurations (rawcaudio).
Chapter 5

Compiler-Directed Instruction Cache
Leakage Energy Optimization

5.1 Introduction

With the increasing number of transistors employed in current microprocessors and the continued reduction in threshold voltages of these transistors, leakage energy consumption has become a major concern [14]. As dense cache memories constitute a significant portion of the transistor budget of current microprocessors, leakage optimization for cache memories is of particular importance. It has been estimated that leakage energy accounts for 30% of L1 cache energy and 70% of L2 cache energy for a 0.13 micron process [97].

There have been several efforts [72, 88, 132, 62, 97, 152] spanning from the circuit level to the architectural level at reducing the cache leakage energy. Circuit mechanisms include adaptive substrate biasing, dynamic supply scaling and supply gating. Many of the circuit techniques have been exploited at the architectural level to control leakage at the cache bank and cache line granularities. The supply gating mechanism was applied at bank level granularity in [97] to dynamically vary the size of the active portion of the cache. The cache miss rates were used to adapt the cache sizes in order to reduce leakage power consumption. The supply gating mechanism was employed at the finer granularity of cache line in [63]. This technique monitors the periods of inactivity in cache lines by associating saturating counters with them. In [152], only the data array of a cache is placed in a low power mode while the tag array is still in active
mode. This helps to dynamically adjust the turn off interval to ensure that performance closely tracks the performance of an equivalent cache without sleep mode. Another approach to leakage control at the cache line granularity involves the reduction of supply voltages to idle cache lines [35]. Specifically, all cache lines are periodically placed in a leakage-controlled mode by scaling down their supply voltage. This implementation also chooses higher threshold voltages for the access transistors to minimize the bitline leakage. In contrast to other approaches, Heo et al. [43] focus on reducing bitline leakage by leaving bitlines of banks that are not accessed open.

Most prior approaches have focused on utilizing hardware monitoring to manage the leakage-control modes of the caches. These techniques transition to leakage control modes after fixed periods or fixed periods of inactivity. They incur the energy penalty for decaying to the low leakage mode only after fixed periods. The approaches that dynamically change the turn-off periods attempt to address this problem. In contrast to these hardware-centric approaches, in this work, we propose a compiler-based leakage optimization strategy for instruction caches. This approach identifies the last use of the instructions and places the corresponding cache lines that contain them into a low leakage mode. The idea of instruction-based leakage control was suggested in [63] for data caches based on profiling. Their work also identified the need for compiler analysis in such an instruction-based approach. In this work, we present and analyze two compiler-based strategies termed as conservative and optimistic. The conservative approach does not put a cache line into a low leakage mode until it is certain that the current instruction in it is dead. On the other hand, the optimistic approach places a cache line in low leakage mode if it detects that the next access to the instruction will occur only after a long gap.

A compiler-based leakage optimization strategy such as ours makes sense in a VLIW environment (which is the focus of our work) where the compiler has control of instruction
execution order. Using the Trimaran infrastructure [130], we demonstrate that it is possible to significantly reduce instruction cache leakage energy using compiler analysis.

5.2 Our Approach

5.2.1 Circuit Support

We rely on the dynamic scaling of the supply voltages to reduce the leakage current in the cache memories. As supply voltage to the cache cells reduces, the leakage current reduces significantly due to short-channel effects. The choice of the supply voltage influences whether the data is retained or not. When the normal supply voltage of 1.0V is reduced below 0.3V (for a 0.07 micron process), we observe that the data in the cells are no longer retained. Thus, we select a 0.3V supply voltage for the state-preserving leakage control mode. However, if state preservation is not a consideration, we switch the supply voltage to 0V to gain more reduction in energy. Except for our hybrid scheme (discussed in Section 5.4) that requires dynamic selection between data-preserving and data-destroying modes, we use a similar circuit to that proposed in [35]. Each cache line is augmented with a power status bit that is used to control the appropriate voltage selection for the cell. A global control signal is used to set the power status and, consequently, set the voltages of all cache lines to 0.3V (0.0V) to place them in a state-preserving (state-destroying) leakage control mode. Whenever a cache line is accessed, its supply voltage is first switched to the normal voltage of 1.0V before access is permitted. This is achieved by using the wordline trigger to reset the power status bit and by preventing the access until the supply voltage settles by gating the wordline. The gating must be performed as data can be corrupted.
when accessing the cache when the supply voltage is low. In our experiments, all cache lines are in the leakage-control mode before their first use for all strategies.

For the hybrid scheme, we augment this circuit as shown in Figure 5.1 to dynamically transition between active, state-preserving and state-destroying modes. The power supply to the cache lines are set to 1.0V, 0.3V or 0V, respectively, for the three modes when using caches designed with 0.07 micron Berkeley predictive technology [5]. Each cache line has a two-bit power status register indicating the mode (00-Active; 01- State-Preserving; 11 - State-Destroying) in which it is placed. There are two global control signals (Set0, Set1) for changing the states. When a cache line in either state-preserving or state-destroying mode is accessed, the access is delayed until the supply voltage recovers to 1.0V. When an access occurs, the status register bits are automatically set to zero. There are two special instructions that are used to place the cache lines into a state-preserving mode or state-destroying mode. The least significant bits (B0) of all the power status registers are globally set when the state-preserving transition instruction is executed. Note that this permits all cache lines in a state-destroying mode to remain in that mode even when the state-preserving instruction is executed. Similarly, the two bits (B0 and B1) of all the power status registers are set when the state-destroying transition instruction is executed.

It must be observed that our approach relies on a specific instruction to place cache lines in state-preserving or state-destroying mode. This is in contrast to the approach used in [35] where a periodic timer is used to place all cache lines in a state-preserving (drowsy) mode. We refer to the variants of the scheme proposed in [35] as Kill-M (where the content of the cache line is destroyed) and Drowsy-M (where the content of the cache line is preserved). Here, M is the periodic timer interval in cycles. In Section 3.4, we present a detailed comparison of
our compiler-based strategies with Kill-M and Drowsy-M. In this section, we refer to strategies Kill-M and Drowsy-M as the fixed period strategies.

![Leakage control circuitry](image)

**Fig. 5.1.** Leakage control circuitry.

The access transistors in our SRAM cells use a higher threshold voltage of 0.3V as compared to the other SRAM transistors that use a threshold voltage of 0.2V. This is performed to keep the contribution of the bitline leakage to a minimum.

### 5.2.2 Compiler Support

In order to exploit the state-destroying and state-preserving leakage control mechanisms explained above, our compiler implements two different approaches for turning off instruction cache lines. The first approach, called the conservative strategy, does not turn off an instruction
Fig. 5.2. A code fragment that contains three loops.

cache line unless it knows for sure that the current instruction that resides in that line is dead. The second approach is called the optimistic strategy and turns off a cache line even if the current instruction instance in it is not dead yet. This might be a viable option if there is a large gap between two successive visits to the cache line.

The conservative strategy is based on determining the last usage of instructions. Once this last use is detected, the corresponding cache line can be turned off. While it is possible to turn off the cache line immediately after the last use, such a strategy would not be very effective, because it would result in significant code expansion due to the large number of turn-off instructions inserted in the code. Also, such frequent turn-off instructions themselves would consume considerable dynamic energy. Consequently, in this work, we turn off instructions at the loop granularity level (We treat the streamline code as a loop which iterates once). More specifically, when we exit a loop and we know for sure that this loop will not be visited again, we turn off the cache lines that hold the instructions belonging to the loop. While ideally we would want to issue turn-offs only for the cache lines that hold the instructions in the loop, identifying
these cache lines is costly (i.e., it either requires some type of circuit support which itself would consume energy, or a software support which would be very slow). As a result, in this work, when we exit the loop, we turn off all cache lines.

The idea behind the conservative strategy is illustrated in Figure 5.2 for a case that contains three loops, two of which are nested within the third one. Assume that once the outer loop is exited, this fragment is not visited again during execution. Here, Loop Body-I and Loop Body-II refer to the loop bodies of the inner loops. In the conservative strategy, when we exit Loop Body-I, we cannot turn off the cache lines occupied by it; because, there is an outer loop that will re-visit this loop body; However, when we exit the outer loop, the conservative strategy turns off all the cache lines that hold the instructions of this code fragment (i.e., all instructions in all three loops). As mentioned above, we in fact turn off all the cache lines for implementation efficiency. It is clear that this strategy may not perform well if there is a large outermost loop that encloses a majority of the instructions in the code. In such cases, the cache lines occupied by the said instructions will not be turned off until the outermost loop finishes its execution. And, when this occurs, it might be too late to save any leakage energy.

The optimistic strategy tries to remedy this drawback of the conservative scheme by turning off the cache lines optimistically. What we mean by optimism here is that the cache lines are turned off even if we know that the corresponding instruction instance(s) will be visited again, but the hope is that the gap (in cycles) between successive executions of a given instruction is large enough so that significant amount of energy can be saved. Obviously, an important question here is how to make sure at compile time (i.e., statically) that there will be a large gap between successive executions of the same instruction. Here, as in the conservative case, we work on a loop granularity. When we exit a loop, we turn off the instructions in the loop body if either
that loop will not be visited again (as in the conservative case) or the loop will be re-visited but there will be execution of another loop between the last and the next visit. Returning to the code fragment in Figure 5.2, when we exit Loop Body-I, we turn off the instructions in it. This is because before Loop Body-I is visited again, the execution should proceed with another loop (the one with Loop Body-II), and we optimistically assume that this latter loop will take long time to finish.\footnote{While a more sophisticated approach would employ profile data to check whether that loop really takes long time, in our current implementation we do not perform such checks. Instead, a reliance is placed upon the observation that most loops (even those in non-array applications) take a long time to execute.} Similarly, when we exit Loop Body-II, we turn off the corresponding instructions. Obviously, this strategy is more aggressive (in turning off the cache lines) than the conservative strategy. The downside is that in each iteration of the outer loop in Figure 5.2, we need to re-activate the cache lines that hold Loop Body-I and Loop Body-II. The energy overhead of such a re-activation depends on the leakage saving mode employed. Also, since each reactivation incurs a performance penalty, the overall execution time impact due to the optimistic strategy can be expected to be much higher than that due to the conservative strategy.

