Designing Energy-Aware Optimization Techniques through Program Behavior Analysis

A Thesis
Submitted For the Degree of
Doctor of Philosophy
in the Faculty of Engineering

by

Ananda Vardhan Kommaraju

Computer Science and Automation
Indian Institute of Science
BANGALORE – 560 012
August 2014
TO

my beloved parents
Acknowledgements

First and foremost, I would like to thank Prof. Y.N. Srikant for his enduring support and guidance all through the research. He has been more than a technical guide with his continuous encouragement and moral support during my time in the compiler lab.

I would also like to thank Prof. Y. Narahari, Prof. M. Narasimha Murthy, Prof. Priti Shankar, Prof. Mathew Jacob, Prof. Govindarajan and the entire Department of Computer Science and Automation for their help and support.

My parents and family members and many friends at IISc, CSA, and compiler lab had been incredible with their continuous encouragement during this period of my life.
Publications based on this Thesis


Abstract

Green computing techniques aim to reduce the power footprint of modern embedded devices with particular emphasis on processors, the power hot-spots of these devices.

In this thesis we propose compiler driven and profile driven optimizations that reduce power consumption in a modern embedded processor. We show that these optimizations reduce power consumption in functional units and memory subsystems with very low performance loss. We present three new techniques to reduce power consumption in processors, namely, transition aware scheduling, leakage reduction in data caches using criticality analysis, and dynamic power reduction in data caches using locality analysis of data regions.

A novel instruction scheduling technique to address leakage power consumption in functional units is proposed. This scheduling technique, transition aware scheduling, is motivated by idle periods that arise in the utilization of functional units during program execution. A continuously large idle period in a functional unit can be exploited to place the unit in low power state. This novel scheduling algorithm increases the duration of idle periods without hampering performance and drives power gating in these periods. A power model defined with idle cycles as a parameter shows that this technique saves up to 25% of leakage power with very low performance impact.

In modern embedded programs, data regions can be classified as critical and non-critical. Critical data regions significantly impact the performance. A new technique to identify such data regions through profiling is proposed. This technique along with a new criticality based cache policy is used to control the power state of the data cache. This scheme allocates non-critical data regions to low-power cache regions, thereby reducing leakage power consumption by up to 40% without compromising the performance.

This profiling technique is extended to identify data regions that have low locality. Some
data regions have high data reuse. A locality based cache policy based on cache parameters like size and associativity is proposed. This scheme reduces dynamic as well as static power consumption in the cache subsystem. This optimization reduces 25% of the total power consumption in the data caches without hampering the execution time.

In this thesis the problem of power consumption of a program is decoupled from the number of processor cores. The underlying architecture model is simplified to abstract away a variety of processor scenarios. This simplified model can be scaled up to be implemented in various multi-core architecture models like Chip Multi-Processors, Simultaneous Multi-Threaded processors, Chip Multi-Threaded to name a few.

The three techniques proposed in this thesis leverage underlying hardware features like low power functional units, drowsy caches and split data caches. These techniques reduce power consumption of a wide range of benchmarks with low performance loss.
Contents

Acknowledgements i
Publications based on this Thesis ii
Abstract iii

1 Introduction 1
   1.1 Focus of the thesis .............................................. 2
   1.2 Background and Motivation ...................................... 4
   1.3 Organization of the Thesis ..................................... 5

2 Low Power Techniques - Review of Literature 7
   2.1 Power and Energy Models ....................................... 8
   2.2 Review of Literature .......................................... 9
   2.3 Conclusion .................................................... 17

3 Leakage Power Reduction In Functional Units 18
   3.1 Introduction .................................................. 18
   3.2 Background and Motivation .................................... 21
      3.2.1 Examples .................................................. 22
      3.2.2 Dynamic Resource Usage .................................. 23
      3.2.3 The Energy Model ........................................ 25
   3.3 Scheduling Algorithm ........................................... 26
      3.3.1 Global Resource Usage Vector ............................... 27
      3.3.2 Closeness Heuristic ...................................... 28
      3.3.3 Working Example .......................................... 30
   3.4 Experimental Results ........................................... 31
      3.4.1 Power Consumption Estimation .............................. 33
      3.4.2 Multi-Issue Processor Energy Estimation .................... 36
   3.5 Related Work .................................................. 39
   3.6 Conclusion .................................................... 40

4 Leakage Power Reduction in Data Caches 42
   4.1 Introduction .................................................. 42
      4.1.1 Cache Leakage Problem .................................... 42
      4.1.2 Contribution ............................................... 44
   4.2 Background .................................................... 45
## CONTENTS

4.2.1 Attributing Criticality to a Data Region ........................................... 46  
4.3 Identifying Disjoint Data Regions ....................................................... 49  
  4.3.1 Context-Sensitive Profiling .......................................................... 49  
  4.3.2 Data Relationship Graph ............................................................. 52  
  4.3.3 Algorithm for the Creation of DRG ................................................. 54  
  4.3.4 Relevance of Context Length ....................................................... 56  
  4.3.5 Quantifying Criticality of DDR .................................................... 60  
4.4 Power Aware Optimization ................................................................. 61  
  4.4.1 Experimentation ............................................................................... 64  
  4.4.2 Results ............................................................................................. 65  
  4.4.3 Sensitivity analysis .......................................................................... 66  
4.5 Related Work ......................................................................................... 67  
4.6 Conclusion .............................................................................................. 69  

5 Locality Analysis of Disjoint Data Regions ................................................. 70  
  5.1 Introduction ........................................................................................... 70  
    5.1.1 Focus of the chapter ........................................................................ 71  
    5.1.2 Methodology .................................................................................. 72  
    5.1.3 Significance of Locality Analysis .................................................... 73  
    5.1.4 Organization .................................................................................. 74  
  5.2 Measurement of Reuse Distance ............................................................ 75  
    5.2.1 Defining Reuse Distance .................................................................. 75  
    5.2.2 Traditional Reuse Distance Measurement ....................................... 75  
    5.2.3 Measurement of Compound Reuse Distance .................................... 75  
    5.2.4 Working of the Algorithm .............................................................. 76  
  5.3 Cache Allocation Optimization for Power Savings .................................. 85  
    5.3.1 Experimental Setup ........................................................................ 89  
    5.3.2 Analysis ......................................................................................... 89  
    5.3.3 Cache partitioning ......................................................................... 91  
  5.4 Related Work ......................................................................................... 95  
  5.5 Conclusions ......................................................................................... 96  

6 Conclusions ............................................................................................... 97  
  6.1 Contributions ....................................................................................... 97  
  6.2 Future Directions ............................................................................... 99  

Bibliography ............................................................................................... 101
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Processor Parameters</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Split Data Cache Configuration</td>
<td>64</td>
</tr>
<tr>
<td>5.1</td>
<td>Split Data Cache Configuration</td>
<td>93</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Overview of the methodology used in the thesis. The simulations are carried out in SimpleScalar variations. We use ATOM based profile gathering framework. 4

3.1 Overview of the functional unit. This shows the transition from a high power mode to a low power mode. 19

3.2 Behavior of a functional unit. This shows the transition from a high power mode to a low power mode. 21

3.3 This is a valid schedule generated by a traditional scheduler. 22

3.4 Scheduling the instructions with a different heuristic. 22

3.5 A snapshot of continuous idle periods when optimization for transitions is not introduced. Note the short and small number of continuous idle periods (sparse). Y-axis - Continuous idle periods, X-axis - Transitions. 23

3.6 A snapshot of continuous idle periods when optimization is introduced. Note the long and large number of continuous idle periods (dense). Y-axis - Continuous idle periods, X-axis - Transitions. 24

3.7 The list scheduler is modified to incorporate the information of global resource usage and compute the closeness of an instruction with respect to the global usage. 27

3.8 A scheduling automaton. 29

3.9 An example DDG. 30

3.10 Scheduler Output using traditional heuristics and Closeness Heuristics. 31

3.11 State diagram to depict the functional of the counters to count number of idle cycles and number of transitions in each functional unit. 33

3.12 Percentage energy savings ALU. 34

3.13 Percentage energy savings in MUL unit. 35

3.14 Percentage energy savings FPU. 35

3.15 Percentage energy savings in Memory Port. 36

3.16 Percentage savings in the performance penalty over a circuit-level technique without our scheduling technique being performed. 37

3.17 Percentage energy savings in MultiIssue model with Multiple ALUs. 37

3.18 Percentage energy savings in memory port. 37

3.19 Percentage power savings in multiplication unit. 38

3.20 Percentage power savings in FPUs. 38

3.21 Multi-issue processor performance penalty. 39

4.1 Overview. 43

4.2 An example from the benchmark vpr. Boxed statements are more critical. Architectural model to control cache lines based on critical path. 45
4.3 Fields critical path model. This example shows a dynamic trace of instructions and dynamic dependency graph. ........................................... 48
4.4 The example considers malloc sites whose characteristics change depending on the context. This is a code snippet extracted from the parser benchmark. The path to the malloc site alters the way in which the created data is accessed. . . . 50
4.5 This example considers malloc sites whose characteristic would change depending on the context. The path to the malloc site changes the way the created data is accessed. ................................................................. 57
4.6 Three cases depicting different path lengths ................................. 58
4.7 Relevance on context length in Parser and Perl benchmarks. We can observe that there is an increase in the number of disjoint regions with context length. . 59
4.8 Criticality of vpr, eon and twolf using equations 4.1 are shown here. ........ 61
4.9 The total power savings compared against a baseline processor with no leakage control. The graph shows total savings in the data cache, both L1 and L2. The penalty due to this optimization is also shown . . . 62
4.10 This graph shows the sensitivity of the optimization. The comparison is between a complete L1 drowsy cache and split cache. The power consumption is compared against a baseline processor with no leakage control . . . 66
4.11 This graph shows the performance penalty between L1 drowsy cache and split cache. ................................................................. 67
5.1 Access Latencies and Power consumption (per-read power and stand by leakage power) ........................................... 71
5.2 Focus of the Chapter .................................................................. 72
5.3 A motivating example to show different regions of data behaving differently with respect to locality ........................................... 74
5.4 A motivating example to show how separate stacks for each data region can be used for reuse distance measurement ........................................... 78
5.5 Markers in different stacks and edges representing the ordering ................ 80
5.6 The pseudocode for computing reuse distance using partitioned stack .......... 81
5.7 An example explaining marker based algorithm to compute Global reuse distance .... 82
5.8 An example explaining marker based algorithm to compute Compound reuse distance using partitioned stack ................................. 84
5.9 Data Region reuse distance histograms and memory savings .................. 85
5.10 Distribution of accesses to each disjoint data region. Y-axis indicates percentage of access to a particular data region. ........................................... 90
5.11 Distribution of misses of each disjoint data region on a 32KB,4 way cache. Y-axis indicates percentage of misses in that particular data region. ........ 90
5.12 Percentage of heap accesses ...................................................... 91
5.13 Percentage of cache misses of heap access ................................... 91
5.14 Distribution of footprint or total size of each disjoint data region .......... 92
5.15 Percentage of heap data footprint. Y-axis indicates percentage of heap data available in the benchmark when compared to total memory footprint. .... 92
5.16 The access latency and power consumption of different smaller caches and the baseline .................................................................. 93
5.17 Different configuration and per-access power consumption .................. 94
5.18 Total Power Savings in both L1 and L2 data cache .......................... 94
## List of Algorithms

1. Clustering Algorithm .......................................................... 52
2. Data Relationship Graph Creation ........................................ 53
3. Creating Connected Components ......................................... 56
4. Traditional Reuse Distance Measurement ............................. 76
5. Modified Reuse Distance Measurement ................................. 77
6. Apriori-based subset selection ............................................. 86
Chapter 1

Introduction

The explosive growth and rapidly advancing pace of technology in embedded devices place a great emphasis on efficient usage of energy and power savings in embedded processors. This issue has attracted significant attention over the last decade. Several studies discussed the trade-off between performance and power consumption. This gave rise to several solutions which were hardware specific, software specific, and hybrid solutions which combined both hardware and software techniques. Among hardware specific techniques, Intel introduced a Tri-Gate 3D transistor which is expected to accelerate mobile device performance to remarkable levels while slashing the CPU power consumption by half [1]. The software based approaches exploit hardware techniques like Dynamic Voltage and Frequency Scaling [2],[3],[4],[5],[6],[7],[8] to schedule and operate different workloads. Emerging multicore architectural models like Chip Multi-core Processor (CMP), Simultaneous Multi-Threaded (SMT) architectures, Multi-Processor System on Chip (MpSOC) etc. have tried to mitigate the power consumption problem by employing small and low power cores.

Despite these solutions, power consumption still remains a primary concern. These solutions have their own disadvantages that restrict their use and effectiveness. Hardware techniques add elaborate complex structures that lead to complicated designs resulting in sub-optimal solutions. Conventional methods like transistor resizing, low voltage design techniques, and frequency control methods can be applied only during early design stages. Hence there is an ever increasing need to reconfigure these solutions to support complex applications like tablets, smart phones, and other embedded devices. On the other hand, software solutions use conservative program analysis techniques on specialized applications. In this thesis we follow
a hybrid approach to mitigate such limitations. The next section describes the focus of this thesis.

1.1 Focus of the thesis

In this dissertation, we explore the possibilities of compiler directed techniques to utilize hardware optimizations more effectively. We experimentally show that such transparency of architectural and micro-architectural details can be exploited by the compiler to design new optimizations. These techniques complement the existing solutions.

To start with, we identify common patterns of behavior in programs. These patterns are nonuniform usage of functional units, variation of criticality, and locality across data regions. We address the problem of power consumption at the compiler level by designing optimizations around these common patterns of behavior. Such effective optimizations can motivate application designers and language designers to even push these optimization opportunities to a higher abstraction level for the use of programmers. In addition, compiler based approaches can be automated, thus help in harnessing the low power techniques supported by the hardware.

Each chapter of the thesis focuses on power saving techniques, placing a greater focus on leakage power in functional units and data caches. We perform a thorough analysis of certain program behavioral properties like resource usage, locality requirements, and latency tolerance. The thesis aims to provide an overall understanding of the techniques at the circuit-level and micro-architectural level that could be utilized to achieve optimal power savings without hampering the performance.

Below, we briefly explain the contributions of the thesis with respect to different components of the processor.

- **Leakage power reduction in functional units**

  The leakage power in the functional units, which are part of hot-spots of the processor, increases significantly as we move to nanometer transistor sizes. We attempt to reduce the leakage power consumption through a compiler-driven technique called *Transition aware scheduling*. The technique models the resource usage through a finite state model which is characterized by the number of idle cycles, active cycles, and transitions. We introduce an instruction scheduling algorithm which reduces the number of transitions between
Chapter 1. Introduction

power modes, thereby increasing idle time in resource units. These idle periods can be exploited to achieve significant power savings by placing the unit in low power mode or turning it off completely. As we increase the number of continuous idle cycles using our instruction scheduling algorithm, we achieve an average power reduction ranging from 4%-15% in resource units without any significant performance penalty.

• **Leakage power reduction in data caches using data criticality**

Memory consumes significant power as compared to other processor components like functional units. An important property that plays a critical role in shaping program memory behavior is latency tolerance. We analyze the latency tolerance of the data structures to quantify the criticality of a disjoint data region using a critical path model. Subsequently, we devise a cache partitioning scheme, where one of the partitions is a drowsy cache that is used to allocate latency-tolerant data structures (non-critical data regions). This leads to an overall power reduction of 28% to 38% and a leakage power reduction ranging from 10% to 20%.

• **Leakage power reduction in data caches using locality analysis**

Finally, we analyze the locality of data structures. We use reuse distance as a guiding metric to characterize the locality of the data structures. We present a novel distributed stack based algorithm to measure the reuse distance. In this algorithm we map each data structure to a stack. This algorithm is used in an apriori-based technique to select subsets of data structures that require smaller or larger cache sizes. Using this algorithm, we measure the reuse distance of subsets of data regions. Such subsets of data regions can be allocated into a partitioned cache with different associativities on the basis of their reuse distance. This results in an average power reduction of 23% in the data cache without significant performance loss.

Figure 1.1 represents an overview of the techniques discussed above. To begin with, we use MediaBench and MiBench benchmarks, to conduct a experimental study and measure the idle periods in different functional units in a simulator. The CPU model is simulated using SimpleScalar and its variants like WATTCH, XTrem and HotLeakage. The power models derived from the simulator help us define algorithms for power optimization. Based on these algorithms, we employ a compiler directed technique, transition aware scheduling, that generates
Figure 1.1: Overview of the methodology used in the thesis. The simulations are carried out in Simplescalar variations. We use ATOM based profile gathering framework.

a power optimized code. These new programs reduce leakage power in functional units.

Next, we consider benchmarks like SPEC2K and Olden. We use program analysis tools like PIN and ATOM for instrumenting these benchmarks and gathering profile information. The profile information thus obtained constitutes the input for Datastructure Analysis, Criticality Analysis, and Locality Analysis. The output from the analyses are used to generate a modified and optimized programs that can be executed on the simulator for power optimization. These programs have memory operations hinted with additional information criticality and locality using a single bit in the instruction.

1.2 Background and Motivation

There are two types of power consumption in a processor: leakage power or static power, which indicates the steady state consumption in different units, and dynamic power, which indicates the active power on account of variations in the types of operations that the circuit performs and the switching of transistors.

In earlier decades, static power dissipation constituted only a small percentage of the total power consumption while the dynamic power consumption assumed a dominant position. With the rapid shrinkage of embedded devices scaling of transistor sizes towards sub-micron and nanometer regimes, leakage power consumption has assumed prime importance [9]. Existing leakage reduction techniques involve turning off the processor completely, which involves high
energy overheads. The overall power consumption studies show that functional units and memory subsystems consume significant leakage and dynamic power. Functional units require around 10% of total die area of a processor. They are the hot spots in the processor as a subset of these units are accessed every cycle during program execution. Data caches (apart from instruction caches and SRAM), also consume significant leakage power because they utilize 30% to 40% of total die area. As the modern mobile devices run highly data intensive applications, the data caches consume significant dynamic power as well.

In this thesis we have isolated the power consumption issue due to the program behaviour independent of the number of cores. The execution model of a single application on multiple cores is similar to running the same application on a single core with a smaller size cache hierarchy. In this thesis we have investigated the primary reasons for power consumption in functional units, caches when running a single task with private cache hierarchy. We have simplified our underlying architecture model to abstract away a variety of multi-core scenarios like CMP, SMT and MpSOC [10]. This simplified model can be scaled up to emulate various multi-core architecture models including task based multi-threaded in-order execution cores or out-of-order execution cores.

