

Copyright
by
Vinod Viswanath
2013

The Dissertation Committee for Vinod Viswanath
certifies that this is the approved version of the following dissertation:

Correct Low Power Design Transformations for Hardware Systems

Committee:

Jacob A. Abraham, Supervisor

Adnan Aziz

Sarfraz Khurshid

Lizy John

Rishiyur S. Nikhil

Correct Low Power Design Transformations for Hardware Systems

by

Vinod Viswanath, B.E.; M.S.; M.Phil.

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2013

Dedicated to all who indulged me in my detached desultory cogitation ...
and, to the particular few who goaded me out of it.

Acknowledgments

I would like to thank Professor Jacob Abraham, for his immense patience and confidence in me through the time it took to tie all ends of this thesis. His insight and inspiring discussions played a major role in deciding the course of this thesis. In addition to the research discussions, I would like to particularly acknowledge his Herculean role in helping me get past various administrative hurdles in being able to submit this thesis. Sir, to you, I owe this thesis.

In 2001, when I was trying to transfer from Yale to UT, Professor Adnan Aziz was responsible in making a quick admission possible for me. I fondly remember attending Adnan's group discussions, and discussing Quadratic Forms and Lyapunov Stability functions. Thank you Adnan, for the stimulating conversations, I still cherish them.

Dr. Rishiyur Nikhil was one of the first reasons I decided to do research. I met him in December of 1996 when I was in my undergraduate junior year, at an IEEE conference in Trivandrum, where he was giving a keynote address. It was one of the most inspiring events of my life at that time, and immediately resulted in my pursuing research and a Masters and PhD degree. Nikhil, I'm always thankful to you for seeding the joy of research in a 19-year old's mind.

I would like to thank the other members of my committee, Professors Sarfraz Khurshid and Lizy John for supporting me through my candidacy. I would also

like to thank my lab-mate and a dear friend from my UT days, Shobha Vasudevan. We worked together on a lot of research, ate late night donuts, wrote papers, and worked very well together. Shobha, I'm ever grateful to you, for keeping me going in good times and bad.

This work has meandered through different research topics and jumped through administrative hoops. None of this would have been possible without the unquestioned support of various admin staff, and Melanie Gulick in particular. Melanie, thanks for being so kind, thoughtful and sweet during the entire process. You are a gem. I would also like to acknowledge all my friends (too many to list here!) for their support in this process.

Lastly, and perhaps therefore, for the most motivation and support, I would like to thank my wife Roopa Taranath. She urged me to move back to Austin to finish the last part of my research and write the thesis. She has been accommodating of all my, sometimes absurd, demands and moods during this process. Thank you, Roopa, for your considerable patience and an irrational belief in my abilities. To you, I dedicate this thesis. There were times when I thought you wanted it more for me, than I wanted it myself.

Correct Low Power Design Transformations for Hardware Systems

Publication No. _____

Vinod Viswanath, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Jacob A. Abraham

We present a generic proof methodology to automatically prove correctness of design transformations introduced at the Register-Transfer Level (RTL) to achieve lower power dissipation in hardware systems. We also introduce a new algorithm to reduce switching activity power dissipation in microprocessors. We further apply our technique in a completely different domain of dynamic power management of Systems-on-Chip (SoCs). We demonstrate our methodology on real-life circuits.

In this thesis, we address the dual problem of transforming hardware systems at higher levels of abstraction to achieve lower power dissipation, and a reliable way to verify the correctness of the afore-mentioned transformations. The thesis is in three parts. The first part introduces *Instruction-driven Slicing*, a new algorithm to automatically introduce RTL/System level annotations in microprocessors to achieve lower switching power dissipation. The second part introduces *Dedicated Rewriting*, a rewriting based generic proof methodology to automatically prove correctness of such high-level transformations for lowering power dissipation. The third

part implements *dedicated rewriting* in the context of dynamically managing power dissipation of mobile and hand-held devices.