### 5.2.3 Alternative Strategies

Since we have two different compiler strategies (conservative and optimistic) and two different leakage saving mechanisms (state-preserving and state-destroying), clearly, we have four different implementation choices. These choices are summarized in Figure 5.3. Among the choices we have, Strategy IV does not make much sense since being conservative means that we do not turn off cache lines unless we are sure that the instructions are dead. Therefore, there is not much point in employing a state-preserving leakage control mechanism. Consequently, in
the rest of this section, we focus only on the remaining three strategies: I, II, and III, and compare them with fixed period strategies Kill-M and Drowsy-M [35]. Note that while one can select the best M value for a given application, it is possible that each application (and even different parts of the same application) demands a different M value. In contrast to fixed period strategies, our compiler strategies can automatically tune the turn-off periods within different phases of the program and also based on the different characteristics of each program. Furthermore, the compiler strategies can even select the appropriate low-leakage mode if there is underlying circuit support.

<table>
<thead>
<tr>
<th></th>
<th>Conservative</th>
<th>Optimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-Destroying</td>
<td>Strategy I</td>
<td>Strategy II</td>
</tr>
<tr>
<td>State-Preserving</td>
<td>Strategy IV</td>
<td>Strategy III</td>
</tr>
</tbody>
</table>

Fig. 5.3. Four different implementation choices depending on the leakage control mechanism (mode) used and the compiler strategy employed.

5.3 Experiments

5.3.1 Benchmarks and Simulation Platform

We target improving leakage energy consumption of the instruction cache in a state-of-the-art VLIW processor. The results reported on here are obtained using a Trimaran-based compiler/simulation infrastructure. Trimaran provides a vehicle for implementation and experimentation in state-of-the-art research in compiler techniques for instruction level parallelism...
A program flows through IMPACT, Elcor, and the cycle-level simulator. IMPACT applies machine-independent classical optimizations and transformations to the source program, whereas Elcor is responsible for machine-dependent optimizations and scheduling. Our conservative and optimistic algorithms are implemented in Elcor, and after all other optimizations have been performed. The increase in compilation time due to our algorithms was around 15% on average (when all benchmark codes are considered). Further, the increase in code size due to the inserted turn-off instructions is less than 5% across all benchmarks and strategies. The cycle-level simulator was augmented with a cache model and modified to recognize the power-mode control instructions for changing the supply voltages to the cache lines. The VLIW configuration used in our experiments has four IALUs (integer ALUs), two FPALUs (floating-point ALUs), one LD/ST (load/store) unit and one branch unit. Other system parameters used for our default setting are provided in Figure 5.4. The energy values reported are based on circuit simulation. In our evaluations, we performed experiments with both basic block scheduling [86] and superblock scheduling [23]; since we did not observe too much difference in trends, we report here only the basic block based scheduling results.

To evaluate the effectiveness of our algorithms, we used a suite of ten programs from different benchmark sets. The salient characteristics of these codes are given in Figure 5.5. The benchmark source is indicated in the second column. The third column in this figure gives the number of code lines and the fourth column gives the input used for running the benchmark. The total execution cycles and the original instruction cache energy consumption are provided in the last two columns. In selecting these programs, we paid attention to ensure diversity. Compress and li are integer codes with mostly irregular access patterns. idea, mpeg2dec, polyphase,
Fig. 5.4. Default parameters used in our simulations.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature size</td>
<td>0.07 micron</td>
</tr>
<tr>
<td>Supply voltage</td>
<td>1.0V</td>
</tr>
<tr>
<td>L1 instruction cache</td>
<td>16KB direct-mapped cache</td>
</tr>
<tr>
<td>L1 instruction cache latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>32KB 2-way cache</td>
</tr>
<tr>
<td>L1 data cache latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Unified L2 cache</td>
<td>512KB 4-way cache</td>
</tr>
<tr>
<td>L2 cache latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Memory latency</td>
<td>100 cycles</td>
</tr>
<tr>
<td>Clock speed</td>
<td>1 GHz</td>
</tr>
<tr>
<td>L1 cache line size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>L2 cache line size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 cache line leakage energy</td>
<td>0.33 pJ/cycle</td>
</tr>
<tr>
<td>L1 state-preserving mode cache line leakage energy</td>
<td>0.01 pJ/cycle</td>
</tr>
<tr>
<td>L1 state-destroying mode cache line leakage energy</td>
<td>0.00 pJ/cycle</td>
</tr>
<tr>
<td>L1 state-transition (dynamic) energy</td>
<td>2.4 pJ/transition</td>
</tr>
<tr>
<td>L1 state-transition latency from state-preserving mode</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L1 state-transition latency from state-destroying mode</td>
<td>1 cycle (excluding miss latency)</td>
</tr>
<tr>
<td>L1 dynamic energy per access</td>
<td>0.11nJ</td>
</tr>
<tr>
<td>L2 dynamic energy per access</td>
<td>0.58nJ</td>
</tr>
</tbody>
</table>
and rawdaudio are typical media applications. The last three benchmarks (adi, btrix, vpenta) and paraffins, on the other hand, represent array-intensive applications.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Lines</th>
<th>Input</th>
<th>Execution Cycles</th>
<th>I-cache Energy(nJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>129.compress</td>
<td>SpecInt95</td>
<td>1939</td>
<td>test.in</td>
<td>42784111</td>
<td>13627628 (53%)</td>
</tr>
<tr>
<td>139.li</td>
<td>SpecInt95</td>
<td>7597</td>
<td>train.lsp</td>
<td>918252701</td>
<td>230649411 (67%)</td>
</tr>
<tr>
<td>idea</td>
<td>Mediabench</td>
<td>1232</td>
<td>/</td>
<td>335180</td>
<td>97343 (58%)</td>
</tr>
<tr>
<td>mpeg2dec</td>
<td>Mediabench</td>
<td>9832</td>
<td>me16v2.m2v</td>
<td>140735320</td>
<td>46702867 (51%)</td>
</tr>
<tr>
<td>paraffins</td>
<td>Trimaran</td>
<td>388</td>
<td>/</td>
<td>523363</td>
<td>111058 (79%)</td>
</tr>
<tr>
<td>polyphase</td>
<td>Mediabench</td>
<td>542</td>
<td>polyphase.IN</td>
<td>587442</td>
<td>181888 (54%)</td>
</tr>
<tr>
<td>rawdaudio</td>
<td>Mediabench</td>
<td>314</td>
<td>clinton.adpcm</td>
<td>7479483</td>
<td>2870793 (44%)</td>
</tr>
<tr>
<td>adi</td>
<td>Livermore</td>
<td>46</td>
<td>274.68MB</td>
<td>1490229</td>
<td>435725 (58%)</td>
</tr>
<tr>
<td>btrix</td>
<td>Specfp92</td>
<td>135</td>
<td>202.53MB</td>
<td>27056699</td>
<td>6337571 (72%)</td>
</tr>
<tr>
<td>vpenta</td>
<td>Specfp92</td>
<td>114</td>
<td>14.42MB</td>
<td>141445594</td>
<td>29645742 (81%)</td>
</tr>
</tbody>
</table>

Fig. 5.5. Benchmark codes used in our evaluations. The last column also contains the percentage contribution of leakage to overall instruction cache energy. Note that no leakage control mechanism is employed in obtaining this data.

Note that our focus in this section is on optimizing the leakage energy consumed in the instruction cache. In doing so, however, our strategies can also incur several energy (and performance) overheads. For example, there is a dynamic energy overhead in the instruction cache due to turning on/off a cache line placed into a leakage-control mode. Also, there is a dynamic energy overhead due to executing turn-off instructions. Since some of our strategies increase execution cycles, the extra leakage energy consumption might also be an issue. In our presentation, where significant, we quantify these overheads to illustrate the energy behavior at larger level (not just in the instruction cache). In the rest of this section, when we mention energy we mean the leakage energy consumed by the instruction cache plus any extra (dynamic) energy.
that occurs as a result of cache line turn-offs/ons and due to any additional L1 instruction cache accesses. This extra energy might be important as some of the strategies evaluated here can incur large performance penalties and significant number of cache line turn-ons. As mentioned earlier, we also compare our optimization strategies with fixed period strategies: Kill-M and Drowsy-M; we experiment with two M values: 2K and 4K.

5.3.2 Cache Life-Time Analysis

![Chart showing the percentage of times cache lines are in leakage control mode for different strategies.](chart.png)

Fig. 5.6. Percentage of times where cache lines are in leakage control mode.

We present in Figure 5.6 the percentage time that cache lines spend in leakage control mode for different optimization strategies. One thing to note in this graph is that, for some versions, this time is very high. Specifically, the percentage of time that cache lines are in a leakage control mode for Strategy-I, Strategy-II, Strategy-III, Kill-2K, Drowsy-2K, Kill-4K, and Drowsy-4K are 40.83%, 85.61%, 86.21%, 86.10%, 79.83%, 81.61%, and 77.09%, respectively.
Since only a hypothetical optimal strategy would achieve 100% execution time spent in the leakage control mode, we believe that some of our strategies perform really well in placing cache lines into leakage control modes.

5.3.3 Energy and Performance Results

![Normalized energy consumption graph](image)

Fig. 5.7. Normalized (w.r.t. Strategy I) energy consumption with different leakage saving strategies.

Figure 5.7 shows the energy consumptions of the strategies (given as fractions of the energy consumption of Strategy I). Note that though not given in the graph, Strategy I improves the energy consumption of the original code (without any leakage control but discounting the leakage of unused cache lines) by 40% on an average (ranging from 1% for mpeg2dec to 84% for btrix). One can make several observations from this figure. First, Strategy I does not perform well as compared to other optimization strategies. In fact, it generates the worst results in most benchmarks. Second, one can see that Strategy III and Strategy II (in some cases) generate very
good energy results. In fact, in 8 of our 10 benchmarks one of these two strategies provide the best energy consumption. Third, in some benchmarks, most notably vpenta, the fixed period strategies generate the best energy behavior. Now, let us try to explain the behavior of these different strategies, starting with our strategies. We can measure the magnitude of energy benefits of a given optimization strategy considering three factors: (1) how soon it turns off cache lines; (2) what leakage control mechanism it uses; (3) whether the energy overheads overwhelm the potential leakage savings. Strategy I does not perform well because it acts very late in turning off cache lines. While Strategy II turns off cache lines quickly and employs state destroying mode to provide significant reduction in leakage energy, the energy cost of frequent extra writes to the instruction cache (when the instructions need to be fetched again from L2) nullifies this benefit in most of the benchmarks. Strategy III also turns off cache lines quickly (after each inner loop); however, it uses state-preserving leakage mode. In contrast to Strategy II, this scheme does not have the additional overhead of instruction cache misses. Moreover, Strategy III can obtain most of the leakage savings provided by Strategy II (due to the small difference in their leakage values).