Recent studies [11] show that smaller, slower multi-core processors have replaced faster, complex single-core processors in different modes of computing systems. But the study observed that there is significant dark silicon present in multi-core processors. Dark silicon indicates regions of cores that are turned off due to power constraints. The authors indicate newer techniques in micro-architecture and other techniques lead us into more energy efficient architectures. This thesis primarily aims at designing software-level optimization that can complement wide variety of architectures leading us into the future. In this thesis we propose compiler-driven and profile driven optimizations that reduce power consumption in a modern embedded processor by studying program properties such as critical data regions and usage of functional units.

1.3 Organization of the Thesis

The rest of the thesis is systematically organized as follows. Chapter 2 presents an overview of related work, Chapter 3 discusses transition aware scheduling, a compiler technique to reduce
leakage energy consumption in functional units. Chapter 4 explains the critical path model and its application to identify critical data structures to drive power aware optimization. Chapter 5 introduces the main data structure analysis and effective reuse distance. This analysis is further employed to design power aware optimization. Chapter 6 concludes this thesis with ideas on how to extend the proposed techniques.
Chapter 2

Low Power Techniques - Review of Literature

The distribution of power consumption by different units in embedded systems depends on the workload characteristics and architectural variations. Earlier, the focus was predominantly concentrated on the processor core, like branch predictors, issue logic, register file etc. With the accelerating pace of technology advancement, hot-spots like functional units became prominent targets for reducing power consumption as they directly influence the processor temperature and overall cooling costs. As the workloads became more and more memory intensive, data caches, DRAM memory, and hard disks assumed primary importance.

Prior studies introduced many proposals for reducing power consumption. These proposals can be successfully implemented through compiler optimizations which can be extended to an existing compiler framework. Power consumption in a processor unit is classified into leakage power and dynamic power. The modern day microprocessors and other semiconductor designs are based on CMOS technology. In order to design energy saving optimization in compilers, one has to understand the models explaining power and energy consumption in CMOS chips. The present chapter begins with an elaborate explanation of these models, followed by a manifestation of various proposals which present novel optimizations designed and implemented in compilers for low power consumptions.
2.1 Power and Energy Models

Energy is the product of power and time. One may deduce that optimizing for power might in turn reduce time or vice versa. However in a practical sense, this is not true. For example, power consumed by a computer system can be reduced by halving the clock frequency. But if the computer takes twice as long to complete the same task, there would be no significant reduction in the total power consumed.

In mobile systems, where there is an ever increasing need for longer life batteries, it is imperative to achieve energy optimizations. On the other hand, servers demand instantaneous power consumption and controlled temperatures with lower emphasis on total energy consumption.

How does power dissipate in a CMOS device? Consider the following equations presented below:

\[ f \propto \frac{(V - V_{th})^\alpha}{V} \]  \hspace{1cm} (2.1)

where, \( V \) is the supply voltage of the device, \( V_{th} \) is its threshold or switching voltage of a MOS transistor, \( f \) is its operating frequency, and \( \alpha \) is a technology dependent constant, which has been experimentally determined to be approximately 1.3. It must be noted that as the voltage is reduced to the level of \( V_{th} \), \( f \) decreases to zero. This equation can be approximated by a linear relationship as in equation 2.2 below.

\[ V_n = \beta + (1 - \beta) * f_n \]  \hspace{1cm} (2.2)

where, \( V_n = V/V_{max} \), and \( f_n = f/f_{max} \), and \( \beta = 0.3 \), for current MOS technology. Substituting for \( \beta \) in Equation 2.2, assuming \( V >> V_{th}, V_{max} >> V_{th} \), and simplifying, Equation 2.2 becomes

\[ f = kV \]  \hspace{1cm} (2.3)

where, \( k = f_{max}/V_{max} \), is a constant within the operating range of voltage and frequency.

Power dissipation in a device is given by

\[ P_{CMOS_{device}} = \frac{1}{2}CVV_{swing}af + I_{leakage}V + I_{short}V \]  \hspace{1cm} (2.4)

where, \( P_{CMOS_{device}} \) is the power dissipation of the device, \( C \) is the output capacitance, \( a \) is the
activity factor \((0 < a < 1)\), \(V_{\text{swing}}\) is the voltage swing across \(C\), \(I_{\text{leakage}}\) is the leakage current in the device, and \(I_{\text{short}}\) is the average short circuit current when the device switches, short-circuiting the drain and the source. The three terms in Equation 2.4 correspond to switching power loss (dynamic power loss), leakage power loss and short-circuit power loss, respectively. As technology scales and transistor size shrinks, the leakage current becomes a dominant factor. As per the current trends, leakage and dynamic power in a processor are approximately similar, advocating the need to develop new techniques to reduce leakage power consumption.

Leakage power and energy loss can be controlled by voltage gating of function units, memory, and cache[18]. Short circuit power loss can be reduced by superior VLSI technology.

2.2 Review of Literature

The subsequent discussion presents a brief overview of various low power techniques.

Amongst the various hardware and software techniques devised in the past for low power consumption, the compilation approach is more advantageous because it can be automated, thereby reducing the hardware costs both in terms of power and area. The background and advanced techniques on various compiler based techniques which form the basis of most of the compiler based low power as well as optimization techniques are discussed in [25] [26]. Also, its capabilities can be augmented further by exploiting the large window of program execution. Such compiler techniques target different components of CPU to reduce total energy consumption. The distribution ratios of power consumption may vary across different processors, but the dominant energy consumers remain the same, the most important being caches and functional units. Other components like branch predictors, register files, Translation Lookaside Buffers also tend to consume high energy. The subsequent section outlines a brief discussion on various proposals which target reduction of power consumption in different components through compilation techniques.

Modifying Memory Refresh Time

Of all the components of a modern computing system that consume energy, DRAM is a significant contributor. This is because of the extended idle time periods of such computing systems, (especially mobile phones), long access latency and high access power consumption
per access. Increased idle time requires the system to be refreshed periodically to protect data, which ultimately leads to leakage of charge and wastage of power (leading to a consumption of approximately 5-30% of the total power [17]). This is unacceptable for users of laptops, tablets, smart phones and other handheld devices since it is a common practice to maintain the system in the standby state for multiple days without recharging. In an attempt to reduce extensive power consumption by DRAM, Intel has been investigating the possibility of extending refresh time and reducing the physical distance between data transfers by allowing the DRAM dies to stack on one another so as to reduce standby power. It has been noted that such leakage of power from DRAM cells follows an exponential distribution with a miniature proportion of memory cells having higher leakage rates than the others [15]. Leveraging on this fact, [27] proposes a software technique to reduce refresh power and thus obtain significant reduction of power in DRAM memories. The technique uses a differential allocation strategy to differentiate between critical and non critical data in applications and reduces the refresh rate for non critical data regions while ignoring the possibility of errors in these pages. Since such errors have limited exposure to applications, it does not lead to significant degradation of the system in terms of reliability. This led to an average power reduction ranging from 20 to 25% in the DRAM with a mere 1% depreciation in the reliability and performance of mobile systems [12].

**Phase Change RAM**

The possibilities of reduction in power consumption in hybrid main memory/Phase change RAM (PRAM), which is inclusive of DRAM and non volatile memory are explored in [28]. This proposition rests on the fact that non volatile memory is more advantageous in the context of low power consumption during standby while DRAM gives better performance and active power [29]. Through this approach, DRAM refresh time is avoided which reduced the energy consumption of the main memory by 23.5% - 94.7% and a performance overhead of 0.004% -0.206% which excels the conventional DRAM [28]. Conventionally the compiler influences the DRAM to a lower extent as compared to the operating system. However, there is a tremendous scope for better approaches through data prefetching, locality analysis and data layout, which enable the compiler to make efficient use of DRAM.
Chapter 2. Low Power Techniques - Review of Literature

Banked Memory

Kandemir and Bhadauria concluded that the compiler can analyze data accesses which alter the operating modes of DRAM [30] [31]. In these studies, the authors analyze the array accesses in loop bodies. This analysis identifies the regions of arrays that are accessed in limited iterations. They assume that the array is being allocated in a banked memory. They further attempt to program the hints so that certain banks can be either activated or deactivated.

Extensive work on reducing power consumption by employing scratch-pad memory was carried out by various proposals [32] [33] [20][21]. These proposals present evidences where in hot data regions (static as well as heap based) are allocated to the scratch-pad memory which in turn reduces the per access power consumption. Also, intelligent data replication techniques in memory banks can direct accesses to a smaller set of banks. Such techniques increase the idle period in the data banks which can be placed in low power modes [34].

Dynamic Voltage Frequency Scaling

Over the years, dynamic voltage and frequency scaling (DVFS) has been an effective technique for controlling power and voltage in modern microprocessors. In the last decade, several prominent studies have devoted considerable attention towards intra task DVFS [2][3][4][5][6][7][8]. In the context of a dynamic compilation or optimization environment, [35] extends and fine tunes this technology so that voltage and frequency can be reduced at the time of program execution to take advantage of application phase changes. This technique yielded better performance as compared to conventional static voltage scaling and led to an average energy savings of 22% which is commendable.

ILP Based Dynamic Voltage Scaling

The preliminary work on using compiler algorithm to reduce power consumption was proposed by Ulrich Kremer et.al. [8]. Their work focuses on the design and implementation of compiler algorithm using dynamic voltage scaling (DVS) with not more than a slack time of 5% of the total execution time. Dynamic voltage scaling is an effective power reduction technique and it lowers the supply voltage significantly to reduce power dissipation. This is based on the fact that a major portion of power of CMOS circuitry scales quadratically with supply voltage [36]. These algorithms based on DVS are effectively implemented to reduce the energy consumption
and energy delay product (EDP) by 11% and 9% respectively. Essentially, these algorithms divided a program into two partitions. During the execution of a selected partition, the supply voltage was reduced thereby reducing the total energy consumption of the program. Selection of an appropriate program region is done using ILP equations which are formulated by taking into account the frequencies of different program regions, size of the program regions, and the number of transitions between subsequent regions.

Register Caching

As the register file occupies a small area on the critical path of the execution, its power density is very high. Moreover, it exhibits a higher temperature as compared to other components. Each of the several threads in a modern GPU stores its register context in the processors register file, leading to large register files that consume almost 20-30% of the total processor dynamic energy [16],[17]. As an example, consider NVIDIA GPU which contains 128 KB register file per processor, with 16 processors per chip [9]. Hence optimizing energy consumption on the register files leads to a significant reduction in the total energy consumed by the processor and hence remains a potential area of research. Using a combination of hardware and software solutions, the register file can be redesigned to reduce the energy consumption without impacting the performance of the processor [16]. Using a compiler controlled register file hierarchy, the energy consumption on the register file can be brought down by 54% which is far more commendable as compared to 34% using hardware only caching approach. In [37], the authors propose a new mechanism to reduce the power density in the register file. They observe that there is a high spatial locality in the register accesses, which indicates that closely placed registers are accessed more frequently. They introduce a new register allocation algorithm which disperses the accesses across the register file so that clustered accesses are minimized.

Hot and Cold Regions

Paritioning a register file into hot and cold regions to exploit skewed usage of variables was proposed in [38]. The hot region is maintained at a high power state where as the cold region is maintained at a lower power state. The more frequently accessed variables are allocated to the hot region of the register file. The other variables are allotted to the cold region. Due to the different voltage levels of these regions the authors strike an optimal balance between power
consumption and performance of the program. Furthermore, the technique in [39] divides the register file into equal partitions. At the time of execution a particular partition is activated while others are maintained in a low power mode. This reduces the total leakage as well as dynamic power.

**Power Gating of functional units**

Functional units are accessed in the critical path of the execution. They are the primary hot-spots for both power density as well as temperature. With exponential advances in technology scaling, leakage power and dynamic power become an important concern in low power CMOS circuits. Power gating is a conventional method of reducing leakage power using sleep transistors by restricting power supply to processor parts which are not functional for extended periods during program execution. This can be achieved by micro architectural techniques as well as compiler techniques. However it takes considerable time for the circuit to become electrically stable once the power supply has been restored (also called activation latency) leading to elevated dynamic power overheads [18]. Hence activation latency and the minimum time required for the circuit to be switched off are important factors which need considerable attention while implementing power gating techniques.

Park et al. [19] extend a power gating functionality using a micro architectural technique that can be applied to embedded microprocessors with multiple execution units like integer multiplier and floating point arithmetic computation unit. This technique leads to a reduction in power dissipation in embedded microprocessors by shutting them down during idle times. By implementing a polynomial time optimal algorithm to insert ON/OFF instructions for the activation/deactivation of power gated execution units, the study achieves significant reduction in the leakage power [19]. ON instruction triggers the active state of the execution units while OFF instruction induces the execution unit into sleep state to reduce leakage. This is done by adjusting the registers to 0 and 1 respectively. The study is based on the premise that if the execution unit which is already in the active state encounters an ON instruction or vice versa, the execution of the instruction is skipped.

High sensitivity of leakage on temperature and process variation result in very high variation of functional unit power across processor dies. Such process variation can reduce the performance of a processor. In [40] the authors describe a technique which leverages on-chip
sensors to reduce leakage in the functional units.

### Sleep Instructions

Roy et al. [18] extend a more detailed investigation into these issues and formulate a compiler level framework with architectural support for leakage reduction in embedded microprocessors. The proposed framework uses break even period and activation latency to identify power gating opportunities within coded segments and insert sleep instructions into the code. This is done by using static code analysis and dynamic profiling information. This activates the required functional units, every time an instruction is decoded, just before reaching the execution stage. This approach leads to a remarkable reduction in the instruction overheads using power gating techniques [18]. The above study also opens avenues for further exploration in designing functional units by using more advanced techniques like dual sleep transistors and charge recycling devices.

### Data Flow Analysis

In [41], the authors present a data flow analysis which identifies functional units that are accessed in a particular program region, e.g. a loop body, function body or a single basic block. The compiler introduces hints at the entry points of these program regions. During the execution of the program, these hints instruct the processor to turn off the functional units which are not accessed in that region. [42] illustrates a profile driven analysis which estimates the usage of functional units and instructs the processor to turn off the functional units dynamically. In a nutshell, long idle times in functional units present excellent opportunities for power gating to save leakage power. In this thesis we mitigate the problem of the transition overheads by increasing the continuous idle periods in the functional units by modifying instruction scheduling techniques. Using this approach, we achieved a reduction in the power consumption by 25%. A more elaborate discussion of this approach is presented in chapter 3.

### Compiler Hints

Instruction caches are accessed in every cycle. While minimizing power consumption has been a critical design parameter for embedded systems, there has been a steady growth in cache sizes to reduce the gap between main memory latency and core clock frequency [43]. Caches
occupy a significant proportion of die area and hence consume considerable leakage power and dynamic power. Thus, increase in the size of instruction caches has aggravated the problem of power dissipation. For example, StrongArm dissipates 27% of its processor power in the instruction cache [44]. One of the main reasons accounting for increase in cache sizes is an exponential growth in leakage power due to shrinking of transistor sizes. One of the ways to reduce leakage power consumption in the instruction caches is to deactivate the cache lines with an appropriate policy to determine when to turn these lines off and on. Some of the architectural based approaches discussed in the earlier literature to save leakage power in data caches include dynamic supply voltage scaling [45], turning of L2 cache lines when the same information is duplicated in L1 cache [46], dynamic resizing of cache based on performance [47][48], supply gating of cache lines during idle periods on the basis of their associated counters [49].

To quote a few more evidences, a compiler based technique in [50] embeds hints in the program to control the state of the cache lines in the instruction caches. [51] portrays banked instruction caches which govern the allocation of caches based on the program region. The per-access power consumption directly depends on the size of the instruction cache. Therefore, by dividing the cache into banks, this technique, reduces per-access power consumption.

Filter Caches

The proposals [52], [53] introduce a filter cache scheme which also increases cache access latency leading to a slight degradation in the performance. This assumed primary importance in subsequent studies which focussed on mitigating the problem of performance degradation of filter cache. For example, Bellas [54] proposed a loop cache which uses compiler support to filter out the frequently accessed instructions from the level one cache. [55] proposes the concept of Trace Reuse cache which is placed at the same level of memory hierarchy as the conventional instruction cache which eliminates the problem of additional cache latency. The Trace Reuse Cache is comprised of a History Trace Buffer (HTB) and Trace entry table (TET). The HTB facilitates collection of instructions from the processor by storing all the executed instructions while the TET assists fast access to the trace buffer by maintaining a list of trace entry addresses and their positions in the HTB. Since the instructions can be delivered in the form of traces, the processor can now sustain a higher instruction rate leading to overall improvement in the performance [55]. To validate the above proposition the authors display the energy usage
rate of a direct mapped 16kB instruction cache (TRC-512) with various instruction delivery schemes. This instruction cache achieved a total energy reduction of 31% with the lowest energy delay product (EDP) of 0.577 [55]. Reduced complexity in the hardware configuration also enables the TR cache to deliver instructions at lower energy costs and achieve significant energy efficiency.

**Data reuse based Drowsy Mechanism**

M. Bhadauria et al. [56] introduce a Reuse Distance drowsy mechanism depending on workload access patterns to reduce the static and dynamic power consumption in multiple access region caches. This approach led to an approximate power reduction of 16%.

Zhang et al. [57] propose a compiler based strategy to reduce leakage power in data caches. The proposed approach rests on the assumption that at a given time, only specific cache lines holding currently used data are relevant for usage and hence needs to be active. The remaining cache lines can be allocated to a leakage saving state. The authors proposed a pre-activation approach where such relevant cache lines are activated well in advance before data embedded in them is accessed. This approach leads to considerable improvement in the performance. Such cache line access patterns can be identified using data reuse analysis [31]. This compiler based approach aided in a significant reduction in the data cache energy consumption which aggregated to approximately 59.6% with a penalty of less than 5% [57].

**Cache Partitioning**

Reddy and Petrov [58] recommend the effectiveness of an application driven judicious cache partitioning scheme to achieve significant power reduction expended by shared on-chip data caches. This approach ensures that each task is associated with its own cache without interfering with other tasks. The partitioning of cache is done on the basis of cache behavior analysis for an identified set of applications using compile time algorithm and profile based information about the cache behavior. Once this partitioning is identified, the profile based information along with the control signals for configuring the cache is transmitted to the OS, which in turn instigates the cache partition of the concerned pre-empting task. As a result, only a fraction of the cache remains active at any given time, leading to considerable reduction in the dynamic and leakage power [58].
2.3 Conclusion

In this thesis we explore the possibilities of reducing leakage in the data caches by identifying program data regions that have less impact on total execution time of the program. We present techniques to identify these data regions and thereby apply cache leakage reduction mechanisms. We also address the problem of dynamic energy consumption in data caches by using this technique. We achieve approximately 40% of reduction of power consumed by the data caches in a typical embedded processor. Further discussion is presented in chapters 4 and 5. Our techniques are also applicable in reducing DRAM power consumption by allocating the non critical data regions into low power state.