We first present *instruction-driven slicing*, a new technique for annotating microprocessor descriptions at the Register Transfer Level in order to achieve lower power dissipation. Our technique automatically annotates existing RTL code to optimize the circuit for lowering power dissipated by switching activity. Our technique can be applied at the architectural level as well, achieving similar power gains. We first demonstrate our technique on architectural and RTL models of a 32-bit OpenRISC pipelined processor (OR1200), showing power gains for the SPEC2000 benchmarks. These annotations achieve reduction in power dissipation by changing the logic of the design. We further extend our technique to an out-of-order superscalar core and demonstrate power gains for the same SPEC2000 benchmarks on architectural and RTL models of PUMA, a fixed point out-of-order PowerPC microprocessor.

We next present *dedicated rewriting*, a novel technique to automatically prove the correctness of low power transformations in hardware systems described at the Register Transfer Level. We guarantee the correctness of any low power transformation by providing a functional equivalence proof of the hardware design before and after the transformation. Dedicated rewriting is a highly automated deductive verification technique specially honed for proving correctness of low power transformations. We provide a notion of equivalence and establish the equivalence proof within our dedicated rewriting system. We demonstrate our technique on a non-trivial case study. We show equivalence of a Verilog RTL implementation of a

Viterbi decoder, a component of the DRM System-On-Chip (SoC), before and after the application of multiple low power transformations.

We next apply *dedicated rewriting* to the broader context of holistic power management of SoCs. This in turn creates a self-checking system and will automatically flag conflicting constraints or rules. Our system will manage power constraint rules using *dedicated rewriting* specially honed for dynamic power management of SoC designs. Together, this provides a common platform and representation to seamlessly cooperate between hardware and software constraints to achieve maximum platform power optimization dynamically during execution. We demonstrate our technique in multiple contexts on an SoC design of the state-of-the-art next generation Intel smartphone platform.

Finally, we give a proof of *instruction-driven slicing*. We first prove that the annotations automatically introduced in the OR1200 processor preserve the original functionality of the machine using the ACL2 theorem prover. Then we establish the same proof within our *dedicated rewriting* system, and discuss the merits of such a technique and a framework.

In the context of today's shrinking hardware and mobile internet devices, lowering power dissipation is a key problem. Verifying the correctness of transformations which achieve that is usually a time-consuming affair. Automatic and reliable methods of verification that are easy to use are extremely important. In this thesis we have presented one such transformation, and a generic framework to prove correctness of that and similar transformations. Our methodology is constructed in a manner that easily and seamlessly fits into the design cycle of creating complicated

hardware systems. Our technique is also general enough to be applied in a completely different context of dynamic power management of mobile and hand-held devices.

Table of Contents

Acknowledgments	v
Abstract	vii
Table of Contents	xi
List of Figures	xiv
Chapter 1. Introduction	1
Chapter 2. Low Power Techniques, Transformations and Verification	11
2.1 Low Power Transformations and Techniques	17
2.1.1 Circuit and Design Optimizations and Considerations for Power	17
2.1.1.1 Transistor Level Optimizations	17
2.1.1.2 Combinational Logic Optimizations	19
2.1.1.3 Sequential Logic Optimization	20
2.1.1.4 Survey of Gate Level Optimizations	21
2.1.2 Behavioral Level (RTL) Optimizations	22
2.1.3 Micro Architectural Techniques for Low Power	23
2.1.4 Dynamic Voltage and Frequency Scaling	24
2.1.5 Operating System Power Management	27
2.1.6 Memory Power Management	29
2.1.7 Compiler Techniques for Low Power	30
2.1.8 Surveys	32
2.2 Low Power Verification	32
2.2.1 Pre-Silicon Verification	34
2.2.2 Platform Verification	40
2.3 Conclusions	41