We turn our attention now to fixed period schemes. Their energy behavior compared to our strategies depends to a large extent on the execution time of the loops. For example, if a given loop takes too long to finish, even our Strategy II and Strategy III will not achieve very good results as it will take a long time before they can turn off the corresponding cache lines. However, the fixed period strategies can turn off the cache lines in such a loop (maybe several times depending on the execution time).

Figure 5.8 and figure 5.9 divides the energy consumption in the instruction cache into different components for Strategies II and III. For Strategy II, each bar is divided into three parts:
the leakage energy consumed during normal operation, the dynamic energy incurred in transition-
ing to and from state-destroying mode, and the dynamic energy consumed in the instruction
cache due to additional cache misses. We see that in most of the benchmarks, the dynamic
energy overhead due to extra misses constitutes a large percentage of the energy consumption,
which explains the poor behavior of this strategy. Also, note that the leakage energy consumed
in state-destroying mode is zero. For Strategy III, each bar is divided into three parts: the leakage
energy consumed during normal operation, the dynamic energy incurred in transitioning to and
from state-preserving mode, and the leakage energy consumed in the state-preserving mode. It
should be observed that the state-transition overhead is small and a considerable portion of the
energy is expended when cache lines are in state-preserving mode.

![Energy Breakdown for Strategy II](image)

**Fig. 5.8.** Energy breakdown for Strategy II

We also need to emphasize that the strategies that kill the data in cache lines prema-
turely (i.e., before the corresponding instructions are dead) can also cause extra dynamic energy
consumption in L2 cache and off-chip memory. Although in this work we focus on on-chip energy consumption, it is also important to know the magnitude of this off-chip energy. Figure 5.10 gives the extra dynamic energy consumption in the off-chip L2 and memory. It should be seen that among our strategies only Strategy II can cause extra off-chip energy consumption. This is because Strategy I kills the contents of a cache line if and only if it is already dead and Strategy III only employs state-preserving mode. Therefore, Strategy III becomes even more preferable when considering off-chip L2 and memory energy.

Obviously, energy behavior is only part of the picture. To have a fair evaluation of all strategies considered, we need to look at their performance behavior as well. The normalized execution cycles with our base configuration are presented in Figure 5.11. All values are normalized with respect to that of Strategy II as it is the one which takes the longest time in most of the cases (It should be noted that the performance degradation of Strategy I is within 1% of the original case). Two factors influencing the performance penalty are the number of cache lines turned on and the number of cycles spent per turn-on. The second factor is dependent on

![Fig. 5.9. Energy breakdown for Strategy III](image-url)
Fig. 5.10. Extra dynamic energy consumption in off-chip L2 and memory. Only the strategies that employ state-destroying mode incur this penalty.

Fig. 5.11. Normalized (w.r.t. Strategy II) number of execution cycles with different leakage saving strategies.
whether the cache line was in state-preserving or state-destroying mode before turn on. On the average, the number of turn-ons for Strategy I, Strategy II, Strategy III, Kill-2K, Drowsy-2K, Kill-4K and Drowsy-4K are 1448, 22777521, 22777521, 14096265, 10968055, 10086726, and 7428348, respectively. The number of turn-ons in our schemes is typically larger than that of the fixed period schemes (except Strategy I). Note that the performance penalty for the same number of turn-ons for Strategy II is much larger than that of Strategy III due to L2 access latencies. This clearly shows the tradeoff between energy savings and performance overhead.

The energy-delay product helps to balance the benefits in energy savings with any potential degradation in performance. Figure 5.12 shows the normalized energy-delay products for our applications. We see that Strategy III is very successful. This is because in many cases its percentage energy benefits are higher than its performance losses, and also it strikes a good balance between performance and energy. We observe that the average normalized energy-delay
product for Strategy III is 0.47 which is 13% better than that of the best fixed-period scheme (Drowsy-2K - 0.54) for the considered applications.

5.4 Hybrid Strategy

Our leakage optimizations evaluated so far use either state-preserving mechanism or state-destroying mechanism exclusively. It is also possible to employ these two mechanisms under the same optimization strategy. That is, a leakage optimization strategy can use these mechanisms selectively. In this section, we discuss such a hybrid strategy and quantify its capability of saving leakage.

Our hybrid strategy proceeds as follows. When exiting a loop, if this loop is not going to be accessed again, the hybrid strategy turns off the associated cache lines using the state-destroying mechanism. On the other hand, if the loop will be visited again, it just places the cache lines into leakage control mode using the state-preserving strategy. Later in execution, when this loop finishes its last execution, the cache lines are turned off via the state-destroying mechanism.

The energy-delay product profile of the hybrid strategy is illustrated in Figure 5.13. Each bar is normalized with respect to the strategy that generated the best (excluding hybrid) energy-delay result (as far as that benchmark is concerned). One can observe from these results that in three benchmarks (compress, li, and mpeg2dec), hybrid and Strategy III generate the same energy-delay product. In polyphase and vpenta, hybrid is outperformed by Kill-4K and Drowsy-4K. However, in the remaining five benchmarks, hybrid generates the best results. Note that the hybrid scheme, however, involves extra circuit overheads for the additional supply voltage to choose dynamically between state-destroying and state-preserving modes. Strategy III performs
well considering that the overhead of the additional (third) supply voltage distribution was not factored in the evaluation of the hybrid scheme. All other schemes discussed so far only use two supply voltages per cache line.

![Fig. 5.13. Normalized energy-delay product for the hybrid strategy.](image)

### 5.5 Impact of Compiler Optimizations

While evaluating a given leakage control mechanism, it is also critical to quantify its behavior under different code optimizations. This is important not only because many compiler optimizations (especially those targeting at improving data locality) can modify the instruction execution order (sequence) dramatically leading to significantly different energy picture, but also because if we can characterize the impact of such optimizations on the effectiveness of the proposed mechanism, this information can be fed-back to compiler writers, leading to better (e.g., energy-aware) compilation strategies.
In this section, we first give a qualitative assessment of two frequently-used loop transformation strategies, loop fission (distribution) and loop fusion. The loop distribution transformation cuts the body of a for-loop statement in two parts [138].

As an example, let us consider the fragment shown in Figure 5.14(a). If we distribute the outermost loop over the two groups of statements (denoted Body-I and Body-II in the figure), we obtain the fragment depicted in Figure 5.14(b). Figures 5.14(c) and (d), on the other hand, illustrate how the instructions in the fragments in Figures 5.14(a) and (b), respectively, would map to the instruction cache. In Figures 5.14(c) and (d), Header is the loop control code. Note that in the distributed version, Header is duplicated. Now, let us try to understand how this optimization would influence the effectiveness of our leakage optimization strategies. First, let us focus on Figure 5.14(c). During execution all three blocks (Header, Body-I and Body-II) need to be accessed very frequently, and there will be little opportunity (or energy benefit) in placing the cache lines in question into leakage control mode. If we consider the picture in Figure 5.14(d), on the other hand, when we are executing the first loop only the first Header and Body-I need to be activated. The second Header and Body-II can be kept in a leakage saving mode. Similarly, when we move to the second loop, during execution, only the second Header and Body-II need to be activated. Therefore, at any given time, the distributed alternative leads to the activation of fewer cache lines. However, the number of cache lines occupied by the code is one part of the big picture. Since we are focusing on the leakage energy consumption, we also need to consider the execution time. If, in this code fragment, data cache locality is a problem, then the first alternative (without distribution) might have shorter execution time if loop distribution destroys data cache locality. Consequently, although the alternative in Figure 5.14(d) will occupy fewer cache lines at a given time, it will keep those cache lines in the active mode
for a longer duration of time. Consequently, there is a tradeoff here between the number of cache lines occupied and the time duration during which they are active.

A similar tradeoff exists when we consider another loop-level optimization: loop fusion. This optimization is the reverse of loop distribution. Specifically, it takes two neighboring loops and combines their loop bodies into a single loop (e.g., going from the code in Figure 5.14(b) to the code in Figure 5.14(a)). It is generally used for enhancing data cache locality by bringing the statements that access the same set of data to the same loop [138]. In our context, applying this optimization will increase the number of cache lines active at a given time. On the other hand, it might also reduce the time duration during which these cache lines are active.

In fact, the preceding discussion can be generalized to other data locality optimizations as well. Many optimizations that target at enhancing data cache performance increase code size (i.e., they reduce instruction reuse). Consequently, during the course of execution, at any given time, larger number of cache lines will be active (as a result of the optimization). However,
if successful, these optimizations will also reduce the number of execution cycles (hence, the
cycles in which the cache lines are active). Iteration space tiling [138, 74] is a typical example
of that. In tiling, a loop is broken into two loops and the order of accesses within the array
is modified. In most cases, this also leads to a larger code size and reduced instruction reuse.
In this section, we evaluate the impact of several data locality-oriented compiler optimizations
using two of our applications.

![Fig. 5.15. Instruction cache leakage energy impact of optimizations on vpenta. All values are
normalized w.r.t. to the case without power management.](image)

Figures 5.15 and 5.16 show, respectively, the instruction cache leakage energy and
performance behavior of different versions of vpenta. When we consider the performance behavior,
we see that the tiled version generates the best code. In fact, the tiled code outperforms all the
other versions in all leakage optimization strategies experimented. An interesting result here
is that the distributed (loop-fissioned) version outperforms the original code. This is because
placing arrays with conflicting references into separate loops reduces L2 conflict misses. When
we look at the energy results, we see that loop distributed version has the best energy behavior. This is because, as compared to the other optimized versions, at any given time the distributed version has fewer number of active cache lines. As a matter of fact, its energy behavior is so good that when one looks at the energy-delay product results, it was observed that in six of seven optimization strategies, it outperforms the tiled version.