In short, the above discussed literature provides an understanding of how a compiler can help to reduce power in various components through different approaches. We intend to contribute to the existing stream of literature by designing compiler and micro-architectural algorithms to utilize the above outlined program behavioral properties to optimally achieve power savings. In this thesis, we primarily focus on achieving power optimization without significant loss in performance in functional units and data caches.
Chapter 3

Leakage Power Reduction In Functional Units

3.1 Introduction

Static power consumption has become a significant factor of the total power consumption in a system. Circuit-level switching techniques reduce static power consumption by exploiting idle periods in processor components and placing them into low power modes or turning them off completely. In this chapter, we propose a modified automaton-based list scheduling technique that augments the circuit-level techniques by reducing the number of transitions between power modes, thereby increasing the number of idle periods in resource units. Our scheduler uses a global resource usage vector and the usage vector of the last issued instruction(s) to select instruction(s) from the ready list such that resource units common to the last issued instruction and the selected instruction(s) are continuously active without creating a transition into low power mode. We estimate the power consumed in resource units using an energy model parameterized by the number of idle cycles, active cycles and transitions. We have implemented our algorithm in GNU C Compiler (gcc) and our simulations for different classes of benchmarks using an ARM simulator, with single and multi issue, indicate an average energy savings of 4%-15% in resource units with an average of 0.25%-0.8% increase in the number of execution cycles.

An overview of the methodology used in the chapter is shown in 3.1. An input program is compiled with gcc either turning on our proposed scheduling technique or turning it off. A
processor simulator with power gating of functional units and modeling the energy is used to run these optimized binaries. The simulator data is analyzed for savings in the energy.

![Diagram](attachment:image.png)

Figure 3.1: Overview

Power consumption in processor cores has become an important concern both in architectural design as well in compiler construction. With the threshold voltages reaching low values, the static power consumption, which is exponentially and indirectly proportional to the threshold voltages, is dominating the dynamic power consumption in all the units of the processor. Static power consumption in functional units like integer arithmetic unit (ALU), floating point unit (FPU), multiplication and accumulation unit (MAC), data cache ports and data memory buses is also significant. Studies show that with the technology trend static power consumption is going to increase linearly whereas active power consumption is going to remain almost constant [59], the former thereby constituting significant portion of the total power consumption.

Static power consumption can be reduced if the number of continuous idle cycles is increased. This provides a good opportunity to hardware driven optimization techniques such as leakage biased bitlines [60] or input vector control to exploit the idle periods of the functional units so that the unit can be placed in low power mode or turned off completely. As indicated by many proposals, the static power consumption in the low power mode is negligible compared to that in active mode of the unit. If the continuous active period is increased and also continuous idle period is increased then the circuit-level switching techniques can be provided with a lot
of opportunity. This can be achieved by reducing the number of transitions between the power modes and thereby reduce the other significant constituent of the total power consumption, the transition energy. [61][60][59] show that transition energy, depending on the controlling technique vary between 1 to 10% of the active energy. Moreover the transition delay ranges from 3-30 cycles in 0.07μ technology [59] [60]. The number of transitions between active and idle modes can be reduced so that the saved transition delay can be added to the idle period of the functional unit. In this way, continuous idle periods of units can be increased. This reduces the frequent discharging and charging of the capacitance in the circuits of functional units, thus reducing the power consumed due to transitions. It also increases the chances of staying in an idle period or active period continuously for longer durations thereby using the resources more efficiently, both from the performance and power point of view. If only the circuit-level switching techniques are employed in order to reduce the static power consumption, transitions can cause certain amount of performance impact too. So reducing the number of transitions lessens the performance impact too. This chapter focuses on increasing the average idle periods that a functional unit can be in and also on reducing the number of transitions that a functional unit can undergo. Regularization of the usage means changing the pattern with which a functional unit is accessed, so that the functional unit stays in an active state continuously as far as possible without becoming idle and also stays in an idle state continuously for a significant number of cycles. Regularization of the usage of resource units such as ALU, FPU, MAC and memory port (memory unit) can be achieved through static techniques. Hardware techniques need more complex logic to implement these techniques, whereas a compiler writer has the information about these units and their usage patterns are provided to him through the machine description. The total power consumption in a unit or port is given by.

\[ E_{\text{total}} = E_{\text{active}} + E_{\text{transition}} + E_{\text{idle}} \]  

(3.1)

We propose to achieve regularization during instruction scheduling.

The remainder of the chapter is organized as follows. In Section 3.2 we discuss the background, motivation and the energy model that we use to estimate the energy consumption in the resource unit that we are trying to model. In Section 3.3 we discuss the instruction
Chapter 3. Leakage Power Reduction In Functional Units

3.2 Background and Motivation

Different hardware techniques have been implemented in today’s processors to place an idle circuit in low leakage power mode. These techniques implement different controlling mechanisms in order to balance between transition energy and transition delay. More the transition delay, higher the performance penalty but less transition delay (sharp transition) can use more transition energy. The break-even time period is calculated as the minimum time that a circuit should be in idle state so that the overhead of transition is nullified. The break even time is estimated to be anywhere between 10 to 30 cycles. Figure 3.2 shows the behavior of a unit when a circuit-level switching technique is implemented. When there is a transition from a high power mode to a low power mode, there is extra power loss and also a delay that is incurred. The total transition energy loss can vary between 1 to 10 % of the total active energy consumption. If the transition delay is critical, then the transition energy loss can even be almost 50% of the dynamic active energy consumption. Moreover, the static active leakage energy required for being in the active state without the unit being used is estimated to be almost 5-10 % of the dynamic energy consumption. The trend shows that these components are becoming important factors as technology improves. Some proposals try to reduce the break-even time and transition energy consumption [60]. In order to avoid the implications of trade-off, new

Figure 3.2: Behavior of a functional unit. This shows the transition from a high power mode to a low power mode.
methods to reduce power consumption at the architecture level and to use compiler support are essential. Many architectural and compiler research proposals targeted the first component of the equation 3.1. The number of accesses to memory[62][63][57], the cache configuration, [45] the number of bit transitions on the buses[64] and the amount of traffic between L1 and L2 caches constitute a major portion of $E_{active}$. The second and third components are also important and these components have an adverse effect on performance too, i.e., execution time, because of the introduction of transition delays.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>ALU</th>
<th>MUL</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>ld r1, 10[r2]</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i+1</td>
<td>add r1,r1,r4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>i+2</td>
<td>mul r3,r1,r3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i+3</td>
<td>ld r5,10[r2]</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i+4</td>
<td>add r4,r1,0x04</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>i+5</td>
<td>mul r4,r1,r4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i+6</td>
<td>add r1,r5,r3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.3: This is a valid schedule generated by a traditional scheduler.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>ALU</th>
<th>MUL</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>ld r1, 10[r2]</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i+1</td>
<td>ld r5,10[r2]</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>i+2</td>
<td>add r1,r1,r4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>i+3</td>
<td>add r4,r1,0x04</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i+4</td>
<td>add r1,r5,r4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i+5</td>
<td>mul r3,r1,r3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>i+6</td>
<td>mul r4,r1,r4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.4: Scheduling the instructions with a different heuristic

### 3.2.1 Examples

Consider the code segment in figure 3.3. The figure shows how different functional units, ALU, MAC and memory ports (RP) are used. Note that functional unit usage in each line does not correspond to demands of the concerned instruction alone. What is shown is the cumulative demand of previous instruction(s) and the current instruction. The load instruction which is scheduled in the cycle $i$ will use an ALU in the first cycle and a read port RP in the next cycle. The initial stages of the pipeline are not modeled as the initial pipeline stages are common to all the instructions. Every time there is a switch from 1 to 0 and 0 to 1, there is a transition from
the active mode to idle mode and vice versa. As we see the add instruction which is scheduled in the \(i+1\) cycle, and a mul instruction scheduled in \(i+2\) force a transition in the ALU. Figure 3.4 shows how to rearrange these instructions such that the number of transitions is reduced. The two load operations are scheduled one after another, the multiplication operations are scheduled last and the add operations are scheduled closer to each other. This reduces the number of transitions and also increases the continuous idle-period in each functional unit. The schedule in figure 3.3 is a valid schedule and is possibly generated by a traditional scheduler such as list scheduler. The heuristics adopted by a traditional scheduler suit well if the aim is to just avoid stalls and improve the total execution time. But in addition to these challenges, the new transition aware scheduler needs to group or schedule together instructions which use the same set of functional units.

![Image](image.png)

Figure 3.5: A snapshot of continuous idle periods when optimization for transitions is not introduced. Note the short and small number of continuous idle periods (sparse). Y-axis - Continuous idle periods, X-axis - Transitions

### 3.2.2 Dynamic Resource Usage

In this chapter a new optimization technique for reducing the number of transitions is presented. We apply our optimization technique on a benchmark sha. We obtain the continuous idle periods in ALU and transitions from high power mode to low power mode and vice versa. Figure 3.5 and figure 3.6 show plots of these metrics with and without the optimization applied. The
figures are the snapshots of the continuous idle periods in a window of 10000 transitions. The X-axis indicates number of transitions. During runtime, we measure the number of transitions and a window was selected so that we can compare the effect. The total number of execution cycles with and without optimization differ by only 0.6 %. This indicates that in the window that we capture, both the programs execute almost the same part of the program. We observe short durations and less number of idle cycles in figure 3.5. The behavior has been transformed, for the same number of transitions, into long and more number of continuous idle cycles after applying the optimization as seen in figure 3.6. This indicates that there is an opportunity to create extra idle periods in functional units.

![Figure 3.6: A snapshot of continuous idle periods when optimization is introduced. Note the long and large number of continuous idle periods (dense). Y-axis - Continuous idle periods, X-axis - Transitions.](image)

The different functional units that we consider are ALU, MAC, FPU and a memory port. Initially, we assume single issue processors, such as most of the current embedded processors and a single instance of each functional unit. We also apply the technique to multi-issue processors. These units are exposed to the compiler through a machine description of the target machine. The other units, which are implicitly affected by accessing these units are writebuffer, memory buses, buses that connect MAC and FPU co-processors. A transition from active to idle mode can be caused if there is an idle period of more than one cycle. The reason for a functional unit being idle can be a cache miss, branch misprediction, functional unit unavailability being ahead of it, or a dependent instruction blocked in the pipeline. The machine description associates
an instruction with a set of functional units and the time for which each functional unit is used. Thus an instruction can access any subset of these components during any cycle of its operation in the pipeline. The problem that we tackle may be stated as follows:

*By using the resource usage information available in the machine description, regularize the usage so that the number of transitions of functional units are reduced and the duration of idle period of functional unit is increased.* We propose changes to a standard automaton based instruction scheduler which, while scheduling a basic block, schedules similar instructions so that the usage of different functional units remains constant.

### 3.2.3 The Energy Model

An energy model aiming to capture the total power consumption in functional units should take into consideration the following parameters.

1. Number of active cycles and idle cycles of different functional units

2. Static idle energy, transition energy, and dynamic energy which vary with the circuit-level control techniques that are used.

3. The number of transitions in the functional unit.

In addition to functional units, we wish to model the energy leakage in the read ports and write ports of the data cache whose usage information can also be provided to the compiler. There are many power models suggested in the literature like WATCH [65], Simpower [66], and Butts-Sohi [67]. These models can also be used with the appropriate constants like dynamic voltage settings, activity factors, sensitivity factors and leakage factors, but the model which is appropriate to our present discussion is derived from [61]. This model is appropriate as it captures the transition of a functional unit for current technology. The authors consider the model for estimating different policies of their circuit-level leakage reduction scheme with various design and technology dependent constants. The model captures the dynamic behavior of the functional unit through the state of each functional unit and the change in the access pattern of the unit. $E_{active}$, $E_{transition}$, and $E_{idle}$ are given by the following equations.

$$E_{active} = N_{active}(\alpha + (1 - D)p) + N_{active}(\alpha sp + (1 - \alpha)p)$$  \hfill (3.2)
\[ E_{\text{transition}} = N_{\text{transitions}}(1 - \alpha + E_{\text{sleep}}/E_{\text{active}}) \]  
\[ E_{\text{idle}} = N_{\text{idle}}sp \]  
\[ N_{\text{active}} = N_{\text{actualactive}} + N_{\text{transitions}}/2 \]

\( E_{\text{sleep}} \) is the energy consumed in the sleep mode. \( E_{\text{active}} \) is the energy consumed in the active mode. \( E_{\text{idle}} \) is the energy consumed in the idle state. \( E_{\text{transition}} \) is the energy consumed during the transitions between sleep and active states.

\( N_{\text{active}} \) in equation 3.2 is the number of active cycles in which the unit is in active mode. \( N_{\text{transitions}} \) is the number of cycles in which the unit is in transition mode. \( N_{\text{actualactive}} \) is the number of cycles in which the unit is used. \( N_{\text{idle}} \) is the number of cycles the unit is unused state.

\( \alpha \) denotes the activity factor, i.e., the fraction of charged cells to discharged cells in the logic. This factor is dependent on the application. \( D \) is the fraction of cycle period required to charge a cell \( p \) is the leakage factor which is fraction of static leakage in idle state to the static leakage in the active mode, \( s \) is the leakage energy fraction of the dynamic energy.

The latter defined six parameters are dependent on the technology and design of the logic. We choose these values from [61] for all the functional units and ports. This model considers the parameters that we have listed above.

### 3.3 Scheduling Algorithm

We modify the list scheduling algorithm for basic blocks to achieve reduction in the number of transitions in functional units. The modified problem of the scheduling can be stated as follows.

*Given an instruction sequence as a data dependence graph and the resource usage information in the form of bit vectors, minimize the number of bit transitions across the usage vectors of the scheduled instruction sequence generated.*

The algorithm shown in figure 3.7 provides a mechanism for fetching the global resource usage vector by querying the scheduling automaton [68] that is constructed from the machine description provided. Then it computes the "closeness" between the vectors and assigns priorities to each instruction. The ready list is sorted based on the priorities of the instructions.
Chapter 3. Leakage Power Reduction In Functional Units

Figure 3.7: The list scheduler is modified to incorporate the information of global resource usage and compute the closeness of an instruction with respect to the global usage.

The scheduler uses the scheduling automaton [68] to choose instructions one at a time from the ready list and schedules them in successive time slots. If no operation is ready then no-ops are introduced till some instruction becomes ready. The scheduler tries to avoid introducing no-ops as in a traditional scheduler.

3.3.1 Global Resource Usage Vector

An important part of the algorithm is to obtain the global resource usage vector (GRV), which contains information on the usage of each of the functional units during the present scheduling cycle. Some instructions scheduled in the previous cycles continue to consume different resources during the current cycle, thus changing the global usage in every cycle. To obtain this vector, we modify the pipeline description model that is used to detect structural hazards during scheduling. The pipeline description model can be of any form, i.e., a resource reservation table (RRT) or an automaton based model. We use the automaton model proposed by [69] and later enhanced by many authors [68] for two particular reasons:

- This model implicitly captures the global resource usage that we need during the construction of the automaton and,
• This reduces the scheduling time and it is easy to specify new resources other than units that are used by traditional scheduler.

For example we add memory port as another resource component in our description. The number of states increase with the addition of new components in the description. Every state consists of a vector which is the global resource usage, which is updated during the building of the automaton. The modified algorithm to build the automaton is described below. The input is a machine description and it generates the automaton along with the global resource usage information in each of the states. Figure 3.8 shows a snapshot of an automaton to model a pipeline. The figure shows three instruction classes \( i_1 \), \( i_2 \) and \( i_3 \) with their reservation tables and the automaton built using these classes. Each state in the automaton is depicted as \( S_1 \), \( S_2 \), \( S_3 \) etc. and they describe the GRV due to an instruction. The two-column matrix in each state is the resource usage matrix in that state (for two cycles). The first column of this matrix is the GRV for the current cycle. Since our machine model is a single issue model, each state is a cycle advancing state [68] and hence we inspect the second column for any resource conflicts while deciding which instructions are issued during the automaton building process. If there are no conflicts, then as in [68] we left-shift the state matrix by one column and OR it with the reservation table of an instruction to generate the matrix for the new state. The rest of the procedure is the same as in [68].

3.3.2 Closeness Heuristic

The next part of the algorithm is to calculate the closeness between the instructions in the ready list and the global resource vector. From the machine description, for each instruction class we associate a resource usage vector (RUV). For example in the figure 3.8, RUVs are \( i_1: 100 \), \( i_2: 020 \), \( i_3: 101 \). The closeness of the resource usage vectors indicate how many transitions can be possible if the instruction in observation is scheduled in the next cycle. The resource vector corresponding to each instruction class is a weighted vector where each element corresponds to the number of cycles that a particular unit is used. (i.e. sum of the 1’s in each row of the RUV). During every iteration of the scheduling process the global resource vector is OR-ed with the RUV of the last scheduled instruction. In this operation we use a bit-version of an RUV that contains 1’s for its non zero elements to obtain a resource reservation vector (RRV). By OR-ing
the two bit-vectors (GRV and the bit vector acquired from the RUV, i.e., the RRV), the current as well as the future usage of the last scheduled instruction can be captured. Suppose the two vectors are \(< i_1, i_2, i_3 >\) and \(< j_1, j_2, j_3 >\) respectively. The Euclidean distance is calculated as

\[
\sqrt{(i_1 - j_1)^2 + (i_2 - j_2)^2 + (i_3 - j_3)^2}
\]

The Euclidean distance between the global resource and RRVs gives an estimate of how close these two vectors are. If this value for an instruction in the ready list is minimal, then that instruction need to be scheduled as early as possible because it is likely to cause less number of transitions.