Chapter 3. Instruction-driven Slicing: Automatic Insertion of Low Power RTL Annotations	43
3.1 Introduction	43
3.1.1 Contributions of this work	44
3.2 Instruction-Driven Slicing	47
3.3 Our Technique	50
3.3.1 Instruction-Driven Slicing Algorithm	50
3.3.2 Methodology	52
3.4 OR1200 - a Pipelined OpenRISC Implementation	54
3.4.1 OR1200	55
3.4.2 Results for OR1200-RTL	55
3.4.3 Results for OR1200-Arch	63
3.5 Instruction-Driven Slicing in the PowerPC Microprocessor	64
3.5.1 PUMA	65
3.5.2 Results for PUMA-RTL	71
3.5.3 Results for PUMA-ARCH	73
3.6 Conclusions	75
Chapter 4. Dedicated Rewriting: Correctness of Low Power Transformations in RTL	77
4.1 Introduction	77
4.2 Rules	81
4.2.1 Structural Rules	85
4.2.2 Logical Rules	87
4.3 Dedicated Rewriting	89
4.3.1 <i>derive</i> (): Verilog RTL to TRS rules	90
4.3.2 <i>execute</i> (): TRS rules to expressions	91
4.3.2.1 Reassignments	92
4.3.3 <i>prove</i> (): Equivalence of expressions	96
4.3.4 Our notion of equivalence	97
4.3.5 Error Detection	98
4.4 Dedicated Rewriting for Combinational Equivalence Checking: Multiplier Verification	100

4.5	Case Study: Viterbi Decoder	104
4.5.1	Combinational low power transformations	107
4.5.2	Sequential low-power optimizations	108
4.5.3	Optimizations for power and timing	109
4.5.4	Correctness of low power transformations on the Viterbi de- coder	109
4.6	Discussion and Conclusions	113
Chapter 5. Holistic Power Management of SoCs using Dedicated Rewriting		115
5.1	Introduction	115
5.2	Rules	118
5.3	Power Constraints Consistency Checker	121
5.4	Dedicated Rewriting as a Dynamic Power Management Rule Engine	123
5.5	Case Studies	124
5.5.1	Audio Playback	126
5.5.2	Panel Self Refresh	127
5.5.3	Results	128
5.5.4	Discussion	128
Chapter 6. Correctness of Instruction-driven Slicing		130
6.1	Introduction	130
6.2	Automatic Proof Technique	131
6.3	Interactive Proof by Deductive Verification	135
6.3.1	Proof using the ACL2 theorem prover	135
6.3.2	Comparing a dedicated rewrite system versus a generic the- orem prover	140
6.4	Discussion and Conclusions	141
Chapter 7. Discussion and Conclusions		142
Bibliography		146
Vita		166

List of Figures

1.1	Power consumption trend in Servers and Data Centers (courtesy, Intel).	3
1.2	Power consumption trend in Embedded and Handheld Devices (courtesy, Nokia).	4
2.1	Energy efficient design	12
2.2	Power dissipation in CMOS devices.	14
2.3	Designing for Low Power	15
2.4	Body Bias vs. Internal Vdd On/Off [17].	18
2.5	New type of bugs introduced in a low power methodology [153]. . .	33
2.6	New verification tasks introduced in a low power methodology [153].	34
2.7	Verification cost of low power transformations.	36
3.1	Overview of the Instruction-driven Slicing Algorithm for RTL. . . .	48
3.2	Incorporating <i>Instruction-driven slicing</i> into the design flow. . . .	52
3.3	OR1200 Processor Block Diagram.	54
3.4	Instruction-driven slicing example.	56
3.5	OR1200-RTL Power dissipation results after slicing on 1, 4 and 10 instructions. These results are for SPECINT2000 benchmarks with Synopsys clock gating	57
3.6	OR1200-RTL Power dissipation results after slicing on 1, 4 and 10 instructions. These results are for SPECINT2000 benchmarks without Synopsys clock gating	58
3.7	OR1200 reduction in power dissipation for SPECINT2000 benchmarks.	59
3.8	OR1200-RTL Power reduction compared to increase in area and delay due to slicing. The first set of comparisons depicts the normalized <i>Energy – Area</i> product. The second set depicts the normalized <i>Energy – Delay²</i> product.	60
3.9	Flop distribution effect of instruction-driven slicing on 1 . add and 1 . lw in the OR1200 RTL.	61