Next, we focus on adi. As can be seen from the results given in Figure 5.18, only loop fusion is useful for adi.\textsuperscript{2} When we look at the (instruction cache) energy results (Figure 5.17), however, the picture changes. As compared to the original code, the fused code incurs much larger energy consumption (except for Strategy II). This is because at each iteration of the fused loops we need to activate more cache lines (i.e., cache lines are not held in the leakage control modes for a long enough duration of time). In contrast, the loop distributed version has a very

\textsuperscript{2}It should be mentioned that we are not trying to come up with the most appropriate use of these compiler optimizations. There might be several reasons why a compiler optimization may not perform as expected. For example, selection of tile size is very critical for the effectiveness of loop tiling [74]. A wrong tile size can lead to increased execution time. Similarly, unrolling factor is a very critical parameter in loop unrolling [16]. In this work, we have used these optimizations without trying to tune their parameters.

Fig. 5.16. Performance impact of optimizations on vpenta. Y-axis is expressed in million cycles.
Fig. 5.17. Instruction cache leakage energy impact of optimizations on adi. All values are normalized w.r.t. to the case without power management.

Fig. 5.18. Performance impact of optimizations on adi.
good instruction cache energy consumption. To see the combined impact of both energy and performance, we also evaluated the energy-delay products for this benchmark. We found that as far as the fused version is concerned, the energy losses cancel out the performance benefits in most cases, and the fused code and the original code exhibit very similar energy-delay behaviors. This tradeoff clearly emphasizes the importance of considering both energy and performance in deciding whether to apply a compiler optimization or not. We also observed that in four strategies the loop distributed version has the best energy-delay product (as a result of its good energy behavior).
Chapter 6

In-Cache Replication for Enhancing Data Cache Reliability

6.1 Introduction

Recent studies [61, 123, 69] have shown that transient hardware errors caused by external factors such as alpha particles and cosmic ray strikes are responsible for a large percentage of system downtime. Denser processing technologies, high clock speeds and low supply voltages can worsen this problem. Consequently, transient-error conscious system design is becoming a necessity for reliable functioning of hardware.

While transient errors can affect all hardware circuits, the cache is particularly more susceptible of the on-chip components. In today’s microprocessors, over 60% of the on-chip real estate is taken by caches, making them more vulnerable to cosmic ray strikes. Further, the high cache leakage power concern is leading to aggressive optimization techniques today that can increase errors. Errors in cache memories can lead to severe consequences on the execution, since they are very close to the main processing unit and can easily propagate to other components through the CPU and register file.

Currently the popular error checking schemes for caches use either byte-parity, which attaches one bit parity per eight-bit data, and SEC-DED (Single Error Correction, Double Error Detection). When we provision such schemes for transient error detection/recovery, there are three problems that arise. The first is the extra space that is needed for storing the redundant information. For instance, 12.5% extra overhead is incurred in storing an extra parity bit for each
eight bit data, and the same overhead is incurred for maintaining an 8 bit SEC-DED for a 64-bit entity. The second, and perhaps more important, problem is the latency for cache operations. Typically, we want caches to serve a request within a processor cycle so that the pipeline of the datapath can be effectively utilized. While simple schemes such as parity may be able to meet such demands, more complex ECC based schemes may find it difficult to sustain such speeds (unless caches themselves are deeply pipelined and there is a steady stream of requests to the cache so that the bandwidth issue is more important than the latency). While this may not pose a problem in low-end (embedded) processors that are clocked at slow frequencies, it is certainly not feasible to provide single cycle latencies for caches of high-end processors clocked over 1GHz. As a result, there is an extra cycle incurred for reliability checks — particularly on a load where this may be in the critical path — which can be an overkill if error rates are not very high. Even if one very pessimistically assumes an error every million cycles [69], this is a high price to pay for the normal functioning. Another reason why the latency issue is more important than the space issue is because there have been recent studies that have shown effective techniques for compressing the redundant information. For example, Kim and Somani [69] propose a very small cache to store parity information or to duplicate recently used data and demonstrate very good hit rates for this cache with such a scheme. The third problem is the additional power consumption for maintaining informational redundancy and verifying data integrity, and this power is much higher for SEC-DED mechanisms as compared to parity [7].

We propose a new mechanism for data caches (dL1) to enhance their reliability without compromising on performance. Note that error detection and correction is more critical for data caches (which can be written into), while detection may suffice for instruction caches which are mainly read-only. We refer to our mechanism as ICR (in-cache replication), wherein we use the
existing cache space to hold replicas of a cache block. This mechanism can be used either with ECC or with parity based schemes. When we use such a mechanism together with ECC, we can get better performance and/or reliability than with a scheme that uses ECC uniformly for all cache lines. Similarly, this mechanism can also be used with parity, to provide higher detection and recovery capabilities without degrading performance or power consumption significantly.

The basic idea for ICR is to perform replication without evicting blocks that may be needed in the near future so that there is no significant performance degradation (by cache misses). For this purpose, we use a dead block prediction strategy [63], wherein blocks that have not been used for a while are declared to be dead and their space in the L1 cache is recycled to maintain duplicates for the blocks in active use. Dead block prediction has been used in prior studies [63, 73, 45] for prefetching and cache leakage optimizations, and this is the first study to explore this idea for cache reliability. This dissertation uses this basic idea and shows several ways by which replication can be accomplished, both across the ways of a cache set, as well as across sets themselves. We investigate a large design space of when such replication should be done, how aggressive such replication should be, how many replicas to provide, how cache lookup should be done in the presence of replicas, and how we can check for errors in unreplicated blocks.

Transient errors are quite rare, and it is important that one does not pay a performance penalty or power cost in the common case when these errors do not occur. Even if the errors do occur, multi-bit errors are rare, in which case the scheme that we propose can catch single-bit errors using parity — without incurring performance penalties — and use the replicas within the cache to correct them. Further, it is conceivable that with replicas provided by our mechanism, one could possibly achieve even higher reliability than ECC in certain error situations.
6.2 Evicting Blocks Not in Use

When we are trying to replicate a block, we would like to put these replicas in the lines that contain data which are not likely to be used in the near future. This would help us alleviate any performance penalties, in terms of miss rates, from such replication. We refer to such lines as dead blocks.

Similar problems have been explored in the context of prefetching [73, 45], and cache leakage control [63]. The specific mechanism used in this work is similar to that proposed by Kaxiras et al [63], wherein a two-bit saturating counter is associated with each cache line. This counter is incremented at a timer tick and when it saturates, the block is declared to be dead. Whenever there is an access to this block in the meantime, the counter is reset. The state transition diagram showing this mechanism is depicted in Figure 6.1. Note that we need only 2 bits per cache block (this works out to 0.39% space overhead for a 64 byte line size) which is quite small (and its power consumption has also been shown to be quite low [63]).

![State transitions for cache decay.](image-url)
6.3 Replication Design Space

6.3.1 Important Questions

How aggressively should we mark blocks to be dead? The dead block prediction mechanism described in the previous section provides a straightforward approach to make space for replicas. A very high counter frequency is more reliability-biased, i.e., it is going to clear more space for replicas, and a low counter frequency is more performance-biased, i.e., it will try to hold on to blocks hoping for better temporal locality exploitation.

When do we replicate? In this work, we use two ways by which replicas can be created: (i) replicating data at the time it is brought from L2 (or memory) to the L1 if there is a suitable dead block at the replication site; and (ii) replicating the data at the time of a write to the block in L1 if possible, and updating both the original and the replicas.

Based on these two mechanisms, we consider two schemes. The first scheme uses both these mechanisms (replicating data at both L1 misses and at writes), while the second scheme uses only the second mechanism (replicating data at the time of a write). The reason that we include the second mechanism in both these schemes is because it is more important to provide higher reliability for blocks that are dirty (since in a write back dL1 these are not flushed to L2), while in the case an error occurs for a clean block, one could always go deeper in the memory hierarchy to bring the block [67].

The main difference between the two schemes is that the read-only data is not replicated in the second scheme, whereas it can be replicated in the first. Therefore, if an error occurs after the time data is brought from L2 to L1 but before it is written (updated) in L1, the first scheme
can provide replica from L1, while the second scheme needs to visit L2 to fetch the data. Since error occurrence is not a very frequent event, paying L2 access latency when these rare events occur can be acceptable. On the other hand, the second scheme may be able to replicate a higher percentage of modified data (as only modified data are replicated). Therefore, one can expect better reliability and lower power consumption with the second scheme. Note that, in both the schemes, if an error occurs for a modified data without replica (and without ECC), recovery may not be possible. However, such cases are expected to be less frequent in the second scheme where a higher percentage of modified blocks would be replicated.

Where do we replicate? We evaluate a class of schemes called “distance-k”, where parameter k is a non-negative integer, to search for dead blocks and to find replicas when the time comes. For example, assuming that original data is stored in set m, the distance-10 scheme places the replica in set (m+10) mod N, where N is the total number of sets in the cache. More specifically, the set (m+10) mod N is first checked if the replica can be placed there (discussed later). If it can, the replica is stored there (writing back the victim block if it is dirty). Otherwise, a fallback strategy (discussed later) is used. In our evaluation, we performed experiments with two values of parameter k: N/2 and 0, as depicted in Figure 6.2(a) and (b). In the rest of this dissertation, these two replication schemes are referred to as “vertical replication” (replication across sets) and “horizontal replication” (replication within the ways of a set), respectively. It should be noted that a bit needs to be associated with each block to indicate whether it is a replica or a primary copy.

How aggressively should we replicate? As discussed above, in cases where we find that set (m+10) mod N does not contain a suitable block to evict, we resort to a fallback strategy. Maybe
the simplest fallback strategy is “do nothing”, where we do not create a replica in dL1 in such cases. It is also possible to look for another place to put the replica. One such strategy is called “power-2”, and works as follows. We first try distance-k, where \( k = 2^i \). If this try is unsuccessful, we try distance-\( 2^{(i+1)} \) or distance-\( 2^{(i-1)} \), depending on which direction we want to proceed towards. If these are also unsuccessful, we next try distance-\( 2^{(i+2)} \) or distance-\( 2^{(i-2)} \), and so on. We can stop trying after a certain number of attempts have been made.