In a traditional scheduler, the priority that is defined for each instruction before scheduling, is computed using the length of the dependency chain from the instruction node to a leaf of the instruction. In our algorithm we introduce an additional priority, power consumption priority, that depends on the closeness factor. This priority for an instruction is high (in the current list), if its closeness is low. We sort the ready list first based on the power consumption priority and then on dependency chain length priority.
3.3.3 Working Example

Consider the data dependency graph (DDG) shown in figure 3.9. The nodes are annotated by the instruction id and the functional units they use. \textit{alu}, \textit{mp} indicates that the instruction accesses \textit{alu} in the first cycle and \textit{memory port} in the second cycle. \textit{mul*2} means that the multiplication unit, \textit{mul} is used for 2 cycles, therefore the issue latency is 2 cycles. We assume a single cycle latency for both \textit{alu} and \textit{memory port}. The weight on an edge indicates the result latency. The scheduler output is given in figure 3.10. The second column indicates the dependency chain priority. The last column indicates the closeness values for the instructions in the ready-list in the same order as they appear in the \textit{list} column. With a traditional list scheduler, instruction C gets higher priority than E because of its dependency chain length. Thus C gets scheduled earlier than E. But when the closeness heuristic is used, E gets higher priority than C, and thus E gets scheduled earlier and closer to A and B. Similarly, after D is scheduled, with traditional heuristics E gets scheduled. But considering closeness criteria, instruction G is scheduled after instruction D as they both use the same functional unit. GRV (shown in the sixth and ninth columns) indicate that the continuous idle cycles of ALU, MP and MUL have increased. The number of transitions in ALU and MUL has reduced from 4 to
3.4 Experimental Results

The scheduling algorithm is implemented in GCC 3.4 in the postpass scheduling phase. The compiler is a cross compiler which generates ARM v5 code. An XScale simulator XTREM [71] which simulates the XScale [72] pipeline and also computes the dynamic power consumption in each of the functional unit is used. The Xscale core has three pipelines one for each of memory, multiplication and ALU operations. The address calculation is done at the ALU stage, whereas multiplication operations are decoded and issued to MAC which is a co-processor. We assume a similar pipeline for floating point operations and have included a floating point unit in the simulator. As XScale\(^1\) does not have floating point instructions we did not tune the
Table 3.1: Processor Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Bandwidth</td>
<td>1</td>
</tr>
<tr>
<td>Issue Width</td>
<td>Single issue In order</td>
</tr>
<tr>
<td>FUnits</td>
<td>IntALU-1,MAC-1</td>
</tr>
<tr>
<td></td>
<td>FPU-1,Memport-2</td>
</tr>
<tr>
<td>Data Cache</td>
<td>L1 32KB fully associative</td>
</tr>
<tr>
<td>I-Cache</td>
<td>L1 32KB fully associative</td>
</tr>
<tr>
<td>Predictor</td>
<td>128 entry direct</td>
</tr>
<tr>
<td></td>
<td>2 bit predictor</td>
</tr>
<tr>
<td>Physical Registers</td>
<td>32</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.75 0.5 0.25</td>
</tr>
<tr>
<td>$p$</td>
<td>0.05</td>
</tr>
<tr>
<td>$s$</td>
<td>0.01</td>
</tr>
<tr>
<td>$E_{\text{sleep}}/E_{\text{active}}$</td>
<td>0.001</td>
</tr>
</tbody>
</table>

gcc to generate XScale code rather used the ARM ISA. The floating point instructions are decoded and issued to a floating point unit. The simulator collects the number of active and idle cycles and the number of transitions that have occurred in each unit modeled. The usage of each functional unit is monitored and statistics are collected in each cycle. The benchmarks are chosen from MIBENCH [73] (adpcm, sha, jpeg, blowfish, susan), MediaBench [74] (mpeg2, gsm), NetBench (dh) and Spec2000 (bzip, meso) [75] suites. The minispec [76] data is used for simulated SPEC benchmarks. The effectiveness of algorithms is determined by the instruction mix in the programs and the available parallelism so that the dependency across different types of instructions makes the scheduler harder to optimize.

We collect metrics for two different schemes. The first scheme is with only circuit-level switching techniques being implemented and the second one has circuit switching techniques and the scheduling algorithm being applied on the code. We calculate the power consumption using the energy model described in Section 3.2. The processor characteristics and the parameters of the energy model are shown in Table 3.1. The simulator is modified to capture the number of cycles during which the unit is in idle state, the number of transitions between power modes and the number of cycles during which the unit is active. Two counters are maintained for each functional unit, one for idle cycles and the other for transitions. The total

\footnotesize{1The Xscale simulator [71] already has a code for a floating point unit but it was disabled, we simply enabled it.
number of active cycles is calculated as the difference between total number of execution cycles and the number of idle cycles. The counters are implemented as depicted in the finite state automaton shown in figure 3.11. The transitions occur every cycle depending on the functional unit state. After these values are collected, the energy model is used to calculate the total power consumption of different units. The statistics are collected for various values of activity factor $\alpha$: 0.25, 0.5 and 0.75. We chose these activity factors as [61] indicates that these are reasonable values for most of the circuit logic that is used to implement functional units.

3.4.1 Power Consumption Estimation

The energy savings that are plotted are the percentage decrease in the total power consumption of the second scheme with respect to the first scheme, i.e., circuit-level switching technique is implemented without out using the scheduling algorithm. The actual power consumption without any circuit-level technique is not shown. Figure 3.12 shows the energy savings for an Integer ALU functional unit for various activity factors. With an increase in activity factor the impact of static leakage energy seems to be reducing. Most of the simulated benchmarks show a significant impact on the usage of this functional unit. As an ALU is used for integer arithmetic operations as well as for memory operations, bringing these operations closer while scheduling, creates a gap between the usages of the ALU. This can be due to the change in the pattern in which dynamic events such as L1 data cache miss occur. In XScale, the load operation result

Figure 3.11: State diagram to depict the functional of the counters to count number of idle cycles and number of transitions in each functional unit.
Chapter 3. Leakage Power Reduction In Functional Units

Figure 3.12: Percentage energy savings ALU

latency is 2-3 cycles and this causes a dependent instruction to wait for the result. In such a case without our scheduling technique, the traditional list scheduler would make the pipeline stall and this causes a transition in the ALU. But with our technique an opportunity for scheduling a low priority instruction ahead of a high priority instruction exists (based on closeness), This results in avoiding a transition. Current circuit logic techniques, which exploit very short intervals of idle periods to save power consumption, may start a transition in the unit from high power mode to low power mode even for short idle periods. In some of the benchmarks like jpeg and mpeg the total number of execution cycles reduce because of this effect but this increases energy consumption. An average of 5% of reduction in the total power consumption is achieved. Figures 3.13 and 3.14 show the impact of the scheduling technique on mul and floating point functional units respectively. As observed earlier, these units are hardly utilized by most of the benchmarks. This may be due to strength reduction optimization applied on these benchmarks. mesa, jpeg, gsm and fft show a significant savings in power with respect to these units as there is a good mix of instructions that use these units in these benchmarks. The total power savings on an average are 7-10% in MAC unit and 4-7% in the floating point unit. The remaining benchmarks do not have any significant activity in these units. We have also measure the memory port power consumption for these benchmarks. We model the port of the
Chapter 3. Leakage Power Reduction In Functional Units

Figure 3.13: Percentage energy savings in MUL unit

Figure 3.14: Percentage energy savings FPU
data cache. Traditionally, data cache usage access patterns are observed to be non coherent. This is due to the low temporal locality that is exhibited by these programs. Figure 3.15 shows that the amount of energy savings is not much when compared to ALU.

When circuit-level switching techniques are implemented, the transition delay would cause the total number of execution cycles to increase. With our scheduling scheme proposed in this chapter, the total number of transitions are reduced. This has an impact on the penalty due to the transitions. The performance penalty reduction in the total number of execution cycles has also been calculated for both the schemes. The extra execution cycles incurred due to circuit switching techniques is given by

\[ N_{actual\_execution\_time} = N_{execution\_time} + N_{transitions} \times penalty \] (3.6)

The penalty can range from 3 to 30 cycles depending on the technology, control characteristics and the circuit-level optimization technique. We choose a moderate figure of 5 cycles as penalty due to [59]. The penalty reduction in usage of ALU determines the overall penalty as this is the most used functional unit. This is shown in figure 3.16

3.4.2 Multi-Issue Processor Energy Estimation

We use the heuristic and compile the benchmarks for a processor with issue width 4. We assume a hypothetical in-order machine with multiple ALUs and multiple FPUs. We assume that the functional units are used in some order and the issue logic issues instructions to the
Figure 3.16: Percentage savings in the performance penalty over a circuit-level technique without our scheduling technique being performed

Figure 3.17: Percentage energy savings in MultiIssue model with Multiple ALUs

Figure 3.18: Percentage energy savings in memory port
units in this fixed order. This means that in case of multiple ALUs, ALU1, ALU2, ALU3 and ALU4, an ALU instruction is issued to ALU2 only if ALU1 is busy. $N_{idle}$ and $N_{transitions}$ are measured for all the ALUs and FPUs together. Fetch bandwidth has been increased to 4. In the scheduling automaton not every state is a cycle advancing state. The GRV is the first column in resource usage matrix of the cycle advancing state. The remaining parameters are similar to that of the Xscale processor. The findings, as shown in figures 3.17 through 3.20 indicate a significant reduction in the power consumption except in benchmarks like jpeg decode and gsm toast. Lack of significant power savings in these benchmarks, i.e., jpeg decode and gsm toast can be attributed to the reduction in the number of idle cycles. The total number of execution cycles in these two benchmarks are estimated to have decreased by 0.5-1.0 % indicating a
reduction in the number of idle cycles. The savings in the FPUs are marginal. This is because of the scattered occurrence of the FP instructions even after the Closeness heuristic is applied.

![Figure 3.21: Multi-issue processor performance penalty](image)

Figure 3.21 indicates the reduction in the penalty compared to a processor with only the circuit-level switching techniques and no closeness heuristic scheduling applied. Figure 3.21 shows that the performance penalty on an average is less than 1% as compared to the 3% on a single issue processor. *gsm-toast*, *jpeg encode* and *decode* exhibit reduction in the number of cycles.

### 3.5 Related Work

There have been a number of proposals to reduce total power consumption through different compiler techniques, targeting different units of the system.

In [77] the power consumption in the functional units is reduced by constructing a power aware control flow graph during compilation and performing static analysis over it to analyze the usage of each functional unit in a basic block and insert specialized instructions at the beginning and end of the basic blocks. These instructions are used to control the functional units. In their methodology, even if we assume that the ISA can be extended without extra frills, it still calls for certain amount of hardware support at the decoding stage of the instructions. Moreover, the technique does not consider the idle times in ALU units and Ports. But as shown in [61] the profit of turning these units off for shorter durations is sizable. In [78] the authors provide a theoretical limit for the peak energy consumption in functional units with an unbalanced distribution of instructions. This limit used in [79] reduces the peak power consumption in functional units. The aim of these studies is to reduce the variation in power consumption due
to various functional units by balancing the usage throughout the program. But the techniques result in very less average idle cycle time for each functional unit thereby increasing the static power consumption. Static power consumption is increasingly becoming a bottleneck as the technology is reaching 0.07um stage and the leakage power assumes a significant proportion of the total power consumption. This motivated the current study where we examine whether increasing the average idle cycle time of functional units leads to significant reduction in power consumption.

In the context of VLIW data paths, [64] proposed an instruction scheduling scheme to reduce the power consumed by the instruction bus in the fetch unit. They provide an optimal algorithm for horizontal scheduling which schedules micro instructions’ packets and vertical scheduling. This provides an order inside each instruction packet. In [57] VLIW functional units are assumed to run at various voltage levels and instructions are issued to these units based on the slack for each instruction. Their technique exhibits a significant reduction in power usage when the performance degradation threshold is very large. In addition, the transition energy costs in the functional units when they operate at different voltage levels are not considered. In [80] a critical path predictor is used to assign critical instructions to functional units that operate at high voltage levels whereas other instructions are issued at low voltage level units. The available benefit varies with the accuracy of the branch predictor, which costs additional power.

### 3.6 Conclusion

In this chapter a scheduling technique is proposed through which the number of transitions between active mode and idle modes in the units ALU, MUL, FPA units and memory ports are reduced, thereby increasing the number of continuous idle cycles of each unit. This reduces the static power as the unit remains in an idle state for a longer time. Moreover, the reduction in the number of transitions reduces the transition delay and transition energy which significantly impact the total execution time and power consumption when circuit-level optimization techniques such as biased bitlines or input vector control are used. We evaluate this technique with an analytical energy model available in the literature. Simulation results indicate a significant reduction of about 4% to 15% in the total energy consumption of the
units and also less than 1% of the performance penalty. In short, this work demonstrates how the grouping of "similar" instructions in more complex architectures with large issue width results in power savings in the units.
Chapter 4

Leakage Power Reduction in Data Caches

4.1 Introduction

The previous chapter focused on the reduction of leakage power in functional units. We established that the nonuniform utilization of these functional units could be exploited for saving leakage power. The current chapter focuses on the cache subsystem, which constitutes a primary power consuming component of a memory subsystem.

In the context of memory subsystems, several parameters like per-memory access latency, per-memory access power consumption, and die-area have a direct and significant impact on the overall power efficiency of embedded computing systems [81]. Various subsystems like data caches, RAM and the disks constitute the memory subsystem.

4.1.1 Cache Leakage Problem

Cache memories emerge as significant contributors to the leakage power problem because they constitute a significant proportion of on-chip transistors in embedded processors [82]. To quote a few examples, on-chip caches occupy 43% of total chip area in SA-110 [83] and about 75% in Itanium-2 [84]. They consume 30% of the total power in the processor cores like StrongARM and 16% of the total power in Alpha 21264. This attracts considerable attention of designers and justifies the need to achieve circuit and architectural optimizations in order to reduce leakage power.
Modern embedded systems like smartphones and tablets provide a platform to run computation and memory intensive applications. The underlying architecture and software layers support the application to execute efficiently both in terms of power as well as performance. As the applications become data intensive, it is imperative to reduce power on the data fetch and storage mechanisms. The current multi-core embedded processors employ a multi-level cache hierarchy with last level sharing across the cores. The front line caches like L1 (or in some cases L0) operate on data private to a thread or a single task. The design of such caches is extremely critical to mitigate overall cache power consumption. One of the significant cache designs in terms of power is a drowsy cache. The drowsy cache has the capability to maintain and preserve the data between different power states and also requires few cycles to switch between these power states.

In this chapter we present a novel software-centric mechanism to utilize the drowsy cache with support from program profiling and cache hints in the instructions. The majority of the traditional data related optimizations focus mainly on size of the data, temporal and spatial locality and frequency of accesses to a data region [85]. We introduce a new program-level characteristic that captures the latency tolerance of data regions. This characteristic brings into perspective the criticality or the impact of different data regions to the program performance which fundamentally differs from the other traditionally used characteristics.
Chapter 4. Leakage Power Reduction in Data Caches

4.1.2 Contribution

We address the problem of power consumption in a data cache by focusing on the latency tolerance of the data regions. Latency tolerance of a data region indicates the criticality of the data region to the overall performance of the program. Figure 4.1 provides a brief overview of our approach.

We implement a context sensitive, profile-driven analysis, to identify disjoint data regions. The length of the calling context and dynamic path length are taken into account.

We now define criticality of the data regions identified in the previous step (profile-driven analysis). We apply a critical path model of a superscalar out-of-order pipeline and quantify latency tolerance of all the memory operations. A memory operation can be either a hit or a miss at some level of the cache hierarchy. This means that the latency of a memory operation could vary in value depending on the particular architectural event. If the impact of this latency changes the instructions per cycle (IPC), then the memory operation is considered critical. By considering the latency tolerance of these memory operations, we analyze the criticality of a disjoint data region. The initial analysis is carried out on test inputs. All the memory operations have high latency compared to other operations. Some of the memory access latency is absorbed by dynamic instruction scheduling or cache hits. A memory operation is critical if, "it lies on the critical path of the execution". For our research, we define critical data regions as the regions of data that are accessed by critical memory operations.

We apply this analysis to steer power-aware optimization techniques by fetching data regions into different L1 caches operated at different leakage control modes. This technique can be used to save around 30% of the total power and 20% of leakage power in the data cache without any significant performance penalty.

In our experimental methodology, we have considered a scaled down processor with reduced data cache compared to a multi-core processor model. This methodology focuses on a single task and characterizes critical regions for such a task simplifying the execution model of benchmarks. This model closely resembles a multi-task execution model on a multi-core processor with each task mapped to a core with reduced cache size. In our methodology we profile a single task on a single processor model. Memory accesses from other tasks or other processes are not considered during data profiling.

We also study the variation of criticality of the data regions across different phases of
program execution. To the best of our knowledge, our work is the first one that describes power-aware optimization using critical data regions.

4.2 Background

Most of the compiler-directed power optimization techniques proposed in the literature operate at the instruction level. While per-instruction based techniques perform effectively in driving fine-grained optimizations, they do not capture the coarser level semantics of programs. Programmers implement meaningful procedures and data structures in their applications. Such coarse-grained information can be utilized in optimizations and can gain significant power savings.

Figure 4.2: An example from the benchmark vpr. Boxed statements are more critical. Architectural model to control cache lines based on critical path.

Consider an example drawn from the vpr [75] benchmark shown in figure 4.2(a). Let us first consider the scenario when per-instruction based optimizations are to be carried out. As shown in the figure, a significant portion of the total memory accesses is concentrated towards two data structures net_bloc Moved and net. The branch instruction in line 7 depends on the memory accesses to net. A branch instruction is a critical instruction. Therefore, these accesses can be considered as more critical when compared to other accesses to the same data structure as
depicted in lines 3, 10 and 11. When the memory operation at line 7 is encountered, the dynamic criticality predictor (derived from the critical path model as explained in [86]) considers that access as critical. Consider that all the fields of net belong to the same cache line. When this cache line is accessed at lines 10 and 11, the predictor considers these accesses to be non-critical. During the execution of the program, this continuous change in the state of the instruction could lead to sub-optimal behavior of an optimization. For example, a cache controller may continuously change the state of cache lines considering the criticality of data accessed by these individual instructions. This demonstrates that considering instruction criticality in isolation for the purpose of classifying data as critical or non-critical is not sufficient for an optimal power-saving technique. We need to take into account the region of memory that gets accessed by non-critical data accesses and apply an optimization on the entire region (for example place the entire region into a low-power state).

Consider a hypothetical processor core with multiple private L1 caches with a shared L2 cache as shown in figure 4.2(b). During profiling, we monitor the critical path of execution and identify the data region that is predominantly accessed on this path. These memory instructions can be annotated with additional hints. During run-time, we control the state of a cache region based on its criticality. If a data region is accessed predominantly on the non-critical path, we maintain those cache lines in drowsy mode. In figure 4.2(b), net and net

belonging

bloc

moved

are accessed on the critical path (as seen from example 4.2(a)), and therefore these data regions need to be maintained in non-drowsy state. The rest of the data regions or data structures can be maintained in a drowsy state in the cache.

4.2.1 Attributing Criticality to a Data Region

Memory access latencies and branch-load dependencies are primary reasons for causing a bottleneck in the system. For instance, a memory access to a data region may be a hit or go through a series of misses. This latency may vary for different memory accesses. Similarly, in the case of branch-load dependencies, a branch misprediction can delay a memory access to a particular data region. Hence such events define the criticality of a data region. We identify these critical data regions through profiling. This helps us to drive power-aware optimizations at the data region level.