3.10	OR1200-Arch Power dissipation results for SPECINT2000 benchmarks after slicing on 1, 4 and 10 instructions.	63
3.11	PUMA Fixed Point Unit Processor Block Diagram.	65
3.12	PUMA-RTL Power dissipation results after slicing on 1, 4 and 10 instructions. These results are for SPECINT2000 benchmarks with Synopsys clock gating	66
3.13	PUMA-RTL Power dissipation results after slicing on 1, 4 and 10 instructions. These results are for SPECINT2000 benchmarks without Synopsys clock gating	67
3.14	PUMA-RTL Power gains for SPECINT2000 benchmarks.	68
3.15	Flop distribution effect of instruction-driven slicing on <code>l.add</code> and <code>l.lw</code> in the PUMA RTL.	69
3.16	Time delay and Area estimate for the PUMA RTL.	70
3.17	Power vs. Delay for the PUMA-RTL.	70
3.18	Power vs. Area for the PUMA-RTL.	71
3.19	PUMA-Arch Power dissipation results for SPECINT2000 benchmarks after slicing on 1, 4 and 10 instructions.	74
3.20	PUMA-Arch Power gains for SPECINT2000 benchmarks.	75
4.1	Term Rewriting Systems: Definitions and concept	81
4.2	Term Rewriting Systems: Definitions and concept (continued)	82
4.3	Sample Verilog RTL and the TRS Rules derived from it.	83
4.4	Clock gating for lower switching activity power dissipation (Verilog RTL).	84
4.5	Clock gating for lower switching activity power dissipation (after translation to TRS).	86
4.6	Sample logical rules in the dedicated rule database.	87
4.7	Dedicated rewriting proof system flow chart.	90
4.8	Dedicated rewriting algorithm	95
4.9	Comparison of execution times of Dedicated Rewriting against two commercial equivalence checkers for Booth, Wallace Tree and Dadda Tree multipliers of varying sizes. In each case the golden model was a shift and add multiplier of the corresponding size.	102
4.10	Number of proof iterations done by <i>reduce()</i> to prove equivalence at the given compare points. The numbers correspond to the 64×64 Booth, Wallace Tree, and Dadda Tree multiplier designs.	103
4.11	Basic Viterbi design.	105

4.12	Viterbi design optimized for low power.	106
4.13	Viterbi design optimized for power and delay.	106
4.14	Results of power estimation due to combinational logic low power transformations in the Viterbi decoder. These estimates were based on the macro-modeling technique employed by Gupta <i>et al</i> [50]. These estimates are only over the trellis computation calculation function of the Viterbi decoder.	108
4.15	Results of power estimation after sequential low power transformations in the Viterbi decoder.	108
4.16	Results of power estimation after sequential optimization for low power and timing in the Viterbi decoder.	109
4.17	Picturization of sequential equivalence checking of transformed TRS_p against the original TRS_o of the Viterbi design. The first row is the proof of the FF buffer (over 8 time cycles). The center row shows the proof of the Trellis Computation and the bottom row shows the proof of the MatDec Decision Table.	111
5.1	Rule-based formal Power Specification and Management	116
5.2	Audio player structural and logical rules	118
5.3	Low power audio playback policy	125
5.4	Panel self refresh policy for video playback and web browsing applications	127
5.5	Summary of results without and with dynamic power management by the rewriting rule engine	129
6.1	Instruction-driven slicing example from Figure 3.4 reproduced here for the reader's convenience.	132
6.2	Rules derived before instruction-driven slicing transformation.	133
6.3	Rules derived after instruction-driven slicing transformation.	134
6.4	Proof methodology.	136
6.5	<i>Verilog2ACL2</i> example.	138
6.6	Instruction-driven slicing example after <i>Verilog2ACL2</i> (pre-transformation).	139
6.7	Instruction-driven slicing example after <i>Verilog2ACL2</i> (post-transformation).	140

Chapter 1

Introduction

The importance of power management research has increased tremendously in the last decade with increasing maturity of low power design techniques in the context of micro architecture for small devices and embedded systems. Unprecedented growth in the proliferation of handheld mobile devices has been one of the catalysts for this research, even as power efficient computing remains the holy grail of almost all aspects of computing systems today. There are several trends here that are worth noting.