**How many replicas do we need?** For higher reliability, one may want to create more replicas of a given block. An important question is then where to create these replicas. One option would be to follow the power-2 scheme summarized above, and replicate the block at each set visited; that is, the difference here is that we have multiple copies of the same block.

**How do we protect unreplicated cache blocks?** We consider two options — just maintaining a parity bit at byte granularity, and maintaining an 8-bit SEC-DED at 64-bit granularity (called ECC) — in our experiments. Note that the consequence of the first option is that when an error occurs to a dirty block that is not replicated, we cannot recover back the data (if we do not have
a write-through cache). On the other hand, the second option can detect and correct such (single bit) errors.

**How do we protect replicated cache blocks?** As in the previous question, we could again use the same two options for protecting replicated blocks (i.e., both the primary and the replicas). However, since replicas automatically enhance reliability, and multi-bit errors are rare, we considered only the option of using parity to protect replicated blocks. Note that we could even have the option of dropping parity and simply use NMR techniques [108, 115] to detect and recover from errors based on the replicas. However, at least parity would be needed to protect non-replicated blocks, and this space should be allocated in any case for each cache line. Hence, we consider only the parity option.

**How do we place a primary copy in a set?** In the case of primary copies, we simply use the normal LRU mechanism to pick a victim regardless of whether it is a dead, replica or another primary block. In this regard, we are not any different from a normal cache placement.

**How do we place a replica in a set?** Here there are several options to find a victim, with the common characteristic being that they do not evict primary copies that are not dead (so that performance is not significantly affected). There are the following options:

- **dead-only**: We can perform LRU only amongst the dead blocks to find a victim. This approach tries to give reliability higher importance in not selecting candidates from replicas.

- **replica-only**: We can perform LRU only amongst the replicas to find a victim, giving performance more importance.
• *dead-first*: We can consider blocks that are dead or replicas for replacement, except that we check the dead blocks first.

• *replica-first*: We can consider blocks that are replicas or dead for replacement, except that we check the replicas first.

Of these, we do not feel that replica-only is very meaningful as it not only tries to eliminate the replicas, but it has also little scope for creating replicas when primary copies fill a set. Of the remaining, we mainly focus on dead-only and dead-first since our ICR mechanism is expected to leverage of the accuracy in dead-block predictions.

**What needs to be done upon a replacement?** When we replace a primary copy, we can either remove the replica or leave it there (for possible performance benefits). In the former case, an L1 load/store miss would directly go to L2. In the latter case, the L1 load/store miss could search for a replica either before going to L2, or could be done in parallel with L2 lookup. If we evict a replica, then we can simply throw it away.

### 6.3.2 Schemes Under Consideration

In this work, we consider the following schemes:

- **BaseP**: This is a normal dL1 cache without dead block prediction, which does not perform any replication. All lines are protected only by parity, and load/stores are modeled to take 1 cycle in our simulations.

- **BaseECC**: This is similar to BaseP, except that all lines are protected by ECC. Stores still take 1 cycle (as the writes can be buffered), but loads take 2 cycles to account for the
ECC verification. Note that in one set of later experiments, we consider speculative loads for BaseECC which complete in one cycle.

- **ICR-P-PS (LS)**: This implements our replication mechanism with dead block prediction that is performed at both dL1 misses and dL1 writes. Parity is used for protecting both replicated and non-replicated lines, and to detect errors (single bit errors at 8 bit granularity). On the occurrence of such errors, in the case of non-replicated blocks, the block is loaded from L2 if it is not dirty, and the error is unrecoverable if the block is dirty. If the parity indicates an error for a replicated block, the parity of the replica is checked. If this also has an error (though this probability is much smaller), we default to the actions taken for an error in the non-replicated case. When an error does not occur, the load only needs to access the primary copy and incurs 1 cycle (since it is only parity computation). If an error occurs, and the replica is needed, we need an extra cycle, making the load take 2 cycles. Writes are always 1 cycle since they are buffered.

- **ICR-P-PS (S)**: This is similar to the previous scheme except that duplication is performed only at writes.

- **ICR-P-PP (LS)**: The difference between this and the ICR-P-PS (LS) scheme is that the replicas are searched in parallel and are both compared before the load returns. We conservatively assume this to take 2 cycles for loads, and stores taking 1 cycle as usual.

- **ICR-P-PP (S)**: This is similar to the previous scheme, except that replications are performed only on writes.
ICR-ECC-PS (LS), ICR-ECC-PS (S), ICR-ECC-PP (LS), ICR-ECC-PP (S): These schemes are similar to the corresponding ones described above, except that ECC is used to protect the non-replicated lines, thus enhancing their reliability. One could ask why not use the ECC for the replicated lines as well, since the space would anyway be available for those lines in these schemes. However, our goal here is performance savings for the loads to such lines since parity comparisons would be much more efficient (and are modeled as 1 cycle versus the 2 cycles needed for ECC checks). This is the case for ICR-ECC-PS (LS) and ICR-ECC-PS (S) where the replica is not looked up until parity fails. On the other hand, ICR-ECC-PP (LS) and ICR-ECC-PP (S) always look up the replica and involve 2 cycles (and the benefits of these are more in their reliability rather than performance compared to BaseECC).

6.4 Experimental Settings

We have evaluated all the different schemes discussed in Section 6.3.2 together with the different parameters outlined in the design space. In order to conduct detailed cycle level simulations, we use the SimpleScalar 3.0 [13] infrastructure with its sim-outorder mode to model a multiple issue superscalar processor. The default simulation values that are used in our experiments are given in Table 6.1. The costs for stores (which are always 1 cycle) and loads to dL1 for each scheme have already been discussed in Section 6.3.2. We use eight applications from the Spec2000 suite for this evaluation.
<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
<td></td>
</tr>
<tr>
<td>Functional Units</td>
<td>4 integer ALUs, 1 integer multiplier/divider, 4 FP ALUs, 1 FP multiplier/divider</td>
</tr>
<tr>
<td>LSQ Size</td>
<td>8 Instructions</td>
</tr>
<tr>
<td>RUU Size</td>
<td>16 Instructions</td>
</tr>
<tr>
<td>Issue Width</td>
<td>4 instructions/cycle</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>1 ns</td>
</tr>
<tr>
<td><strong>Cache and Memory Hierarchy</strong></td>
<td></td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>16KB, 1-way, 32 byte blocks, 1 cycle latency</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>16KB, 4-way, 64 byte blocks, 1 cycle latency</td>
</tr>
<tr>
<td>L2</td>
<td>256K unified, 4-way, 64 byte blocks, 6 cycle latency</td>
</tr>
<tr>
<td>Memory</td>
<td>100 cycle latency</td>
</tr>
<tr>
<td><strong>Branch Logic</strong></td>
<td></td>
</tr>
<tr>
<td>Predictor</td>
<td>combined, bimodal 2KB table, two-level 1KB table, 8 bit history</td>
</tr>
<tr>
<td>BTB</td>
<td>512 entry, 4-way</td>
</tr>
<tr>
<td>Misprediction Penalty</td>
<td>3 cycles</td>
</tr>
</tbody>
</table>

Table 6.1. Configuration parameters and their values in our base configuration for a superscalar architecture. All caches are write-back.
6.4.1 Evaluation Metrics

We use different metrics in this study to compare the performance of the schemes and their effectiveness:

- **Execution Cycles** is the time taken for the execution of 500 million application instructions.

- **Replication Ability** is the fraction of times that one is able to replicate a cache line upon a load miss or a store (depending on the scheme used).

- **Loads with Replica** is the fraction of read hits that also find a replica in the cache at the time of the read. A higher fraction indicates higher reliability for the cache data.

- **Energy** is the total dynamic energy incurred because of accesses to dL1 and L2 caches.

6.5 Results

The results in Sections 6.5.1 and 6.5.2 use an aggressive dead block prediction strategy, wherein the block is immediately pronounced dead, as soon as the access for that block is complete making it a more promising candidate to hold replicas. We use the dead-only policy to choose the victim, which give higher bias towards reliability and the performance results that we obtain (in Section 6.5.2) will represent a pessimistic estimate of performance.

We then use the results of the impact of a larger decay window in Section 6.5.3 to evaluate the performance and reliability trade-offs in Sections 6.5.4 and 6.5.5, respectively. Section 6.5.6 explores the possibility of performance enhancements with choices upon replacement. Sections 6.5.7 summarizes the results from our sensitivity analysis. Finally, Sections 6.5.8 and 6.5.9 compare our schemes with other alternatives for addressing cache reliability and ECC performance problems.
6.5.1 Replication Mechanisms

Before we present the results for all the applications and configurations, we first investigate what is the effect of aggressiveness in replicating cache lines.

Figure 6.3 shows the replication ability for our benchmarks, with a single attempt strategy (Distance-N/2) and another which considers multiple attempts (Distance-N/2 and N/4 in that order). Note that in either case we are attempting to create only one replica. As can be seen, the multiple attempt strategy does allow a higher probability of replicating cache lines. However, when we look at the loads with replica (Figure 6.4), we find negligible improvement from multiple attempts at replicating. This is because the loads are mainly to the data lines which were replicated even with the single attempt. In general, in these results we find the overall replication ability relatively low since we are using the dead-only strategy for finding candidate positions, and after a point the number of such positions may become less with high replication rates. Despite this fact, the loads with replicas are much higher and this is what really matters as far as reliability is concerned.

The replication ability in creating multiple copies with the attempts being done on writes alone is shown in Figure 6.5. We show the ability to create just one replica (the alternate location(s) were not available for replication) as well as the ability to successfully create two replicas (i.e., three copies of a block exist in the cache). We see that we are able to create more than one copy around 12% of the time (averaged across the applications). Several copies can provide higher degrees of reliability with schemes such as NMR being possible. However, the space taken by these multiple copies can evict more useful blocks thereby worsening the locality and increasing miss rates as is shown in Figure 6.6. It is not clear whether the added reliability warrants this deterioration in miss rates (in some cases such as mesa, the miss rate nearly doubles).
Fig. 6.3. Replication ability for single and multiple attempt cases for ICR-P-PS (S). In either case, we are trying to create a single replica for the primary block. Multiple attempt case tries Distance-N/2 and Distance-N/4.