In a nutshell the critical path model of an out-of-order processor proposed by [86] can be
summarized as follows. The performance characteristics of a program including the architectural events and inherent data dependencies are modeled by a dynamic dependency graph. Each node in a dynamic dependency graph represents a mode of instruction. An instruction can be in the dispatch mode $D$, the execution mode $E$ or in the commit mode $C$. These three nodes denote events within the machine pertaining to the instruction. The model captures different dependencies between these modes across the instructions. For each instruction there is a dependency between $D$ and $E$ and between $E$ and $C$. This indicates the execution of the instruction, i.e. dispatch followed by execution and finally commit. The model also captures dependencies between two instructions. A $DD$ edge captures in-order dispatch, $CC$ captures in-order commit and $EE$ edge between two instructions captures data dependency. A mis-predicted branch’s execution outcome is needed to dispatch instructions from the right target. Such control dependencies are captured by $ED$ edge. A reorder buffer dispatches instructions in program order. The size of this buffer places a constraint on the dispatch of new instructions unless the instruction at the head of the buffer commits. Such resource constraints are captured by a $CE$ dependency edge. Each of these edges is attributed with a latency, indicating the time taken for the architectural event. The commit of the last instruction ends this dependency graph. The edges of $EE$ are weighted with sum of latencies due to execution and waiting time for functional unit.

By traversing backwards on the longest path, we can identify the critical path of the program. The memory instructions on the critical path are the critical memory operations. We run different programs on this model and identify critical memory operations. The regions of data accessed by these critical memory operations are defined as critical data regions. Fields et al. [86] also predict the criticality of in-flight instructions by planting tokens through the dependency edges. The critical path predictor aids in the accurate prediction of the critical path. This further helps in tuning architectural parameters which harmonize with the critical instructions thereby improving the performance.

An example of this model is shown in figure 4.3. The figure shows a dynamic trace of a program. Each instruction is represented by three nodes as described earlier. The instructions $I2$ and $I1$ have $EE$ edge because of data dependency. Instructions $I1$, $I3$ and $I5$ are constrained by reorder buffer (ROB) size of length 2. Instructions $I5$ and $I6$ have a branch mis-prediction dependency. Here to denote a mis-prediction latency, there is a latency of 6 cycles between $E$
Figure 4.3: Fields critical path model. This example shows a dynamic trace of instructions and dynamic dependency graph.

node of $I_5$ and $D$ node of $I_6$. Though there is latency associated between EE nodes of $I_3$ and $I_6$, this dependency is neutralized due to ROB size dependency. When $I_3$ is in the ROB, $I_5$ cannot be issued. So, by the time $I_6$ is issued the value of $r6$ is written by $I_3$ and there is no dynamic latency associated with it.

By traversing backwards from the last instruction’s commit node and taking the maximum latency edges, the critical path can be identified. The darkened edges indicate the critical path in the instruction trace. Using a trace variant of this model, where we generate the trace of the program and feed it to a backward pass model as explained above, we identify the critical path and critical instructions. We attribute disjoint data regions as critical or non-critical by measuring the criticality of these data regions whenever they get accessed on the critical path.

The Fields model also has a predictor model, where by maintaining history of the event latencies, instructions are predicted either to be critical or non-critical. This model needs significant amount of hardware to maintain the history and prediction tables.

In modern application more than 90% of the total accesses are to either heap-allocated or stack-allocated data regions. Hence it is profitable for any power-aware optimization to be designed for such data accesses. A dynamically allocated data structure is a set of data
addresses distributed across the memory space in a non-contiguous manner. The memory, which is required by a data structure, is allocated (using malloc or any other custom memory management method) and accessed at different points of execution. The initial task is to identify data regions whose accesses do not originate from the same set of memory instructions. We define them as *disjoint data regions* (DDRs). In our technique we utilize this disjoint nature of accesses during profiling and apply criticality model. During execution, we exploit this to reduce power state variations in the cache.

### 4.3 Identifying Disjoint Data Regions

Identification of disjoint data regions (DDR) is done in two stages. First, we profile allocation sites and memory operations such as loads and stores. By profiling these allocation sites and collecting the access behavior of these allocated regions, we generate an undirected graph where each node is a static memory instruction (PC value of memory accessing instruction). This undirected graph is called Data Relationship Graph (DRG). Next, we use a clustering algorithm to create disjoint data regions. Each DDR can be mapped to a data structure in a program or a logical data region of a data structure.

#### 4.3.1 Context-Sensitive Profiling

The LLVM compiler provides data structure analysis for automatic pool allocation [87], where data regions corresponding to a particular data structure (for e.g., a linked list) are allocated contiguously in memory. Their analysis is a context-sensitive, flow-insensitive, alias analysis. It captures heap-allocated data structures. They perform alias analysis at each of the address references. By using the alias analysis results as well as heap allocation tables, they map malloc sites to the references. They do not consider program paths. However, for some benchmarks, program paths also impact the allocation criteria significantly. Moreover, static program analysis is conservative and is an expensive technique both with respect to computing as well as memory footprint when used to perform a context, flow and path analysis. Static program analysis conservatively creates large data regions, thereby impeding the opportunities. Considering the above constraints, we suggest that profiling an application with representative inputs before performing this analysis could be a more acceptable and scalable technique. We follow
Figure 4.4: The example considers malloc sites whose characteristics change depending on the context. This is a code snippet extracted from the parser benchmark. The path to the malloc site alters the way in which the created data is accessed.

Data partitioning techniques [88][89] have been used to map memory operations to data addresses. They distribute data into different cache partitions by forming a relationship between memory operations and individual data addresses. The authors do not consider context-sensitive profiling which is crucial for SPEC integer benchmarks, where dynamically allocated data significantly depends on the context as well as the path in which it was allocated. Moreover, it should be noted that considering individual data addresses and creating relationships could lead to finer granular regions, where applying criticality based techniques becomes infeasible.

An Example: In the parser [75] benchmark, a table is used for most of the computation. This table consists of linked-lists to store words along with the attributes that are associated with these words. Each entry of the table stores a single sentence and the table is initialized for each sentence. The connector variables in each table entry are updated through other functions.
Each of these variables points to a list of attributes that vary across sentences. During the dictionary lookup and parsing of the sentence, the Table-connector data structure is accessed frequently. The computation logic for each of the sentence varies depending on the grammatical notations.

Consider the example shown in figure 4.4 derived from the parser benchmark. The program is instrumented for collecting data addresses allocated by each malloc call, represented as a SiteID. In addition, we maintain an abstract heap table which comprises of the start address, malloc site\(^1\), and the size of the data allocated. Whenever there is an access to a certain data address, a lookup on the heap table provides the malloc site id which is associated with the data address. Thus we create a mapping between malloc site id and program counter value (PC) of the memory operation. We use a program instrumentation and analysis tool, ATOM [90], for profiling. Note that the free routines which deallocate memory seldom have an impact due to rare occurrences of address reuse.

The function indicated in figure 4.4 allocates data for the parser benchmark. The context of the malloc (in this case *xalloc*) site, which can be obtained either by a string of procedure calls (calling-context) or a string of branches (path), varies during the execution of the program. Here we obtain the context of the malloc site through a string of branches. The instrumentation points for branches and memory allocation sites are indicated in bold letters. As seen in figure 4.4, BR1, BR2, BR3, JMP5, JMP6 indicate branches. Here we consider only the last branch before the allocation happens, as the context.

The first table is a context table which is maintained to record the context. Each entry has a context and we associate a path id with it. If the context of the malloc site is obtained by a string of function calls (calling-context), we can associate a context id to each of these strings. For example, in the figure the path string BR1 is associated with Path ID 1.

The second table presents the heap table. This table stores each of the memory allocation sites and the range of (beginning and ending) addresses of the blocks allocated in an instance of execution. During the execution of the program, we encounter a series of branches. As maintained in the heap table, the context at a particular execution point indicates the corresponding context id or path id, depending on the type of the context. A data region is represented by a set of ranges of data addresses at each malloc site along with a corresponding path id or a

\(^1\)An identification code based on program counter value of the malloc call.
context id. The SiteID 1 has two dynamic instances as shown in the heap table. These two instances are identified with the path IDs 1 and 2, due to BR1 and BR2. There are three tables in this program. We can identify these three tables from the three malloc sites.

The above example indicates that the data regions allocated at various points of execution are dependent on the input sentence in the parser benchmark. By profiling these allocation sites and collecting the access behavior of these regions, we generate groups or clusters of data regions that can be deployed for further analysis. This motivates us to consider context-sensitive profiling.

**Algorithm 1** Clustering Algorithm

**Input:** an instrumented program P

**Output:** set of disjoint data regions

1: C Context length initialized to 1
2: D Number of disjoint data regions
3: DRG is Data Relationship Graph
4: while D keeps changing do
5: createDRG(P) //Algorithm 2
6: D=createConnectedComponents(DRG) // Algorithm 3
7: C=C+1
8: if Limitation Guidelines Encountered then
9: report warning
10: exit
11: end if
12: end while

4.3.2 Data Relationship Graph

Here we describe an algorithm to create disjoint regions of data allocated during program execution and form clusters of static memory operations (PCs) accessing these disjoint regions. While a particular instance of a memory operation (a memory access) maps to a single malloc site, a static memory operation may map to more than one malloc site. In order to capture
Algorithm 2 Data Relationship Graph Creation

1: includeDRG($O_i$): Procedure to include a node in DRG
2: createEdgeDRG($O_p$, $O_q$): Procedure to create an edge between $O_p$, $O_q$ in DRG
3: pushContext: Procedure to push a branch up to length C
4: ContextTable: A map between context and $M_j$
5: HeapTable: A map between malloc site and data address
6: $i = 1, 2, 3...O_i$ indicates memory operations
7: $j = 1, 2, 3...M_j$ indicates malloc site instances
8: 
9: **Procedure** CreateDRG($P$)
10: 
11: for each dynamic Instruction of $P$ do
12: 
13: if Instruction is a Control instruction and jump to Address then
14: if Considering path then
15: currentContext=Push(Context, Address)
16: 
17: end if
18: else if Instruction is a function call to an Address then
19: if Considering call string then
20: currentContext=Push(Context, Address)
21: 
22: end if
23: else if Instruction is a malloc call $M_p$ then
24: ContextTableInsert(currentContext, $M_p$)
25: 
26: HeapTableInsert(mallocSite,createdData)
27: 
28: else if Instruction is a memory operation $O_p$ then
29: effadd=EffectiveAddress(Instruction)
30: 
31: context of $O_p$=ContextTableQuery(HeapTableQuery(effadd))
32: 
33: if $O_p \in DRG$ then
34: if $\forall O_q :$ context of $O_q$ = context of $O_p$ then
35: createEdgeDRG($O_p$, $O_q$)
36: 
37: end if
38: else
39: includeDRG($O_p$)
40: 
41: end if
42: end if
43: end if
44: end for
this complex relationship between malloc site instances and memory operation instances, we first introduce a **Data Relationship Graph (DRG)**. DRG is defined as follows:

**Definition 4.3.1.** A data region is a set of data addresses.

**Definition 4.3.2.** Let $M_i$ be a set of data regions (denoted by $D$) allocated from an allocation site $S$ at different instances of execution. Let $a$ and $b$ be program counter values. Let $M_a$ and $M_b$ be the set of data addresses accessed by $a$ and $b$ respectively. We define the data region $D$ as a common data region, if $M_a \cap M_b \subset D$.

**Definition 4.3.3.** A data relationship graph is defined as $G=(V,E)$, where $V$ is the set of nodes corresponding to the static memory operations, i.e., program counter values of memory accessing instructions and $E = \{(a, b) | a, b \in V, \text{ and } M_a \cap M_b \neq \emptyset \}$

Algorithm 1 describes the high level clustering routine for generating disjoint data regions. An instrumented program, as described earlier, is used to generate these data regions. We consider a context in the form of a sequence of branches or string of function calls. We increment the length of the context until we reach a fix point with respect to the number of disjoint regions.\(^1\) The inner loop of the algorithm creates the data relationship graph for a particular context length. It is to be noted that differentiating a call-site can be achieved with increase in the context length. The context length is limited for a program run, though it may impede system resources. Therefore this algorithm guarantees that we reach a fixed point. An exit criteria that can help the execution of algorithm practical without stressing the system resources and quickly able to tune it for different benchmarks is explained in the subsequent section.

### 4.3.3 Algorithm for the Creation of DRG

Algorithm 2 describes the creation of a data relationship graph. During instrumentation, we instrument branch, jump and memory instructions and also malloc sites. The memory operations are indicated as $O_i$ and malloc sites as $M_i$. The instrumentation routine inserts profiling functions before each of these primitives. Here we can recall that the heap table and context table discussed earlier maintain mappings between data addresses, malloc site instances and current contexts.

\(^1\)Here we increment the context length by one, but in general depending on the availability of the memory one can choose to increment it by more than one.
The algorithm performs different operations for each type of instruction as described below:

- **If the instruction is a branch**
  
  During the execution of a program, we may encounter a conditional jump or a branch. If we consider a path as an indicator of the context, we push the encountered branch target address onto a context-stack. The size of the context-stack is pre-defined in the outer loop.

- **If the instruction is a function call**
  
  If we encounter a function call, which is a jump instruction, we push the callee address onto a context-stack in case we consider call string as an indicator of the context.

- **If the instruction is a call to a malloc $M_p$:**
  
  If we encounter a call to the malloc routine, we maintain the data addresses created at that instance in the HeapTable. For the corresponding malloc instance, we consider the current context from the context stack and update the ContextTable.

- **If the instruction is a memory operation $O_p$:** During execution, each memory operation is instrumented to capture the effective data address. Effective data address is the actual data address obtained by adding the base address and the offset. With this effective data address we lookup the HeapTable. This yields us the malloc instance and this is used to obtain the context from the ContextTable. The query operations, ContextTableQuery and HeapTableQuery provide us with the context at which the corresponding data address was created.

  While executing the memory operations, if the context identifier of any operation matches with the context identifier of the current operation, it indicates that these two operations have accessed data created at the same context. Therefore we insert an edge between these two operations. Each operation can have more than one context identifier, since it can access multiple data addresses during execution at different times. The DRG captures the mapping between memory operations and accessed data regions. The connected components of DRG are identified using a variant of Depth First Search (DFS) based algorithm as described in Algorithm 3. During DFS, we check if a node already belongs to a component. If it does, all the nodes reachable from that node associate with the same component identifier. During
DFS, if a node is visited, we annotate the node with the component id and include the node into the set of nodes of the component id. We obtain a forest of disjoint components. Each component can now be considered as a disjoint data region.

**Algorithm 3** Creating Connected Components

**Procedure** `createConnectComponents()`

- N is set of nodes in DRG
- C is set of connected components
- $C_i$ is connected component $i$, set of nodes

while $N \neq \emptyset$ do

  node $\in N$

  $N_{node}$ is set of nodes reachable from node

  $\forall n \in N_{node}$

  if node $\in C_i$ then

    $C_i = n \cup C_i$
    $N = \{N\} - C_i$

  else

    create new component $C_{new}$
    $C_{new} = n \cup C_{new}$
    $N = \{N\} - C_{new}$

  end if

end while

### 4.3.4 Relevance of Context Length

One of the important parameters to the algorithm and also used for context sensitive profiling is the context length. Context length directly affects the number of DDRs that are formed using profiling and clustering. Evidently, as we increase the context length, the number of edges in the DRG increases and then decreases. Hence, identifying the right set of DDRs (considering context length) is crucial for the efficient operation of any optimization technique. In this section, we elaborate with the help of an example, how the number of DDRs varies with respect to the context length.

An example of code for memory allocation that usually occurs in the benchmarks is shown
Chapter 4. Leakage Power Reduction in Data Caches

Figure 4.5: This example considers malloc sites whose characteristic would change depending on the context. The path to the malloc site changes the way the created data is accessed.

in figure 4.5. The figure depicts four paths that lead to the actual allocation site S1. At S1, the program calls a custom allocator, which gets fixed values for the variables p and q in all the four paths. In this example, the path for the creation of data structure p is S1, S2, S4 where as that for the data structure q is S1, S3, S4. There are two functions foo and bar which access these data structures. The function bar has two loads from q, whereas foo has four loads from p. There are three cases to consider:

**When the path length is zero:** In this case the graph has too many disjoint regions and it cannot capture the essence of data regions. This gives scope for a lot of interference across the operations since the data addresses share a memory block. For example, all the addresses of data structure p might be available in a few other memory blocks as well. This does not meaningfully capture the meaning of data regions. The resulting DRG is shown in figure 4.6(a). **When the path length is one:** The heap table associated with site S4 captures all the data that is allocated at that site. In other words, at S4 both p and q are allocated. The DRG formed by using path length 1 is shown in the figure 4.6(b). All the memory operations in both the functions are connected to each other. This is because of the following reason.

When we look up a data address and associate the context for that instance, we get S4 as the
context identifier for all the accesses. The context S4 along with the site, identifies all the data allocated for both \( p \) and \( q \). \textbf{When the path length is two:} By increasing the path length to 2, we are able to distinguish between the two allocations. The paths now considered are S2, S4 and S3, S4. The DRG in this case is shown in figure 4.6(c).

In figures 4.7(a) and 4.7(b), DRGs for Parser benchmark with path lengths 2 and 5 respectively are shown. In figures 4.7(c) and 4.7(d), we show DRG generated when call string is considered in perl benchmark. These DRGs indicate an increase in DDRs when either path length or call string length are increased. It is to be noted that there is a significant increase in the DDRs from path length 2 to 5, whereas not much as if we consider call string. Essentially, the call sites can be differentiated with a small path length in perl. Also, the incremental change is small after certain path-length or call-string length.

As seen in the above three cases, the total number of disjoint components tends to increase with the context string length (length=0 is an exception). The reason for this increase can be attributed to the connectivity of the DRG. Increasing the context length beyond certain limit removes more and more edges from the DRG. However there is a maximum limit to the number of components that can be obtained by increasing the context length. Beyond this threshold limit, once the maximum number of components is attained, this number remains a constant notwithstanding any subsequent increase in the context length. In some benchmarks, a large number of paths to an allocation site tends to generate more components, as in parser and vpr. For other benchmarks, these paths do not guarantee improved contexts unless a very long path is considered. In such programs, we consider call graph profiles as the context for the
Chapter 4. Leakage Power Reduction in Data Caches

Figure 4.7: Relevance on context length in Parser and Perl benchmarks. We can observe that there is an increase in the number of disjoint regions with context length.

malloc sites. For this purpose we experiment with both path and calling context, with different context lengths and we choose the one that generates a higher number of components. In some cases, for example gzip, memory allocation and usage happen in a loop body. So, unless we consider the entire program path, we cannot capture the disjoint regions. One may choose a trial-and-error method for various path lengths or calling contexts, but this does not guarantee reaching the fixed point of Algorithm 1 (for maximum number of meaningful data regions).

A set of Limitation Guidelines are proposed to identify such benchmarks and exit graciously. The aim is to track heap table and disjoint data regions, and exit from Algorithm 1 at appropriate scenarios. One such scenario is: if the size of the heap table, i.e. number of entries, is small and not changing with context length. Another scenario is: if the number of disjoint regions remaining constant at small caller context lengths while increase linearly with path lengths. As gzip falls under this class, we do not consider it for our experiments.
4.3.5 Quantifying Criticality of DDR

As discussed earlier, dynamic data structures that do not impact the overall execution time can be used as candidates for designing power-aware optimizations. Due to complex pipelines, quantifying the criticality or the latency tolerance of these data regions is challenging. We quantify the criticality of the memory operations using Fields criticality model \[86\] as described in the subsection 4.2.1. Further, we use a pipeline simulator \[91\] to obtain the frequency of execution of each memory instruction and the total number of critical instances of the instruction.

We first define the criticality of a data region using equation 4.1.

\[
C_{op} = \frac{T_{Crit_{op}}}{Total} \tag{4.1}
\]

\[
C_{data} = \sum_{\forall \text{op}} C_{op} \tag{4.2}
\]

\(C_{op}\): criticality of operations

\(C_{data}\): criticality of data region

\(T_{Crit_{op}}\): critical instances of operations

\(Total\): total number of instances

During profiling, we gather the total number of times a particular instruction was commit critical, i.e., \(T_{Crit_{op}}\). The section \(Total\) is the total number of times that the instruction was executed. The summation over all the critical operations that access a particular data region is given as \(C_{data}\). The criticality of the data region accessed by these operations is higher with the high value of \(C_{op}\).

Applying equation 4.1, we get criticality data regions across the benchmarks \(vpr, eon\) and \(twolf\) as shown in figure 4.8. Here we compare this weighted average of criticality to footprint and hotness of data regions. The graph depicts data footprint, shown as X-axis along with criticality and frequency of accesses as Y-axes. Each point indicates a data region’s size, criticality and frequency of accesses. Size of the data is measured as number of kilo bytes, criticality by applying equation 4.1 and frequency of accesses as percentage of accesses to this region measured during simulation. As we can observe from figure 4.8, the data regions can be marked as critical or latency tolerant adhering to a particular threshold value. For example the
data regions above 5% of criticality are critical, rest of the regions are latency tolerant. It is to be noted that across these benchmarks more than 80% of the data regions can be classified as non-critical. The threshold value is a design parameter that can be tuned according to the requirements.

Also to be noted is that frequency of accesses, data size and criticality of the data regions do not correlate with each other. Except for the point P1, (8, 22%, 80%), most of the data regions dominantly exhibit either one or two of these characteristics. For example the point P2, (40, 1%, 80%), depicts a data region which is of significant size with large number of accesses. But the criticality is is very low. \textit{Thereby we can say that a hot component or a large size data region is not necessarily a critical region.} Other benchmarks also show a notable partitioning between critical and non-critical data regions. We also observe that the latency-tolerant data regions comprise of 30 to 50% of total memory accesses.

This indicates that in order to design new optimizations, criticality of data regions becomes a vital parameter.

### 4.4 Power Aware Optimization

In order to design an effective power-aware optimization, we first split the L1 cache into a drowsy cache and a normal cache. The drowsy cache has varying latency and its parameters
Chapter 4. Leakage Power Reduction in Data Caches

are given in the Table 4.1.

![Power and Performance Penalty](image)

Figure 4.9: The total power savings compared against a baseline processor with no leakage control. The graph shows total savings in the data cache, both L1 and L2. The penalty due to this optimization is also shown.

We assume an additional bit to annotate each instruction to be either critical or non-critical. Once we identify the critical and non-critical data regions, we use DRG to remap these data regions to memory operations. We now annotate all the memory operations accessing critical data regions as critical. Note that some of these operations might not have been considered critical earlier during profiling. The intuition behind this method is, by using the disjoint nature of these data regions we are minimizing variations in power state of cache lines.

Here we describe our cache policy split data cache with data region policy: Split data cache with Data Region based Policy (SDRP), of handling critical and non-critical data accesses. During instruction fetch, the instruction decoder dynamically identifies it as either critical or non-critical. When critical memory instructions are missed data is fetched into the normal cache. When non-critical instructions are missed, data is fetched into the drowsy cache. Note that latency of the accesses to the drowsy cache is higher than that of the normal cache. This causes considerable delay in the pipeline. But since these accesses emanate from non-critical data regions or latency-tolerant data regions, this delay is absorbed by the pipeline, thereby
minimizing the impact on performance.

An important limitation of our study is that the profile-driven technique that is employed here does not guarantee precision in finding disjoint data regions. In other words, parts of critical regions may not be critical at all and similarly, parts of latency-tolerant regions may be critical. Hence it may be necessary to check both the partitions in the case of a miss. This operation may not guarantee reduction in the power consumption since the per-access power may either remain the same or even slightly higher than the baseline. Addressing this concern we devise a policy for misses.

**Case 1: When there is a miss due to a critical memory operation**

The optimization performs a tag-search in the drowsy cache. If it is found, the rest of memory operation is executed by fetching the data. However, this additional penalty especially for critical memory operations can hamper overall performance. Hence our technique of power optimization simultaneously searches both drowsy cache as well as the L2 cache, causing a significant reduction in access latencies. This mitigates the above problem and minimizes performance penalty. In our power consumption model, we have considered the additional power consumption due to simultaneous searches of both drowsy as well as the L2 cache.

**Case 2: When there is a miss in the drowsy cache due to non-critical memory operation**

In this case, a tag look up in the normal cache is performed. If we encounter a miss, an L2 cache access is performed. In this case, the access latency is higher than the access latency observed in case 1. However, since the access emanates from non critical memory operations, the additional latency is absorbed.

We compare SDRP with a baseline configuration where L1 cache of size 32KB is not split and is a normal cache. A 32KB drowsy cache is employed in Full Drowsy cache Simple Policy (FDSP). This policy was proposed in [45]. In FDSP a complete drowsy cache is turned off periodically. In FDSP we employ a 2000 cycle window. Considering 0.02um technology and 3 cycle latency for bringing a drowsy cache line to active state. As we discussed earlier, we assume a simplified model to validate our technique. In most of the current multi-core architectures, a private L1 cache is employed which caters to the single application. We assume such a cache which is small in size and associativity. We presume a large, scaled up cache architecture can also be employed with SDRP by rerunning the whole methodology.
The other configuration is to control the split cache with instruction criticality based control policy: Split data cache with Instruction Criticality Policy \textit{SICP}. In \textit{SICP}, critical path predictor model as explained in sub-section 4.2.1 is used at run-time. The data fetch policy is similar to our policy, but we use the critical path model to predict whether an \textit{instruction} is critical or not. We fetch and place the data into drowsy and normal modes based on the \textit{instruction} criticality.

### 4.4.1 Experimentation

For experimentation, we consider benchmarks from \textit{SPEC-INT2k} [75] such as \textit{parser}, \textit{perl}, \textit{vpr}, \textit{twolf}, \textit{eon}, pointer intensive benchmarks like \textit{bc} and olden benchmarks: \textit{em3d}, \textit{bh}. It is not possible to experiment with other benchmarks because they have less number of disjoint regions. Moreover due to custom memory allocation, especially \textit{gcc}, it is difficult to instrument the memory allocation points. In spite of a custom memory allocator in \textit{parser}, the allocation happens through a single function which could be instrumented. For our experiments, we modified the \textit{Hotleakage simulator} [91] to add extra cache and measured the total leakage savings using the power model provided in the simulator. \textit{Hotleakage} implements various low leakage controls like drowsy cache, \textit{gatedV\textsubscript{dd}} etc. The L1 data cache is split into two halves: a normal mode cache, and another with a leakage control mode. We consider the drowsy cache as the leakage control cache.

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>O-O-O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>4</td>
</tr>
<tr>
<td>L1 - Split cache</td>
<td>16KB, 16KB, 2 cycles</td>
</tr>
<tr>
<td>L1 variable voltages</td>
<td>0.8v-0.6v</td>
</tr>
<tr>
<td>L1 variable latencies</td>
<td>3-5 cycles</td>
</tr>
<tr>
<td>L2 cache</td>
<td>512KB 4 way, 6 cycles</td>
</tr>
</tbody>
</table>

Table 4.1: Split Data Cache Configuration

Next, we measure the power consumption in two parts. We first measure the total dynamic power consumption. As the L1 data cache is split into two halves, the dynamic power per access changes for each access. The splitting of the cache could alter the behavior of the L2 level data cache, thereby changing the power consumption in the L2 cache. We then measure the total leakage consumption. Usually cache leakage is measured from the tag array and data array parameters. Existing power models like \textit{cacti} [92] compute per-cycle leakage of a cache.
Chapter 4. Leakage Power Reduction in Data Caches

We measure the total leakage in the cache as well as total savings in the drowsy cache. The total power savings are in both forms of power consumption: dynamic as well as leakage power.

4.4.2 Results

The dynamic and leakage power savings compared to a baseline processor for the configuration given in Table 4.1 are shown in the figure 4.9.

The graph depicts that with SDRP an overall power savings ranging from 35% to 38% and a leakage energy savings ranging from 13% to 20% can be achieved. Since L1 cache is split, the per access power consumption is almost halved. By employing drowsy cache, we achieve significant leakage savings. We compare our savings against FDSP and SICP. These savings are comparable to our technique. The FDSP can reduce the leakage significantly as most of the cache lines are placed in drowsy state. But the performance penalty of this policy is significantly higher than our technique. In FDSP we observe an average of 21% savings in the leakage power with 27% of total savings.

In SICP we observe an average of 25% total savings and 15% of leakage savings. This configuration is moderately less energy-efficient than using the FDSP configuration. But due to critical instruction prediction, the performance penalty is lower than that of FDSP, which is 4% compared to FDSP which 6%.

Leakage savings of SDRP are higher and in some cases comparable to FDSP, which is 32KB drowsy cache where in our technique we employ 16KB drowsy cache, but our total savings are significantly higher than both the other configurations. The main reason is very low performance penalty, which is less than 1% even with a split data cache.

Note that as the cache is split, the leakage energy consumption savings are only attributed to the split data cache. The performance penalty (though very insignificant) of SDRP, is due to the increased latency of the drowsy accesses and to some extent, due to increased L2 level accesses. In some cases like parser, there is a slight performance improvement. This is due to a reduction in the number of L2 level accesses and reduction in L1 cache pollution. We do not employ drowsy cache at L2 because it is a shared resource. Also, a minor increase in the access time in L2 can impact the overall performance because the memory access is already on the critical path.
4.4.3 Sensitivity analysis

Figure 4.10: This graph shows the sensitivity of the optimization. The comparison is between a complete L1 drowsy cache and split cache. The power consumption is compared against a baseline processor with no leakage control.

Since critical data regions have a higher impact on the performance as compared to non-critical regions, we perform a sensitivity analysis of the split cache at various voltage levels. As indicated in the experimental results, the split cache contributes to significant power savings. Now, we vary the voltage levels of L1 data cache. The configurations 0.8S, 0.7S, 0.6S indicate the voltage levels of the drowsy cache pertaining to the split L1 cache. Similarly 0.8D, 0.7D, 0.6D indicate the voltage levels of the total L1 drowsy cache. The variation of the voltage levels causes variation in leakage power per cycle and also varies per-access cycle time. A limitation of Cacti [92] is that it can estimate leakage power as well cycle time at a fixed voltage level. Since it does not consider different voltage levels, we assume that the latencies would be 3, 4 and 5 cycles for voltage levels 0.8v, 0.7v and 0.6v respectively.

The results demonstrate that split cache outperforms the total drowsy cache. Even though the per-cycle leakage power in the total drowsy cache L1 is lower than that of the split cache, the extra leakage savings are dominated by an increase in performance penalty. The total number of execution cycles increase due to an increase in the access times. Thus the total power consumption increases with increase in access latencies. Due to increased number of transitions in the total drowsy cache, extra penalty (which is 3 cycles from low to high and 300 cycles from high to low) increases the power consumption. We use the same penalty values for
the split data cache. This is further illustrated in figures 4.10 and 4.11. As shown in figure 4.10, vpr benchmark consumes more power as compared to the baseline processor. Figure 4.11 shows the increase in the penalties for both the caches by varying the supply voltage. As we can observe, the total L1 drowsy cache quickly starts showing significant penalties (as for twolf), whereas the split cache absorbs the extra latencies due to the split drowsy cache and the penalties do not grow at a similar rate. For benchmarks bh, em3d and bc, we do not notice a remarkable difference as compared to the split cache. This is because the data regions that are fetched into the drowsy cache are tolerant or non-critical. We also observe an extra leakage savings of around 5% as compared to the baseline processor when we vary the supply voltage. These savings can be attributed to the reduced cycles or improved performance of split cache.

4.5 Related Work

Traditional compiler algorithms statically identify critical memory operations based on the data dependency graph[70]. These techniques associate predefined latencies with the edges of these dependency graphs. Srinivasan et.al. [93] discussed run-time metrics and other dynamic dependency-based characteristics to identify critical memory operations. The authors demonstrate that these metrics perform well in identifying the critical memory instructions.

As already cited, Fields et.al [86] developed a more rigorous pipeline based critical path
model that not only identifies critical instructions but can also be used to predict the criticality of in-flight instructions. Their model extends the work of Srinivasan et al. [93] by taking into consideration the resource dependencies and classifies instructions as either fetch critical, execution critical or commit critical.

More recently, newer metrics [94] have been developed to improve the notion of criticality. These distinguished studies signify that using critical instruction predictors may lead to an increased likelihood of non-critical instructions being predicted as critical. These metrics are used to identify critical instructions from program structure. Fields model provides an architecture dependent trace based critical path model which can capture dynamic events.

[95] performs a limit-study on critical loads which are not vital for execution time. In [96] the authors introduce a new metric for quantifying the criticality of instructions. This architecture-dependent metric, tautness, is used to design more efficient critical instruction predictors.

In a nutshell, most of the optimizations designed based on the criticality models worked at the instruction level [97][98]. They conclude that a data cache block gets accessed non-uniformly by critical and non-critical instructions from the time it is available until it gets replaced. This causes inefficiency in the energy savings. All this research points to the need to quantify the criticality of a data region from the program point of view instead of using critical instruction predictors to dynamically tune data cache voltage levels. We go further and analyze chunks of memory regions for their latency tolerance as well.

In [99] authors propose smart caches where sizes and associativity of caches can be varied. They provide additional index bits in the tagging structure of the cache to choose number of sets there by varying the associativity of the cache. In [100] authors propose an extension to drowsy caches where additional bits control word level voltage control of cache lines. This can enable finer control of voltage supply at word level. In [101] a phase adaptive drowsy cache is proposed. The authors show that by considering the lifetime of a cache line they either place it in an active cache or a drowsy cache. In [102], the authors propose a level 0 cache, which is small in size and can determine whether a hit or miss going to occur prior to L1 cache. L0 and L1 caches are at similar level. They propose a flow cache where in a cache line is maintained for longer duration by sharing it between L0 and L1. They also propose a hit cache, which filters hits to L1 cache and highly reused cache lines are held in L0. In most of these techniques either
frequency or size of the data plays a crucial role. In our technique we introduce criticality to be the driving factor.

4.6 Conclusion

In this chapter we addressed the problem of energy consumption in data caches of modern embedded systems. We used a profile-based technique focusing on context-as well as path-sensitive analysis to identify disjoint data regions. We analyze the criticality behavior of these data regions at a coarser level than at a single memory operation level.

We used this analysis to drive an energy aware optimization technique. We used a split data cache, operating at normal and drowsy modes, to critical and non-critical data regions respectively. We proposed a cache policy to minimize performance penalty. This technique saves around 35% of total power and 20% of leakage power in the data cache without any significant performance penalty on a subset of SPEC benchmarks.
Chapter 5

Locality Analysis of Disjoint Data Regions

5.1 Introduction

The previous chapter illustrated that on-core power consumption is mainly dominated by data caches. Data caches are accessed very frequently and they occupy a large die area. The technique described in the previous chapter reduces leakage energy in the data caches. In this we leverage the data regions described in the previous chapter and perform locality analysis. As memory requirements of mobile or embedded applications increase, the size and associativity of the data cache also become critical. These design parameters play a crucial role in per-access power consumption and latencies. In this chapter we address the problem of dynamic power in the data caches. We analyze the locality of the data regions and propose a new cache management scheme.

Some of the memory management techniques include self clocking, dynamic and static power management strategies to reduce dynamic power are described in [103]. Figure 5.1 shows the relationship between sizes, associativities, power consumption and access latency of various configurations modelled using CACTI [92] for a 70nm technology. As can be seen from the figure, the power consumption as well as the access latency increase significantly with size and associativity. Therefore it is crucial to employ a smaller and lower associative cache for power savings without compromising on the overall performance of the programs.
5.1.1 Focus of the chapter

Figure 5.2 depicts the focus of the chapter.

The previous chapter elaborated that a disjoint region is a mapping between a set of data addresses and a set of memory operations, where in the memory operations access only the mapped subset of data addresses. We use context sensitive profiling to capture these disjoint data regions. These data regions approximately correspond to heap allocated data structures in a program.

In this chapter we discuss a data region-centric locality analysis technique. We assume a partitioned data cache with different associativity. We apply this locality analysis to the partitioned L1 data cache by allocating low locality data regions to higher associative partition and high locality data regions to lower associative partition. We use a metric called reuse distance [104] to analyze the locality of these disjoint data regions and also the overall locality of the program. Reuse distance of an address is defined as \textit{the number of unique addresses accessed between two accesses to that particular address}. This metric is predominantly used to estimate the temporal locality available in the program.
We introduce a partitioned stack based approach to measure the reuse distance in each of the data regions. The objective of this analysis is to classify the data regions in terms of locality. It can be easily observed that data regions with low locality can harm the performance due to the misses at various levels of the memory hierarchy. If a data region has high locality, its operating footprint would be less, hence cache replacements would be lower. Such a data region can do little harm in terms of both performance and power consumption. There have been multiple proposals on asymmetric cache designs [105], non-uniform caches [106]. Here we assume a hypothetical model with data cache with different associativities. This technique results in 23% of the power savings without any significant loss in performance.

In most of the programs observed, there are more than one data regions with significant total footprint and access frequency, contributing to the overall memory accesses. Therefore, the problem is to identify a subset of these data regions, which have either lower or higher locality.

5.1.2 Methodology

In short our approach can be outlined as follows:
• Using context sensitive profiling, we generate an access stream of data addresses and map each data address to a data region.

• Next, we classify the data regions with respect to the locality. We find a subset of the data regions which when considered together have very high data locality. We use Reuse distance to find such subsets of data regions.