Fig. 6.4. Loads with replica for single and multiple attempt cases for ICR-P-PS (S). In either case, we are trying to create a single replica for the primary block. Multiple attempt case tries Distance-N/2 and Distance-N/4. Only the benchmarks that exhibit some difference are shown.
Fig. 6.5. Replication ability for single replica and two replicas for ICR-P-PS (S). For the single replica case, we use Distance-N/2. For the multiple replica case, the first replica tries Distance-N/2, and the second one tries Distance-N/4.

We next examine how the Distance-N/2 and Distance-0 replication mechanisms affect the behavior in Figure 6.7. We can see that there is little difference between these schemes for our default cache configuration. A reason for this behavior is that the sets in the cache are more or less evenly balanced in terms of the live/dead data lines that they hold. We have also conducted experiments with Distance-7 (a prime number) and the results were not any different from those obtained in the Distance-N/2 case.

All these results give us directions on setting the replication mechanism for the next set of experiments. We set the number of replicas to at most one and make only a single attempt to replicate. This is also favorable from the hardware viewpoint in simplifying cache lookup, and requires only 1 extra bit per cache line to indicate whether it is a primary copy or a replica. This can also be more power efficient than having more replicas since each lookup expends dynamic power. We have also fixed the alternate location choice as Distance-N/2 as our experiments with other distance values did not make any difference.
Fig. 6.6. Miss rates when single replica and multiple (two) replicas are created ICR-P-PS (S).
6.5.2 Comparing Performance — Aggressive Dead Block Prediction

Using the above settings for replication, we next compare the performance implications of the ten schemes described in Section 6.3.2. In these results, when the primary copy is evicted, all the replicas are evicted as well. Alternatives for these decisions will be explored later in the dissertation. Further, we are aggressively predicting a block to be dead as soon as its access completes, and the performance cycles are thus expected to be much worse than is possible with a more relaxed dead block prediction.

We begin by looking at the replication ability of our proposed ICR schemes based on the two opportunities when the replication can be attempted — upon a store alone (S), or at both stores and load misses (LS) — in Figure 6.8. As can be seen, LS allows higher replication abilities than S alone. The effect of the replication ability on subsequent load hits is shown in Figure 6.9. These results show very good promise for the ICR strategies since over 65% of read hits also find replicas in the cache across all applications. When we attempt to create replicas
Fig. 6.8. Replication ability for ICS-*LS) and ICS-*S) schemes. The results show that ICS-*LS) replicates more data than ICS-*S).

Fig. 6.9. Loads with replica for ICS-*LS) and ICS-*S) schemes.
at both opportunities (LS), we find over 90% of read hits finding replicas (almost resulting in complete duplication in mcf). The results in Figures 6.8 and 6.9 show that even if opportunities for replication may not be very high, the chances of finding a replica when needed may be extremely good. This is a consequence of the *locality* of the program, wherein a few data items (that are also getting replicated because of this reason) are in high demand, i.e., hot data items are getting automatically replicated (we do not need a separate cache for achieving this compared to that needed by [69]). These results suggest that this approach can be very useful in enhancing the integrity of data. We next move on to evaluating the performance consequences of such enhancements.

Fig. 6.10. Miss rates for Base*, ICR-*(LS), and ICR-*(S) schemes. Both ICR-*(LS) and ICR-*(S) increase the number of dL1 misses.

We give the miss rates for the normal cache (BaseP or BaseECC) and compare them to the deterioration in miss rates due to replication with the S and LS strategies in Figure 6.10. We can see that while in applications like mcf, there is not a significant difference in miss rates (in
Fig. 6.11. Normalized execution cycles for all schemes. Compared to the increases in the number of misses, the increase in execution cycles is not excessive in some of our schemes. This is because superscalar execution hides many cache misses. The average increases over BaseP due to ICR-P-PS(S) and ICR-ECC-PS(S) are 3.6% and 21.0%, respectively.

In this application, there is anyway very poor locality that evicting existing blocks to make way for replicas does not hurt performance), in some others there is a significant percentage change.

We next look at the performance impact of the replication strategies in Figure 6.11. All the bars are normalized for each application with respect to the bar for BaseP, which provides the best performance (no deterioration in miss rates, and both loads/stores take 1 cycle). BaseECC, on the other hand, does not deteriorate miss rates, but incurs 2 cycles for each load hit, leading to approximately 30% degradation in performance on the average. The ICR-*-PP schemes are comparable to ECC in general because it takes two cycles for each load hit to a block with a replica. This suggests that more than miss rate deterioration, the cost of the hit is much more important to optimize from the performance viewpoint. Even in ICR-P-PP, since most of the loads are to blocks with replicas (refer to Figure 6.9), the performance penalty of 2 cycle load latencies are more significant, making its behavior comparable to BaseECC or ICR-ECC-PP.
The BaseECC and all ICR-*-PP schemes are around 25-45% worse in terms of performance compared to BaseP.

The performance cycles of the ICR-*-PS schemes do not directly obey the deterioration of miss rates shown earlier. This is because of the savings in load latencies for accesses to replicated blocks (needing only parity calculations and not ECC) which become more important than the miss rates. Of these schemes, the ICR-*-PS (S) schemes fare better because of their lower probability of evicting useful data from the cache. Within this space, we find ICR-P-PS (S) doing slightly better than ICR-ECC-PS (S) as it is to be expected, since it does not incur 2 cycle latencies for loads to non-replicated data. But, this difference is less than 6% on the average across applications.

The two most attractive schemes from these results are ICR-P-PS (S) and ICR-ECC-PS (S). The former is only 3.6% worse than BaseP on the average across applications, while providing better reliability for replicated data which as pointed out constitute a large portion of the data in active use. The latter, while 21% worse than BaseP, is better than BaseECC by 15% performance-wise, while still protecting unduplicated data by ECC and providing higher reliability than BaseP for replicated data. It should be emphasized that we obtain such good results even with a very aggressive dead block prediction (with dead-only replacement) strategy. It is important to understand the performance and reliability ramifications of a more lenient dead block prediction strategy which is investigated in the rest of this dissertation. We also change from the dead-only to dead-first to give more options to find replica sites while not deteriorating miss rates. In the interest of space and clarity, we mainly focus on ICR-P-PS(S) and ICR-ECC-PS(S) in the rest of the dissertation, comparing them to BaseP and BaseECC.
6.5.3 Aggressiveness in Predicting Dead Blocks

We vary the number of cycles after an access is made to predict a block to be dead and the results are shown in terms of the replication ability and loads with replica in Figure 6.12 for vpr. The corresponding normalized increased execution cycles taken by ICR-P-PS (S) and ICR-ECC-PS (S) are given in Figure 6.13. We can see that while the replication ability reduces with an increasing decay window size as is to be expected, the corresponding effect on the loads with replicas is negligible. This is because the program locality is good enough that only a few replicas (of the hot data items) are needed, and these are anyway available in the cache. On the other hand, dead block prediction aggressiveness can affect the miss rate of the program and have an effect on the execution cycles as is shown in Figure 6.13. Considering these trade-offs, we use a 1000 cycle decay window for the rest of the experiments in this dissertation.

![Graph showing replication ability and loads with replica for ICR-P-PS (S) with different decay window sizes (vpr).](image-url)
Fig. 6.13. Normalized increased execution cycles with different decay window sizes (vpr). The increase in execution cycles due to ICR-P-PS (S) over BaseP is less than 4% for 1000 cycle window size and around 1.7% for 10000 cycle window size.

6.5.4 Comparing Performance — Relaxed Dead Block Prediction

The execution cycles of the schemes with a 1000 cycle decay window is given in Figure 6.14. In fact, with this set of parameters, ICR-P-PS (S) and ICR-ECC-PS (S) are only 2.4% and 10.1% away, respectively, from the best performing BaseP on the average, while providing much better reliability. Further, the performance of ICR-ECC-PS (S) turns out to be 16.8% better than that of BaseECC. Figure 6.15 shows that the loads with replicas for a 1000 cycle dead block decay window are not significantly different from that by setting the window to 0 cycles (even if there are some differences in replication ability), suggesting that the reliability behavior is not going to be significantly compromised. However, an investigation of the reliability behavior with error conditions is needed as is undertaken in the next section.
Fig. 6.14. Normalized execution cycles with a decay window size of 1000 cycles. The average increases in execution cycles (over BaseP) due to BaseECC, ICR-P-PS (S), and ICR-ECC-PS (S) are 30.9%, 2.4%, and 10.2%, respectively.

Fig. 6.15. Replication ability and loads with decay window sizes of 1000 cycles and 0 cycles.
6.5.5 Comparing the Schemes with Error Injection

We have considered several transient error models for the cache (direct, adjacent, column and random) described in [69]. Since the overall results are similar, we present the results specifically for the random injection model. In this model, an error is injected in a random bit of a random word present in the dL1 cache. Such errors are injected at each clock cycle based on a constant probability that is varied in these experiments.

Figure 6.16 shows the fraction of loads that could not recover from a single-bit error when the data is loaded as a function of the probability of an error occurring in each cycle for BaseP, ICR-P-PS (S) and ICR-ECC-PS (S) in vortex. BaseECC can detect and correct all such 1-bit errors. We find that even at such high (and unrealistic) error rates, the ICR schemes exhibit much better error resilient behavior compared to BaseP. Note that the intention here is to mainly show the benefits of ICR under intense error behavior, and hence very high error rates are depicted. In a more realistic situation, error rates would be much lower (in fact, for 1/100000, the error rates even for BaseP tend to become zero for all of our applications). This reiterates the importance of improving performance in the more normal case of error-free operation.

6.5.6 Performance Improvements

So far we have been focusing on enhancing reliability without degrading performance. However, our replication strategy can be adapted to enhance performance in certain cases. For instance, when there is a need for replacing a primary copy, it is possible to leave the replica in dL1, and simply replace the primary copy in isolation. This can help fill certain primary copy misses at a much lower cost than going to L2 (and can thus make the cache appear to have higher associativity sometimes [95]). In such cases, upon a primary copy miss in ICR-*-PS schemes,
there is only one extra cycle of load latency if the replica is present in dL1. This is much better than going to L2 (assumed to be 6 cycles in our experiments) that will be incurred in BaseP and BaseECC.