• We then propose a cache partitioning scheme where the high locality data regions are allocated into lower associativity auxiliary caches. This scheme aims to reduce total power consumption in the caches.

5.1.3 Significance of Locality Analysis

Locality is defined in two ways. Temporal locality, which is defined as frequency of successive accesses to a particular data address, and spatial locality, is defined as the frequency of accesses of a particular pattern or sequence of memory addresses. It is well known that every program uses only a part of its data actively during execution [107]. Hence it is of prime importance to measure the active data usage of the program to understand and improve its use of cache memory [107]. This program dependent property directly influences the design parameters of modern processors and therefore has a direct impact on the performance as well as the total power consumption of the programs. In modern processors, a data cache, at different levels of hierarchy, exploits the locality in a program.

Example The example illustrated below explains the significance of locality analysis to design power aware optimizations.

Consider the example shown in Figure 5.3 based on the parser benchmark. The example shows two parts of a data structure: a table, whose size depends on the input set of sentences or strings, and a list of words corresponding to each of the strings. Each entry of the table is a list (linked list) of words carrying different attributes for that word. The main table is indexed by a hash function value. The function parse_string sets certain attributes for each entry. The function parts_of_speech performs a series of computations on the linked lists. For such a data structure, the frequency of accesses to the main table and the lists are more or less comparable. However, the access patterns of the main table and the lists are different.
Therefore, the temporal locality of the entries in the main table would be different from the locality of the nodes in the lists.

So the whole data structure has two types of locality behaviour. A cache allocation policy, with a partitioned data cache can exploit such behaviour and utilize the data cache efficiently by associating a smaller cache for the main table and a larger cache for the linked lists. A data centric locality analysis can therefore be effective for designing power aware optimizations.

5.1.4 Organization

Section 5.2 begins with a brief introduction of the traditional approach of measuring reuse distance between two data regions. We modify this measure and extend it to measure the compound reuse distance between disjoint data regions. Here we propose an \textit{apriori-based} selection scheme to find the subsets of data regions that require low associativity. We further venture to extend this traditional monolithic stack approach and devise a partitioned stack based reuse distance algorithm. This mitigates the complexities of compound reuse distance measurement. In section 5.3, we propose a cache allocation optimization based on reuse distance. The subset
of data regions which have a high percentage of accesses below a certain value (called threshold value), are allocated to a lower associative cache. We can apply this algorithm at any level of cache hierarchy. In section 5.4 we conclude.

5.2 Measurement of Reuse Distance

5.2.1 Defining Reuse Distance

Reuse distance analysis characterizes application memory access behaviour without attaching itself to a specific memory hierarchy[108]. It refers to the number of distinct data elements that have been referenced since the last access of an element. Reuse distance yields a generalized machine independent measure of program locality for predicting cache performance and providing cache hints during code generation. It can also be used to improve temporal locality by generating models that can restructure code and data structures [104][109][110][111][112][113][114][115].

Reuse distance of an address is defined as the number of unique addresses accessed between two accesses to that particular address. For example, in the access stream a, b, c, b, c, a, the reuse distance of a is 2, as there are only two unique addresses b and c between two accesses to a.

5.2.2 Traditional Reuse Distance Measurement

Traditionally, reuse distance has been measured using a polynomial-time algorithm with respect to a monolithic stack. This is shown in algorithm 4. All the existing algorithms work on a single stack built dynamically from the memory access stream. Each new access to a particular address causes a lookup on the stack, which is built dynamically. If the address exists in the stack, the distance from the top of the stack is measured. Then the address is removed from the stack and pushed on the top of the stack. The depth of the stack where the address was removed from, gives the reuse distance of the address at that instance.

5.2.3 Measurement of Compound Reuse Distance

The traditional reuse distance algorithm shown in Algorithm 4 can be extended to capture the compound reuse distance of the data regions. The compound reuse distance of a data region can be defined as the reuse distance measured for a combination of data regions. We also define relative reuse distance as reuse distance of a data region with respect to the other data regions
Chapter 5. Locality Analysis of Disjoint Data Regions

Algorithm 4 Traditional Reuse Distance Measurement

\begin{itemize}
  \item \textit{Input} is the Access sequence
  \item $A$ is a Set of addresses, initialized to empty
  \item $S$ is a Stack initialized to empty
  \item $d_i$ is the reuse distance measured from the top of address $i$
  \item $a$ is an address in from \textit{Input}

  \textbf{while} $\neq$ EOF(Input) \textbf{do}
  \begin{itemize}
    \item read(a, Input)
      \begin{itemize}
        \item if $a \in A$ \textbf{then}
          \begin{itemize}
            \item starting from the top of S;
            \item if $a$ is found \textbf{then}
              \begin{itemize}
                \item $d_a =$ offset of the location of $a$ from the top of S
                \item remove($a$)
                \item rearrange $S$ to cover the hole left by $a$
              \end{itemize}
          \end{itemize}
        \item else
          \begin{itemize}
            \item $A = A \cup a$
            \item $d_a = -1$
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \item push($S,a$)
  \end{itemize}
\textbf{end while}
\end{itemize}

of the program. In this extended approach, a vector of counters for each address in a data region is maintained to capture the accesses to other data regions. Each counter in the vector is associated with a data region. A mapping between the addresses and data region also needs to be maintained.

5.2.4 Working of the Algorithm

When an address $a$ is accessed, during the stack lookup, the data region corresponding to that address is acquired using the mapping. The stack lookup encounters all the addresses that were accessed after the previous access to address $a$. The counter corresponding to the data region $M$, to which the address belongs to, is incremented for each of these addresses. Every time an access incurs a lookup, apart from the reuse distance, we can also capture the relative reuse distances with respect to other data regions. The relative reuse distance values at any instance of the access, for each of the addresses are available in the counters. The compound reuse distance of any combination of data regions can be computed by adding up the corresponding counter values.
Algorithm 5 Modified Reuse Distance Measurement

Input is the Access sequence

$A_i$ is Set of addresses belonging to data region $i$

$S$ is the Stack initialized to empty

$s_j[i]$ is vector corresponding to number of accesses to data region $i$ for address $j$

$d_i$ is distance from the top of element $i$

$a$ is the address

$M = map < A, a >$ indicates data region of $a$

while $\neq$ EOF(Input) do
read (a, Input)

if $a \in A_M$ then

   Starting from the top of $S$
   $d_a = d_a + 1$
   $s_{current\text{element}}[M] = s[M] + 1$  \(\triangleright current\text{element}$ indicates the element encountered during the search

   if a is found then
      remove(a)
      rearrange $S$
   end if

else

   $A = A \cup a$
   $d_a = -1$
   $s_a[\forall M]=-1$
end if
push($S, a$)

end while
This approach additionally requires a vector of counters for each address on the stack. Moreover when the compound reuse distance of a subset is computed, we need to build the stack for that subset and compute the compound reuse distance at the end of the lookup for that particular subset. In the algorithm 5, each stack corresponds to a particular combination of data regions. In order to compute compound reuse distance of all the combination of data regions, we need to execute this algorithm as many as the number of combination times. This increases the time complexity. As an alternative if we maintain a single stack for all the subsets, it results in the inclusion of the addresses from other data regions in the stack. This will increase the lookup time for each of the data addresses in the subset that we are interested in. An example depicting this approach is shown in figure. In order to compute compound reuse distance, the necessary components of the relative reuse distance can be used. In the example, at the end of second a2, the relative reuse distance of data region A, with respect to B is 2 and with respect to C is 1. The compound reuse distance at this access for AB combination is 3, as it can be calculated as relative reuse of a2 with respect to be B as 2 and local reuse distance is 1.
Measuring partitioned stack based reuse distance

In this chapter, we propose a partitioned stack based reuse distance measurement technique. In this approach we maintain stacks for individual data regions and compute the compound reuse distance. For example, if there are three data regions: A, B and C, we maintain three stacks corresponding to each of these data regions. By doing so, we can eliminate unnecessary lookups. Further, we can compute compound reuse distance of any subset without building stacks for each of the subsets.

Creation of markers

An additional data structure that keeps track of the beginning and end of accesses to a particular data region in the form of markers is maintained. These markers record the relative reuse distance. A hash function is used to map the address to a particular stack. Whenever there is an access to a stack succeeded by a transition to a new stack, we create a marker on top of the previously accessed stack. The creation of markers is shown in figure 5.5. An edge is created to the recently created marker from the markers on the top of the other stacks. The counters which are on top of the other stacks are incremented with the unique memory addresses seen in the last stack. Figure 5.6 illustrates the algorithm to create markers at the global level.

An Example

In the example shown in figure 5.5, the ordering of the accesses to data regions A and B are depicted using edges W, X, Y and Z. W indicates accesses to data region A, before any access to B. Therefore we create this edge from base of B to A. Similarly X indicates a transition from A to B, i.e., there are accesses to B following A. Subsequently, N unique addresses are accessed in A and a marker is created on top of A before transitioning to B. The edge Y points to this marker. Y and W point to two markers separated by N unique addresses. These N addresses are added to the counter of the corresponding marker in the stack B as shown in the figure. Finally during transition from B to A, Z is created. At this stage, M unique addresses witnessed in B are entered in the counter of the corresponding marker in A as shown in the figure.
Computation of global and local reuse distance

Figure 5.6 illustrates the algorithm for computation of global and local reuse distance after the creation of markers as discussed above.

The routine ComputeReuseDistance computes the relative reuse distance. While searching for an address in a stack, we acquire the relative reuse distance with respect to other stacks. Before the address is searched in the stack corresponding to the data region,

LocalRD computes the local reuse distance of the address in the stack. This is the reuse distance of the address when considering all the addresses of a the data region to which the address belongs. This is the distance of the address from top of the stack.

The global reuse distance GlobalRD is computed by the getGlobal routine. This is the relative reuse distance of the address with respect to addresses belonging to other data regions.

While searching for an address, we encounter markers in the stack.

Before we search for the address in the corresponding stack, all the stacks have a marker on top. The top marker on the last visited stack will have the unique memory addresses pushed in between two visits of the stack. CreateMarkerEdge creates an edge between the markers. The edge is from all the stacks and last visited stack. We use these edges to compute unique
addresses corresponding to other stacks. As mentioned earlier, a unique memory address count is maintained in these counters. In order to maintain the count, we decrement the counter value of the markers that correspond to edges from the last marker that was encountered. This is carried out by updateOthers routine. The total reuse distance of the address is LocalRd + GlobalRD. When computing GlobalRD we can add values corresponding to the data regions that we are interested in. For compound reuse distance, we add only the values corresponding to the data regions of the subset we are interested in.

An Example

Computation of global reuse distance using partitioned stack approach

In figure 5.7 the access stream is given by $a_1, a_2, b_1, b_2, a_3, b_3, b_2, a_1$. The growth of the stacks is shown in figure 5.7. When there is a transition from a stack say, A to B, we create a marker shown as $m_1$. A pointer from the base of the stack B, i.e., $m_{B_0}$ to this marker is created. After stack B grows, when there is a transition from B to A, a new marker $m_2$ is created. This marker has an incoming edge from $m_1$ created after accessing $a_2$. When the access stream
Figure 5.7: An example explaining marker based algorithm to compute Global reuse distance reaches $b_2$, which was accessed previously, it is moved to the top of the stack recording its reuse distance. Subsequently, the counter in marker $m_1$ is decremented. This is carried out by acquiring the marker $m_{B_0}$ (which is the marker preceding $b_2$), and $m_1$ (the marker pointed in A). In $m_1$, we decrement the counter. Later when $a_1$ is accessed the relative reuse distance of $a_1$ to B is computed by adding the counter values corresponding to B, in all the markers seen while searching for $a_1$. The relative reuse distance of $a_1$ to B is 3 and local reuse distance is 2. Therefore the total reuse distance of $a_1$ is 5. In short, we increment or decrement the counters to capture the relative reuse distance of any element in a stack with respect to other stacks.

Figure 5.7 also shows the counter values of all the markers recorded at each step. As the counters are specific to each stack, we can compute relative reuse distance of an element with respect to any subset of other stacks. This gives us an advantage in computing reuse distance corresponding to different combinations of the stacks.

**Computing compound reuse distance using partitioned stack approach:**

Consider the example shown in figure 5.8, where the access stream from data regions $A, B, C, D$ is as shown below. The different subsets of the access stream $\{A,B,C\}$, $\{A,B\}$ and $\{A,C\}$ are also shown.
Each partition of the stack is mapped to a disjoint data region. As already indicated in the beginning of the section, we consider three data regions: A, B and C. We maintain three stacks corresponding to each of these data regions and we compute the compound reuse distances of AB and AC. Figure 5.8 shows the snapshot of the stacks after $d_2$ in the stream. As seen in the figure, the markers $m_2$ and $m_3$ are pointed from $m_1$ indicating the order in which each data region is accessed. The values in $m_1$ corresponding to A, B and C are 0, 1, 2 representing $b_1$ of data region B and $c_1$, $c_2$ of data region C.

After $b_1$ is accessed the counter corresponding to A in $m_2$ is 0. Thus the reuse distance for AB is 0. When $a_1$ gets accessed, the counter in $m_1$ is 0, 1 and 2. Hence the reuse distance for AB is 1 and AC is 2. The table in figure 5.8 indicates the values of the counters in the markers. The reuse distance values are also shown in figure. NA indicates an instance when a data address not mapping to the subset gets accessed. Following this approach, we can compute the compound reuse distance without building the stacks for all the subsets and computing the reuse distance separately for each of the subsets.

**Evaluating partitioned stack algorithm**

One may argue that the time complexity of the partitioned stack algorithm is similar to the modified traditional algorithm. This stems from the fact that we need to add the values in the counters in each of the markers in the partition based algorithm.

To address this question, we present the number of markers that are created during the partitioned algorithm. The number of markers created depends on the locality in each data region and also across data regions. If the same data region gets repeatedly accessed in succession, the number of markers created would be less. Figure 5.9(b) indicates the amount of memory saved using partition based algorithm as against modified traditional algorithm. The high memory savings in some of the benchmarks is due to the high frequency of accesses with low reuse distance as shown in figure 5.9(a). The normalized graph shows the distribution of
reuse distance with respect to the percentage of frequency of the accesses. The reuse distance increases vertically, the dark bar indicating low reuse distance. The other benchmarks which do not show any savings are due to the low locality of the disjoint data regions. Though there is no apparent memory saving for other benchmarks, there is a scope for improvement in managing the markers. We also observe that significant markers are created due to small percentage of data regions (either 2 or 3). This indicates that there is a frequent switching among these data regions.
5.3 Cache Allocation Optimization for Power Savings

In this section, we propose a cache allocation optimization based on reuse distance. The subset of data regions incorporating higher percentage of accesses below a threshold value of reuse distance are allocated to a lower associative cache. The goal is to find the right combination of data regions which need a lower associative cache.

Our approach can be summarized in three steps:

- First, we establish a proper threshold value both from reuse distance point of view and accesses threshold point of view. Consider an L1 cache partitioned into two; one with a higher associativity, say 4 and another partition with lower associativity 2. Assigning a threshold value to the optimization forms an important part, as it can impact both the cache misses and also the total number of accesses. The reuse distance below a certain value can vary among the benchmarks. It is also important to consider the percentage of accesses below the threshold value. For example, if we encounter 80% of the accesses with a conservative reuse distance below 2, it results in no subset selection from the apriori. On the other hand, if we encounter around 40% of accesses with a relatively higher reuse distance of 6, it may have a higher impact on the total number of misses. Hence it is very important for a designer or a programmer to set a proper threshold both from reuse distance point of view as well as assesses point of view to achieve maximum benefits. In order to achieve low power savings we need to find a combination of data regions whose reuse distance will indicate lower associativity requirement.
• Secondly, we partition the cache using this threshold.

• Finally, we assign the required data regions to lower and higher associative caches, depending on the reuse distance and their respective footprint.

**Selecting the right partition for disjoint data regions**

**Algorithm 6** Apriori-based subset selection

\[
\begin{align*}
N &= \text{Total Number of Disjoint Regions} \\
C_k &= \text{Candidate subsets of cardinality } k \\
P_k &= \text{Pruned subsets of cardinality } k
\end{align*}
\]

\[
\text{for } k = 1 \text{ to } N \text{ do} \\
\quad \text{while } C_k \neq \emptyset \text{ do} \\
\quad \quad \text{pick } s \in C_k \\
\quad \quad \text{remove } s \text{ from } C_k \\
\quad \quad \text{RdVector} = \text{ComputeRd}(s) \\
\quad \quad \text{if } \text{RdVector} \leq \text{Threshold} \text{ then} \\
\quad \quad \quad P_{k+1} = P_{k+1} \cup s \\
\quad \quad \text{end if} \\
\quad \text{end while} \\
\quad C_{k+1} = P_{k+1} \\
\text{end for}
\]

In order to find the right partition, the algorithm needs an exponential space, considering all the possible subsets. If the total number of sets are \(M\), then it requires, \(M \cdot C_1 + M \cdot C_2 + M \cdot C_3 + \ldots + M \cdot C_M\) amount of space. The reuse distance of each element carries a property; hence we can eliminate the lower cardinal subsets before we examine the higher ones. We propose, an apriori algorithm (Refer Algorithm 5.3) for elimination and selection of the right partition at each level of cardinality.

\(C_k\) indicates subsets of data regions of cardinality \(k\). The maximum cardinality of the subset is the total number of data regions, i.e. \(S\). \textit{ComputeRd} procedure computes compound reuse distance for the subset \(s\) and returns a reuse distance vector \textit{RdVector}. \textit{RdVector} indicates reuse distance vector, a histogram to quantify reuse distance. \textit{RdVector}[i] indicates number of accesses with reuse distant \(i\). All the selected subsets are collected by \(P_k\), where \(k\) is the size.
Chapter 5. Locality Analysis of Disjoint Data Regions

of subset. *Step 0:*

We first compute reuse distances of each of the disjoint regions using sample access streams. Then we select candidates that need either higher associativity or lower associativity depending on the cache architecture under consideration. In order to find a subset of elements that require lower associativity, we establish a predetermined threshold based on some preliminary analysis as explained earlier. The threshold could either be computed with expected power savings (which in turn is computed based on a profiled data) or the total number of memory accesses. A designer or an architect can also define the threshold value. The designer can iterate over multiple values of threshold in order to find the right threshold values. Let us assume the threshold to be $T$. Suppose the threshold reuse distance is $20\%$. This means for any vector at $M^{C_1}$ level, number of accesses which have reuse distance greater than 2 constitutes $20\%$ of the total accesses. In *Step 0*, we eliminate all single data regions whose reuse distance vector indicates that this particular region cannot be optimally allocated into a low associative cache.

*Step 1:*

After selecting the candidates from the *step 0*, we proceed to form subsets of size 2 using $M^{C_2}$ reuse distance. During the reuse distance computation, for each $M^{C_2}$, we again maintain a threshold.