Figure 6.17 shows the performance results by leaving replicas in dL1 upon primary copy eviction for the ICR-*.S (S) schemes. We see that ICR-P-PS (S) and ICR-ECC-PS (S) are providing as good performance as BaseP (and much better than BaseECC) in nearly all cases, while providing much better reliability behavior as explained in the previous section. In fact, one could opt to even protect unreplicated blocks with ECC — which incurs higher load overheads — and still reap these benefits since most of the loads are to replicated blocks. We find that in mcf and vpr (and to a smaller extent in gcc, gzip and vortex), both these schemes outperform even BaseP (as much as 24% in one case), while providing better reliability characteristics. These results clearly demonstrate the benefits of in-cache replication for enhancing cache reliability without compromising on performance, and perhaps even enhancing performance in certain cases.
6.5.7 Sensitivity to Cache Parameters

We also conducted experiments with other cache sizes and associativities, and observed the overall results/trends to be similar. More specifically, the replication ability increases with increasing cache size since there are more replication sites available. However, the increase in the loads with replicas is not that significant. The other way of viewing this is that even in a small cache, we are replicating the data that is really the most in demand, and thereby providing higher reliability in the common case. This behavior is also observed when one keeps the cache size the same and varies the associativity. These results are not detailed here in the interest of space.

6.5.8 Comparison with Write-Through Data Cache

In addition (or in conjunction) to ECC, another way of enhancing the ability to recover data from errors, is to force data L1 writes to propagate to L2 as well. When a subsequent L1 read indicates an error (using parity), the value can be compared with that in L2 to possibly recover
from the error(s). Write-through L1 is thus another approach to enhance L1 data integrity as is the case in IBM POWER4 [9]. However, this has both performance and power ramifications. In order to reduce stalls on writes, a write-buffer is typically introduced between L1 and L2 [118]. Stalls are still encountered when the write-buffer fills up. Figure 6.18(a) illustrates the effect of these stalls on the overall execution cycles by giving the execution cycles of BaseP with a write-through data L1 (using a coalescing write-buffer of 8 entries), normalized with respect to our ICR-P-PS (S) scheme with a write-back L1. We find that our solution is around 5.7% better in terms of performance, on the average, across the applications.

A write-through data L1 also increases the dynamic energy consumption by increasing the number of writes to L2. Figure 6.18(b) compares the energy expended in the L1 and L2 caches for BaseP (with write-through), normalized with respect to ICR-P-PS(S) (using write-back). The per access dynamic energy figures have been obtained using the CACTI 3.0 tool [15]. We find that despite the higher L1 power incurred by ICR-P-PS(S) due to the duplicate writes performed when there are replicas, the overall energy expended in the data cache hierarchy (L1 plus L2) is still less than half of that for BaseP with write-through dL1.

6.5.9 Comparison with BaseECC with Speculative Loads

One way of circumventing the ECC time cost is to perform the checks in the background while the data is loaded and the computation proceeds speculatively. If an error is detected, the computation can then be squashed. It is to be noted that this does make the hardware more complex and may not be very suitable for embedded systems. Such speculative load capabilities are available today in certain high-end processors. Even if a processor supports speculative loads, while the ECC computation may be performed in the background to not get in the critical
Fig. 6.18. Performance and energy consumption (of L1 and L2 caches) with write-through dL1 for BaseP normalized with respect to ICR-P-PS(S) that uses a write-back dL1.
path, it does not affect the power consumption of the ECC mechanism itself. As noted earlier, ECC computations can be much more power consuming than simple parity calculations [7]. Figure 6.19(a) gives the execution cycles of BaseECC (with 1-cycle latency speculative loads), normalized with respect to the performance-optimized ICR-P-PS(S) version that leaves replicas in place. One can observe that, except mcf, ICR-P-PS(S) is still 2.5% better on the average than BaseECC with speculative loads. In mcf, our scheme is 30.8% better than the speculative BaseECC scheme.

Figures 6.19(b) and (c) show the energy consumption of the speculative BaseECC mechanism (with 1 cycle load latency) in the cache hierarchy (L1 plus L2), normalized to the energy consumption (L1 plus L2) of the ICR-P-PS(S) scheme for different values of energy consumption taken by parity and ECC computations. These values are given as a percentage of the cost taken by a normal L1 access. As can be seen from Figure 6.19(b), with a conservative assumption of ECC taking only twice the energy of a parity computation [7], we find ICR-P-PS(S) cache energy is more or less comparable to BaseECC (the difference is around 0.2% on the average across the applications) despite incurring overheads of performing two writes for replicated lines. On the other hand, if ECC computation turns out to be three times more costly than the parity computation from the energy perspective, the average cache energy increase of the speculative BaseECC mechanism over ICR-P-PS(S) is around 3.1% across these applications (see Figure 6.19(c)). It should be noted that though we have considered different energy costs for parity and ECC computations we are only giving representative results here in the interest of space.
Fig. 6.19. Performance and energy consumption (of L1 and L2 caches) for BaseECC assuming speculative (1-cycle) loads normalized with respect to ICR-P-PS(S), where replicas are left in place when primary copies are evicted. Representative results for Parity and ECC computations taking (b) 15% and 30% and (c) 10% and 30% respectively of the L1 access energy are given.
6.5.10 Performance, Energy, and Reliability Tradeoffs in Replicating Hot Cache Lines

While ICR scheme can improve the reliability of dL1 significantly in terms of our reads with replica metric, it may not be energy efficient to try to replicate all the cache lines in use. In this section, we study a modified strategy, which replicates only the “hot” lines, i.e., the decision as to whether a write needs to create a replica depends on the “hotness” of the data item in question. A data cache line can be classified to be hot if it is accessed more than the “hot-block threshold“ since it was brought into L1. We study the trade-offs between leakage saving and reliability by modulating the hot-block threshold.

To examine the three inter-related goals described earlier, we focus mainly on the following important statistics:

- **Execution Cycles.** This is the total number of cycles taken by the simulated instructions. Note that changes in miss rate due to line replication and in time due to bringing a line placed in low-leakage mode into normal operating mode have a consequence on the execution cycles.

- **Leakage Energy.** This is the leakage energy consumed in the cache during the entire execution. It is assumed that, when in the low leakage mode, a cache line consumes (conservatively) 10% of its original leakage energy.

- **Dynamic Energy.** This is the dynamic energy consumed in cache accesses (including writes to replicas and energy required to transition lines from low-leakage mode to normal mode, and vice versa).
• **Reads with Replica.** As discussed earlier, this is the fraction of read hits that also find a replica in the cache at the time of the read. A higher fraction indicates higher reliability for the cache data.

Figure 6.20 shows the behavior of our metrics for all benchmarks as a function of the hot-block threshold on the x-axis. All curves in the graph are normalized with respect to the corresponding values when no leakage control is done and no replicas are created (called *original* from now onwards). The rightmost values on the x-axes (default) correspond to the case where only leakage management is done (as per the approach proposed by [63]) and no replicas are created (consequently, there are no reads with replica). In general, we find that the reads with replica is the metric that is most affected by the hot-block threshold. As the hot-block threshold increases, there are fewer candidates for replication, and thus, the reads with replica decrease. This is most pronounced beyond a threshold of 100 cycles (e.g., see vortex as an example). While the execution cycles, leakage energy, and dynamic energy are also affected, the changes are less significant.

Execution cycles decrease slightly with higher thresholds since fewer replicas may be created in the places occupied by (possibly) useful data, thereby reducing miss rates. However, these results indicate the fact that dead-block prediction is doing a fairly good job and very few data items constitute the working set of the application, thereby not unduly increasing the misses beyond original misses. Regardless, we see across the entire spectrum of hot-block threshold values, the execution time increase does not cross 7.5%. Typically, this increase is due to the additional misses rather than the overhead of performing multiple writes to replicas since the latter can usually be done in the background without stalling the processor. This can also hide the overhead of waking up the replica from low-leakage mode when a write occurs. It should
also be emphasized that the performance overhead associated with replication is not significantly larger than that when using pure leakage management alone (shown as default in Figure 6.20) which has been accepted to be a good compromise between power and performance in numerous studies [35, 63, 66, 78, 147].

Recall that we are using the Normal Decay mechanism to transition replicas to low-leakage mode. As a result, creating replicas forces some of the lines (which are identified to be dead in the default case) to remain in the normal operating mode from the time of replica creation or write to the replica until the decay threshold is exceeded. Consequently, decreasing the hot-block threshold reduces leakage savings. However, even if we mark a line to be hot extremely aggressively (that is, when the hot-block threshold is 0), we are still able to keep lines in low-leakage mode for 75% of the time (which is only 13% worse than the default across all applications).

In the original execution, dynamic energy is expended only on loads, stores, and cache fills. The default execution, which puts lines in low-leakage state, expends additional dynamic energy to effect state changes to these lines. Consequently, we see an 10% increase in dynamic energy on the average across applications, even in the default execution which can be viewed as a compromise between leakage power savings and dynamic power increase. When we create replicas, some extra dynamic power is expended in creating replicas, in addition to causing more state transitions. The breakdown of the additional energy spent in these two components is given in Figure 6.21 for two benchmarks (bzip2 and vortex). We find that the energy spent in multiple writes due to replicas is not significantly different across a wide range of hot-block threshold values (when examining Figures 6.21 and 6.20 together). Overall, the dynamic energy expended with replicas is only around 8% more than the default (with respect to the case without any
Fig. 6.20. Baseline results. x-axis represents hot-block threshold (in cycles). The dead-block threshold is fixed at 1000 cycles.
power management) in the threshold range of 10-1000. We made similar observations with the remaining benchmarks.

Based on the results presented in Figure 6.20, one can observe that we can get very good leakage power savings (close to 84% on the average) with reasonable performance overhead (close to 4% on the average), and still find replicas 76% of the time when we use hot-block threshold values between 10 to 100. The dynamic energy increase in this operating range amounts to around 12% over the default. These results suggest that we can be fairly aggressive in marking cache lines hot (for replication) without having to worry about power and performance overheads.

One can visualize these observations by examining an integrated metric of our three goals, \( PER \) (Performance-Energy-Reliability product), wherein one can investigate the relative changes of the four different characteristics — cycles, loads with replica, leakage power, and dynamic power — to find a good operating point for the hot-block threshold. The \( PER \) metric can be defined as:

\[
PER = \left\{ \frac{1}{\text{Execution Cycles}} \right\} \times \{ \text{Reads with Replica} \} \\
\times \{ \alpha \times \frac{1}{\text{Leakage Energy}} + \{(1-\alpha) \times \frac{1}{\text{Dynamic Energy}} \} \},
\]

where \( \alpha \) captures the relative importance of leakage and dynamic power. We conducted experiments with different \( \alpha \) values, and in Figures 6.22(a) and (b) we present the results for \( \alpha = 0.08 \) (corresponding to roughly 180nm process technology) and \( \alpha = 0.50 \) (corresponding to roughly
70nm process technology), respectively. Note that a higher PER value indicates a better operating point. In the results given in Figures 6.22, leakage energy, dynamic energy, and execution cycles are all normalized with respect to the original case (where we do not have any leakage optimization or cache line replication).

We first observe that the trends are not very different across the two very different $\alpha$ values (while the absolute PER values are different). Second, we notice that the PER value is more influenced by the loads with replica metric than the others, primarily because this metric shows more variance compared to the others as mentioned earlier. Third, we can see that, while each application may have a different hot-block threshold giving a maximum PER value, the PER values for the hot-block thresholds in the 10-1000 range are not significantly lower from the maximum PER values for each application, suggesting that we can operate in this range.

Such a hot-block threshold range is also reiterated when we dig deeper into the reliability characteristics since this has the most influence on PER. For instance, Figure 6.23 shows the vulnerable reads (i.e., reads that do not have a replica and which have been modified) as two components — those that occur before the hot-block threshold was reached (before the replica is created), and those that occur after the hot-block threshold was reached (i.e., the replica could either not be created or was evicted after being created) for bzip2 and vortex. We find that beyond a threshold of 100, we find that more than 50% of the vulnerability is due to waiting for lines to become hot. That is, it may not be a good idea to work with large hot-block threshold values. At the lower end, we find that creating too many replicas starts thrashing these replicas in the cache, once again making the reads vulnerable.
Fig. 6.21. Breakdown of extra dynamic energy.
Fig. 6.22. PER (Performance-Energy-Reliability) metric. A higher value indicates a better operating point.
Fig. 6.23. Breakdown of reads with no replicas.
Chapter 7

Concluding Remarks and Directions for Future Research

7.1 Summary

A recent trend has been to consider energy consumption at all phases of hardware and software design. This dissertation has proposed three compiler-directed approaches to reduce energy consumption of datapath and instruction cache with minimal impact on performance. By using these approaches, one can tune energy consumption and performance based on the needs of the application at hand.

The dissertation is based on the observation that today’s high-performance processors are designed for general purpose and peak-performance oriented computation, which are not necessarily energy efficient. Therefore, one option is to exploit the idle execution cycles and idling hardware resources to reduce both dynamic energy and leakage energy as much as possible, with minimal impact on performance. Based on this idea, we have proposed three innovative approaches to exploit the idleness of functional units and instruction cache in VLIW machines. The first optimization exploits the scheduling slacks to reduce both the dynamic and leakage energy with no impact on performance or under a pre-determined performance constraint [151]. Since in general, a leakage control mechanism benefits more from longer slacks, our second optimization is built upon a data-flow analysis to identify idleness of a particular functional unit across basic blocks; and leakage energy can be reduced by exploiting the idleness detected [149]. Our third optimization is based on the generational behavior of the memory system [63]. We propose a
compiler-directed scheme to exploit this behavior and reduce leakage energy consumption of instruction cache significantly [147]. According to the experimental results, our compiler-directed approaches are very competitive, in terms of energy consumption or energy-delay product, compared to pure-hardware based approaches [63, 35]. This dissertation shows that an optimizing compiler can be very successful in finding the opportunities for energy optimizations by analyzing the program, transforming the program, and exploiting the available architecture/circuits hooks. As a result, energy consumption can be reduced significantly without compromising performance by combining the circuit level, architectural and compiler techniques.

Since the aggressive use of energy control mechanisms can make the hardware circuits more susceptible to soft errors, this dissertation has also proposed a novel solution to enhance the reliability for data caches without compromising on performance [148]. The solution proposed in this thesis replicates data that is in active use within the cache itself while evicting those that may not be needed in the near future. In addition, we study the trade-offs between leakage saving and reliability by modulating the hot-block threshold, which determines whether a cache block is active enough to be replicated or not. Our experiments show that a large fraction of the data read from the cache have replicas available with this optimization. The results also indicate that having a hot-block threshold in the range of 10-1000 cycles can provide good reliability characteristics, without compromising on performance or power.
7.2 Future Research Directions

7.2.1 Integrating Software and Hardware Techniques for Optimizing Cache Energy

While our compiler-directed approach for cache leakage optimization is very successful in reducing energy consumption compared to pure hardware-based approaches; there are some applications, for which the pure hardware-based solution outperforms the compiler-directed strategies. The reason is that the pure hardware-based approach can turn off cache lines periodically, which is independent of how long the loop nest in question executes. In contrast, the compiler-directed strategies cannot exploit opportunities inside the inner loop. Therefore, if the inner loop takes a long time to execute, the compiler-directed approach can miss opportunities to reduce leakage energy, compared to pure hardware-based approach. Our future work will focus on integrating compiler-directed and pure hardware-based approaches for reducing the leakage energy consumption further. Specifically, one can assume hardware counters to monitor accesses to cache lines; but these counters will not be activated unless the compiler reports that the hardware-based approach can be more beneficial. For those code fragment that the compiler can predict the execution times of the inner loops at compilation time, we can employ the compiler-directed approach if the inner loop does not take too long to execute. If the inner loop in question is predicted to take very long time to execute, however, the compiler can apply loop distribution [86] if possible to obtain multiple smaller inner loops to enable the compiler-directed approach. Otherwise, the compiler can report that the pure hardware-based approach is needed, and all the hardware counters would be activated and the hardware-based approach would be in charge. After the execution of the code fragment in question, all the hardware counters can be reset/deactivated.
7.2.2 Compiler-Directed Hot Data Replication for Reliability

This dissertation proposed a scheme to replicate hot cache lines only based on the access frequencies of the cache lines in question. While this scheme can provide good reliability characteristics without compromising much on performance or power, it is expensive to associate frequency information with each cache line and to update the frequency information at each access. Another disadvantage of this approach is that it does not distinguish the read and write operation, since the frequency is updated whenever there is a read or write operation on the cache block in question. Therefore, the hotness determined by this frequency cannot accurately find the cache blocks that will be read very frequently, which should be the main target to improve the reliability. Our future work will focus on investigating a less expensive technique to select the hot data to replicate. We will also study techniques that can distinguish between read and write operations.

An optimizing compiler can provide valuable information to determine the hotness of the data. Since we want to improve the reads with replica metric, the data should be replicated after its last write but before its read, and priority should be given to the data that will be read very frequently after the write. The compiler can analyze the definition (def) and use of variables. One can define the read-ratio of a variable to be the number of read (use) after the last definition (def) of the variable. Note that if a variable is defined before the loop but used within the loop, we can scale its read-ratio according to the loop count, if it can be determined at the compilation time or through profiling. In this way, we can try to replicate the data with read-ratio larger than a threshold (read-ratio-threshold), which can be specified by users. The read-ratio-threshold
controls the tradeoff between performance, power and reliability for our in-cache replication scheme.

7.2.3 Compiler-Directed Functional Units Leakage Reduction for Superscalar Architectures

While the compiler-directed leakage energy reduction for functional units proposed in this thesis is focused on VLIW architectures; we argue that the same idea can also be applied to reduce the leakage energy for superscalar architectures. The basic idea is that compiler can analyze usage of functional units to detect the idleness, predict the length of the idleness based on profiling information and then insert activate/deactivate instructions to reduce the leakage energy of functional units without compromising on performance.

However, the fundamental difference between the superscalar architecture and VLIW architecture is that superscalar architecture relies on the hardware scheduler to find the dependence and schedule instructions at runtime; while in VLIW architecture, compiler schedules all the instructions at compilation time. Also, since many superscalar processors can issue the instructions out-of-order; it may happen that the de-activate instructions are executed ahead so that the functional units are turned off even though they are needed to execute some instructions; which has both performance and energy penalties. Based on this observation, we re-define the semantics of the de-activate instruction. The de-activate instruction is designed to turn off a particular functional unit only at the commit stage; which means it can only turn off the functional unit after all the instructions before it have finished. Based on the new design of de-activate instruction, we will employ the same compiler-directed idleness analysis to detect the idleness of functional
units, to pass the information to the hardware and to study the leakage energy reductions for superscalar architectures.
References


[30]


[140] K.-L. Wu, W. K. Fuchs, and J. H. Patel. Error recovery in shared memory multiproces-
sors using private caches. IEEE Transactions on Parallel and Distributed Systems, Vol. 1,
Number 2, April 1990, pp. 231–240.

[141] W. Xu, A. Parikh, M. Kandemir, and M. J. Irwin, Fine-grain instruction scheduling for
low energy, In Proc. IEEE Workshop on Signal Processing Systems (SIPS’02), San Diego,
California, USA, October 16-18, 2002.


[143] W.Ye, Architecture Level Power Estimation And Experimentation, Ph.D Thesis, Depart-

[144] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simple-
power: A cycle-accurate energy estimation tool. In Proc. of Design Automation Confer-
ence, 2000.

[145] Y. Ye, S. Borkar, and V. De. A new technique for standby leakage reduction in high-

[146] W. Zhang, G. Chen, M. Kandemir, and M. Karakoy, Optimizations for Improving Data
Cache Performance of Array-Intensive Embedded Applications. In Proc. 40th Design Au-
tomation Conference (DAC-03), Anaheim, CA, June, 2003.


Wei Zhang obtained his B.S. degree in computer science from Peking University in July, 1997 at Beijing, China. He received his M.S degree in computer science from the Institute of Software, Chinese Academy of Sciences in July, 2000 at Beijing, China. In August, 2000, He came to Pennsylvania State University and enrolled in the Ph.D program in computer science and engineering. Since 2000 he has been employed in the Computer Science and Engineering Department as a teaching assistant or research assistant. In summer 2002, he went to Reservoir Labs Inc to do research in compiler optimization as a summer intern.

Wei Zhang is a member of ACM and IEEE.