During the computation, if any of the accesses at $M^{C_2}$ level crosses the threshold limit, we eliminate the corresponding vectors at $M^{C_2}$ level and also the corresponding levels at $M^{C_3}$, $M^{C_4}$,... etc.. That is, if at $M^{C_2}$ level, the sets S3, S4 cross the threshold, we eliminate all the combinations with S3 and S4.

After the elimination at $M^{C_2}$ level is done, we proceed to $M^{C_3}$ and other subsets.

*Step 2:*

The algorithm terminates by finding the subset whose requirements fall below associativity of the cache and we cannot proceed to higher levels. If the threshold value is too high, we may not get any subset. In that case, we reduce the threshold or devise new ways of estimating the threshold.

*Setting the threshold* The threshold function depends on the optimization we consider. It also depends on the selection criteria at different levels. For example at level 1, we need to eliminate data regions that have insignificant number of accesses as compared to other data
regions. Thereby we can reduce the iterations of the algorithm and also the space complexity in the reuse distance measurement. The \textit{apriori} algorithm can terminate at any level and it is guaranteed to terminate due to both the size of the number of subsets and also the threshold condition. After considering various types of thresholds, we establish a threshold based on the total fraction of accesses of a data region at the first level. Then from the next level, we use reuse distance based threshold. We implement this algorithm using bitvectors. Most of the benchmarks have less than 20 data regions. Few others which have more than 20 data regions are reduced using frequency threshold.

The above approach can be illustrated with the help of the following example.

\textbf{Example}

<table>
<thead>
<tr>
<th>Case</th>
<th>Access Stream</th>
<th>Reuse Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Reuse Distance</td>
<td>$a_1, b_1, c_1, c_2, d_1, d_2, b_1, a_1, c_1, c_2, d_2, d_1, a_1, b_1, \ldots$</td>
<td>$-1,-1,-1,-1,-1,1,4,5,5,4,4,6$</td>
</tr>
<tr>
<td>Eliminating C and D</td>
<td>$a_1, b_1, a_1, a_1, b_1, \ldots$</td>
<td>$-1,-1,0,1,0,1\ldots$</td>
</tr>
<tr>
<td>Eliminating A and B</td>
<td>$c_1, c_2, d_1, d_3, c_1, c_2, d_2, d_1,$</td>
<td>$-1,-1,-1,3,3,-1,4$</td>
</tr>
</tbody>
</table>

In this example $a_i$ belongs to a particular data region $A$, $b_i$ belongs to $B$, $c_i$ belongs to $C$, and $d_i$ belongs to $D$.

It can be observed that in the second stream, where we eliminated $C$ and $D$, the reuse distances are very less compared to the last access stream where we eliminated $A$ and $B$. Hence we can form two subsets, $\{A, B\}$ and $\{C, D\}$. It has been observed that associativity of a cache and size of the cache play an important role in the total power consumption of the program. Higher the associativity, higher is the access power. The same is applied to the size of the cache as well.

Our goal is to efficiently utilize the cache resources so that we get power savings without compromising on the performance. The first subset $\{A, B\}$ indicates higher locality regions and therefore can use a cache with lower associativity. On the other hand, we can use a cache with higher associativity for $\{C, D\}$. This results in considerable reduction in the total power consumption in the caches.

In this way, we can compute the reuse distance of all the combinations of $\{A, B, C, D\}$. By exploring all the combinations of these data regions, we can find the right subset which requires lower associativity. For this purpose we use compound reuse distance which is defined as reuse...
distance computed by considering each of the subsets of this set \( \{A,B,C,D\} \).

### 5.3.1 Experimental Setup

The profiling and cache modelling are carried out using PIN instrumentation tool [116]. First, we instrument the program to collect reuse distances of a given subset of disjoint data regions. These reuse distances are then used by \textit{apriori} algorithm to select the subsets that have lower as well as higher reuse distances. Then we use an L1 cache model in PIN to measure the total number of misses and compute total power consumption. We use Cacti model [92] to measure power per access and compute the total power consumption in both L1 and L2. The additional accesses which are needed to handle the misses in the L1 cache are also taken into account. The total power consumption in both the single cache as well as the partitioned stack is measured according to the equations below.

\[
PAE = Per - AccessPower
\]

\[
L_{1\text{power}} = PAE \times \text{No. of accesses to L1}
\]

\[
L_{1\text{small}_\text{power}} = \text{No. of accesses to L1 smaller partition} \times PAE
\]

\[
L_{1\text{large}_\text{power}} = \text{No. of accesses to L1 large partition} \times PAE
\]

\[
L_{2\text{power}} = \text{No. of accesses to L2} \times PAE
\]

\[
TotalPower_{\text{Single}} = L_{1\text{power}} + L_{2\text{power}}
\]

\[
TotalPower_{\text{Partitioned}} = L_{1\text{small}_\text{power}} + L_{1\text{large}_\text{power}} + L_{2\text{power}}
\]

\[
\text{Power savings} = \frac{TotalPower_{\text{Single}} - TotalPower_{\text{Partitioned}}}{TotalPower_{\text{Single}}}
\]

### 5.3.2 Analysis

**Properties of disjoint data regions**

We present the properties of disjoint data regions in the form of graphs as displayed below.

Figure 5.10 shows the normalized distribution of the disjoint data regions. Each color indicates a disjoint heap allocated data region. The corresponding cache misses due to these
Figure 5.10: Distribution of accesses to each disjoint data region. Y-axis indicates percentage of access to a particular data region.

Figure 5.11: Distribution of misses of each disjoint data region on a 32KB,4 way cache. Y-axis indicates percentage of misses in that particular data region.

disjoint regions are shown in figure 5.11. The amount of heap accesses along with the percentage of cache misses in them are shown in figure 5.12 and figure 5.13 respectively. The corresponding distribution of footprint along with the percentage of heap data footprint are displayed in figure 5.14 and 5.15 respectively. As it is evident from the graphs, the benchmarks exhibit varied behavior.

As it can be observed, benchmarks like vpr, parser and twolf show correlation with respect to both the accesses, footprint and misses. This means, data regions which have high footprint have high percentage accesses and also high percentage of misses. On the other hand, benchmarks like bzip, perl and ammp show inconsistency across these results. The number of accesses, footprint and misses of the heap data regions vary across the benchmarks. Thus it is particularly difficult to assign a single cache configuration for all the benchmarks. We design a partitioned cache configuration for each of the programs using apriori based technique.
5.3.3 Cache partitioning

**Categorization of disjoint data regions:**

The disjoint data regions can be categorized in three ways. The *apriori* based selection, picks regions which require low associativity. The footprint and accesses of these subsets indicate that the data regions can be classified as follows:

- Those having higher footprint and low associativity requirements. For e.g. benchmarks like *bzip, vpr, wolf, and ammp*.
- Regions with smaller footprint which require lower associativity. For e.g. *perl and art*.
- Regions with smaller footprint requiring higher associativity or large size. For e.g. *parser, equake and em3d*. 
Chapter 5. Locality Analysis of Disjoint Data Regions

Figure 5.14: Distribution of footprint or total size of each disjoint data region

Figure 5.15: Percentage of heap data footprint. Y-axis indicates percentage of heap data available in the benchmark when compared to total memory footprint.

- Regions with higher footprint requiring higher associativity. We do not consider this category in our study since it is not possible to apply cache partitioning to achieve optimization for these benchmarks.

Finding the right partition for each of these categories is decided on the footprint, i.e. the size of the data regions and number of accesses to the data regions selected. The overall goal of this chapter is to find these right partitions. Initially, all the data regions are assumed to be needing high associative cache. The aprior-based scheme selects the data region subsets by performing reuse distance measurement. Finally, once we partition the data into two categories as: low associative requirement data regions whose reuse distance is approximately less than the considered threshold and high associative requirement data regions.

The data cache can be partitioned in multiple ways as shown in the table 5.1. The architect or designer can use the above data to find the which partitions can be considered best for the
Chapter 5. Locality Analysis of Disjoint Data Regions

Figure 5.16: The access latency and power consumption of different smaller caches and the baseline L1 cache Type 1, baseline
L1 cache Type 2
L1 cache Type 3
L1 cache Type 3
L2 cache
32KB 4 way
32KB Direct Mapped and 16KB 2 way
32KB Direct Mapped and 8KB 4 way
16KB Direct Mapped and 8KB 4 way
256KB 8 way associative

Table 5.1: Split Data Cache Configuration

program.

The above categories provide us a heuristic to choose the right partition. Once we select a category, we partition the cache by exploring various combinations of size and associativity. This is done by exploring the limits provided by total per-access power consumption of the baseline cache using Cacti tool. The disjoint partitions ensure that there are no duplicate accesses to both the partitions.

Figure 5.16 shows the access latencies and power consumption per access of different cache configurations that we consider for partitioning. The graph shows that latency as well as power consumption increase with size and associativity.

We partition the cache as shown in Table 5.1.

The total power consumption of these partitioned cache configurations is shown in figure 5.17. As it can be seen, there is drop of 50% of total power per access for second and third configurations. These configurations approximately partition the total number of blocks in the baseline data cache.

The total power savings are shown in figure 5.18. The power savings are due to reduction
Chapter 5. Locality Analysis of Disjoint Data Regions

As shown in figures 5.17 and figure 5.18, we use three different configurations. The first and the second configurations use 32 KB direct mapped cache with 16 KB 2 way and 8KB 4 way respectively. The third configuration introduces 16 KB direct mapped cache with 8 KB 4 way. Applying the equations for power savings as described in the earlier section, we estimate a total power saving of 15% in the total data cache (L1+L2) subsystem.

It is important to note that each of the benchmarks respond differently to each of the configurations, thus we cannot uniformly partition the data cache. The first category *bzip, vpr, twolf* and *ammp* respond well for the first and second configurations indicating that they need a bigger cache and associativity is not much of a concern. The second category responds well...
with first and third configuration indicating that they require lower associativity and lower size. The benchmark *em3d* responds well for the second configuration though it falls in the third category.

### 5.4 Related Work

A plethora of literature elaborates the measurement of reuse distance and its application to measure spatial as well as temporal locality of whole programs, predicting locality and cache optimizations. To begin with, [108] describes a stack based LRU distance to represent reuse distance. Further, as presented in [117], a marker based algorithm optimizes the lookup time. This algorithm reduces the lookup time on the stack by jumping through a set of markers, rather than single elements, thereby reducing the number of elements seen on the stack. Although the earlier proposed algorithms for the measurement of reuse distance were simple, they consumed significant amount of memory space and hence were impractical for runtime measurements. The fastest proposed algorithm used a binary tree to improve the search and the insert [118]. A splay tree was used to maintain the stack. The search involves, searching the tree which is logarithmic. The deletion and insert operations also are logarithmic due to the readjustment of the splay tree. All of these algorithms were implemented on a single stack, which differs from the stack model employed in our study.

Several approaches of measuring spatial locality through reuse distance have been discussed in [119][120]. These techniques apply reuse distance at various levels of hierarchy to get a substring of addresses that are hot. Partitioned stack based reuse distance measurement also can be used to measure spatial locality across disjoint data regions than individual addresses. It is proven that finding spatial locality of a program is a NP-Complete problem [121]. More recently reuse distance has been applied to multi-threaded programs running on multicore processors [122][123].

Following the research on reuse distance measurement, there had been proposals on finding reuse distance of each path as explained in [124]. The paper characterizes paths with respect to reuse distance and uses the property to design new optimizations. Steve Carr et.al. [125] extended that reuse distance can also measured on per-instruction basis by identifying critical memory operations that incur high miss-rates. [126] introduced data structure optimizations
like array grouping and structure splitting which also use reuse distance measurement at the core.

[127][110] suggested the application of reuse distance as a measure to optimize cache allocation. In [127], cache conflicts are captured using offline analysis. This analysis is used to direct the memory accesses so that these conflicts are reduced thereby improving the miss-rate. In [110], reuse distance information is used to predict misses and generate hints in the instructions. These hints are used for both instruction scheduling and data retention. The reuse distance based hints provide more accurate latencies which are incurred by a set of instructions. The cache hints help in retaining data at a particular cache level, improving the miss-rate. Most of the reuse distance based optimizations are designed to predict misses. But the reuse distance measurement in the hardware is very expensive and hence the proposals use a representative reuse distance measured by the total number of misses or accesses [109]. Cache replacement policies are configured using the reuse distance in [128]. In this chapter the cache decay is estimated using the reuse distance. Cache decay is characterized as a property wherein the cache line accesses fade off or reduce even when the cache line is still available in the cache.

5.5 Conclusions

In this chapter, we discussed the locality analysis of disjoint data regions. We propose a new algorithm for measuring reuse distance using partitioned stack based approach. Using this algorithm, we measure reuse distance of subsets of data regions efficiently. These subsets of data regions can be classified in terms of their reuse distances and thus can be allocated into a partitioned cache with different associativities. Due to this technique, we achieve an average of 23% power savings in the best cache without any significant performance loss.

The exploration of cache partitions needs special focus on the total area increase as well. Here we have not explored that parameter. But as the cache sizes increase due to the process technology, the area, per-access power consumption and the leakage play an important role. We believe that finding the right partitioning approach either dynamically or statically, by taking into account disjoint data regions and compound reuse distances as well as total area, could be an interesting problem that can extend this work.
Chapter 6

Conclusions

Excessive energy consumption in computing systems hinders their performance and can even cause reliability problems. Apart from designing new low-power computing systems, it is crucial to analyze behavior of the programs, which are becoming increasingly complex both in terms of memory as well as computation. In this thesis we proposed three different compiler-driven solutions to mitigate the problem of power consumption. We have shown that some of these solutions can be implemented at compilation time. Some of the solutions can be done during profile driven optimization phases of the application development. The advantages of these techniques are to leverage existing low complexity hardware models and seamlessly integrate into development phase of software applications. By applying existing power models we have shown by extensive experiments that these techniques guarantee significant leakage power savings with minimal performance impact.

6.1 Contributions

In this thesis we analyzed various properties of programs, i.e., criticality and locality of data regions, and resource usage of instructions. Using our analysis, we applied various architectural optimizations as well as compiler driven optimizations in order to reduce total energy consumption in the processor as well as in the cache subsystem.

The contributions of the thesis are outlined below.

1. We presented a scheduling algorithm based on automaton based resource model to take
Chapter 6. Conclusions

into account the activities of functional units and schedule the instructions so that functional units are idle continuously for longer durations. We observed that leakage power in the functional units is dominating in future microprocessors. We analyzed the idle characteristics of the functional units during the execution of the programs. We measured the power consumption during idle periods compared against active periods. We observed that if a functional unit is placed in a low leakage mode continuously for longer durations as opposed to short bursts, then the overall power consumption can be reduced. We applied this scheduling technique on embedded programs and achieved 25% savings on an average in the functional unit power consumption with less than 1% performance penalty. Transition aware scheduling can be implemented in any production compiler as it extends list scheduling. This technique fills gap in terms of resource utilization improvement algorithms which primarily dealt as data flow analysis problems.

2. In order to analyze data usage characteristics of programs, we proposed a new profile driven technique to capture disjoint data regions, which predominantly map to the data structures in a program. We used instrumentation tools like ATOM and PIN to gather profile information. In this technique we use context, path as well as hybrid sensitive information to capture disjoint data regions.

3. After achieving disjoint data regions through profiling, we proposed a new technique to reduce leakage power consumption in data caches. We quantified the criticality of the accesses to the data regions. We observed that there is a variation and skew in the criticality of these data regions. We capture non-critical or latency tolerant data regions in the programs. We propose a new partitioned cache architecture which has both drowsy as well as active caches. The non-critical data region accesses are then mapped to the drowsy cache. The overall power consumption is reduced to around 25% with less than 1% performance penalty. The idea of critical data regions is unique in terms of existing literature. This technique is realistic in terms of implementation and also binds the heuristics with cache policies.

4. Finally, we analyzed the data locality characteristics of these data regions. We proposed a new algorithm to compute reuse distance, which can be used to measure locality, by independently measuring reuse distance for each data region and effectively measuring
Chapter 6. Conclusions

compound reuse distance. We applied apriori-based algorithm to capture compound reuse distance. Further, similar to earlier proposed cache architectures, we partitioned the data cache into two parts- one with higher size and associativity and the other with lower size and associativity. The lower size and associativity cache has low power accesses compared to the other partition. This technique achieves reduction in the overall power consumption by 40% with less than 2% performance penalty.

### 6.2 Future Directions

The above described contributions can be used as basis for further investigations in improving power efficiency and paving a way for green computing.

- The scheduling algorithm proposed for leakage can be extended to other components in a processor, disks and memory. As in this technique, if the information on the usage of various components can be modeled, through static analysis or profiling, the scheduling algorithm can be used to increase idle periods there by improving power efficiency.

- A more promising problem is to analyze critical of types of data structures. If we can use analysis of data structures for different types of data structures like linked lists, vectors or queues either through shape analysis or profiling, we can implement new optimizations that can improve the performance as well as power consumption of the programs.

- We experimentally demonstrated that critical memory operations can be found through dynamically building data dependency graph and reducing it into *memory dependency graph*. In a data dependency graph, if we eliminate instructions other than memory operations and place edges between these operations indicating the flow of the data, we get memory dependency graph. A hardware model to identify critical memory operations can be expensive. So, if we use an architecture independent metric like reuse distance as weights of the edges and compute the critical path, we get critical memory operations. Our studies show an overlap of around 70% critical memory operations which hardware based model. The remaining instructions are branch dependent operations, which we can safely ignore if a new memory based optimization is designed. Such machine independent characteristics could be used to quantify criticality of memory accesses so that optimizations can be applied providing efficient software for wide range of devices.
• Apriori-based reuse distance measurement can be used as a basis for measuring reuse distance in multi-threaded environment. The partitioned stack algorithm can be directly applied in these environments and by improving the performance of the algorithm, this can be used in high performance systems.
Bibliography


international symposium on Low power electronics and design (ISLPED), pages 90–95, 2000.


grated hardware-software optimizations using simpower. In *International Symposium on


[68] V.Bala and N.Rubin. Efficient instruction scheduling using finite state automata. In

[69] T.A.Proebsting and C.W.Fraser. Detecting pipeline structural hazards quickly. In *Sym-

[70] Stephen Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann,
1998.

for the intel XScale core. In *Conference on Language, Compiler and Tool Support for

[72] Intel Corporation. Intel XScale microarchitecture for the PXA255 processor: User’s
manual.

Mibench: A free, commercially representative embedded benchmark suite. In *IEEE

and synthesizing multimedia and communications systems. In *Thirteenth International


[76] AJ Kleinosowski and D.J. Lilja. MinneSPEC: A new SPEC benchmark workload for