The last few years have seen the emergence of highly integrated embedded System-on-a-chip (SoC) architectures for several usages and platforms such as high end mobile devices and smartphones. While each SoC component or accelerator can be optimized in various ways, overall platform integration and platform power optimization is a growing challenge since it is not clear how to quantify the impact of applications and workloads on individual platform components. Another trend has been the emergence of multi-core and multi-threaded architectures for all kinds of computing devices, ranging from cell phones to tablets, laptops, and netbooks, to high end computing systems, servers, etc. As the number of cores and threads-per-core increases, such systems present unique challenges in terms of scheduling,

energy efficiency, temperature, heterogeneity, etc.

On such increasingly large and complex digital IC and SoC designs, design power closure and circuit power integrity are starting to become the key engineering challenges, thereby impacting the device's total time-to-market. Power dissipation has emerged as an important design parameter in the design of microelectronic circuits, especially in portable computing and personal communication applications where usage models are becoming increasingly important to the consumer with more and more usages of small devices demanding the performance and connectivity of traditional larger computing systems.

The amount of power consumed by some devices can cause significant design problems. Even in the case of devices intended for use in non-portable equipment where ample power is readily available, power-aware designs can offer competitive advantages with respect to such considerations as size and cost of the power supply and cooling systems. These trends have forced platform designers to abandon traditional performance metrics in favor of performance-per-watt. Additionally, aspects like overall platform security, application quality of service, critical thermal conditions, etc. are considered highly critical in the usability of a device.

One of the biggest challenges for data-center operators today is the increasing cost of power and cooling as a portion of the total cost of operations. As shown in Figure 1.1, the cost of power and cooling has increased 400%, and these costs are only on the rise. In some cases, power costs account for 40-50% of the total data-center operation budget. The need to deploy more servers to support new business solutions is only increasing. Data centers are therefore faced with the twin problem

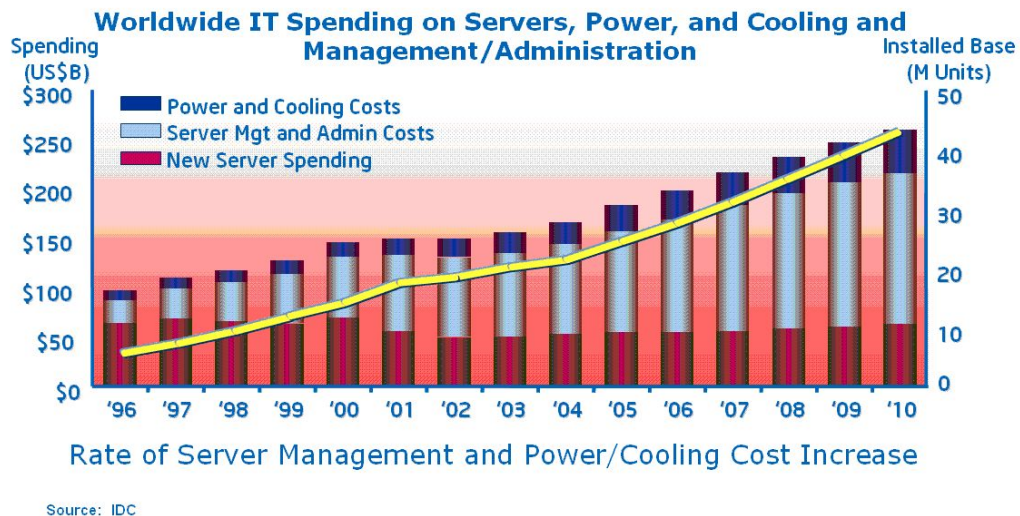


Figure 1.1: Power consumption trend in Servers and Data Centers (courtesy, Intel).

of how to deploy new services in the face of rising power and cooling costs.

On the other end of the spectrum, power consumption of handheld devices are also on the rise with increasing demands being placed on such devices. Figure 1.2 from Nokia shows the trend of power consumption in the last few years in handhelds. As can be seen, the last few years since 2003 has seen a dramatic increase in the power consumption as well as the usages. Increasingly devices are using up more bandwidth with higher power radios with the emergence of 3G, LTE, etc. and users are expecting the same power of computing from handhelds as was previously being accomplished on larger laptops, desktops, etc.

Power management and optimization research in the last couple of decades has spanned multiple areas - at circuit/design level, hardware and micro-architectural

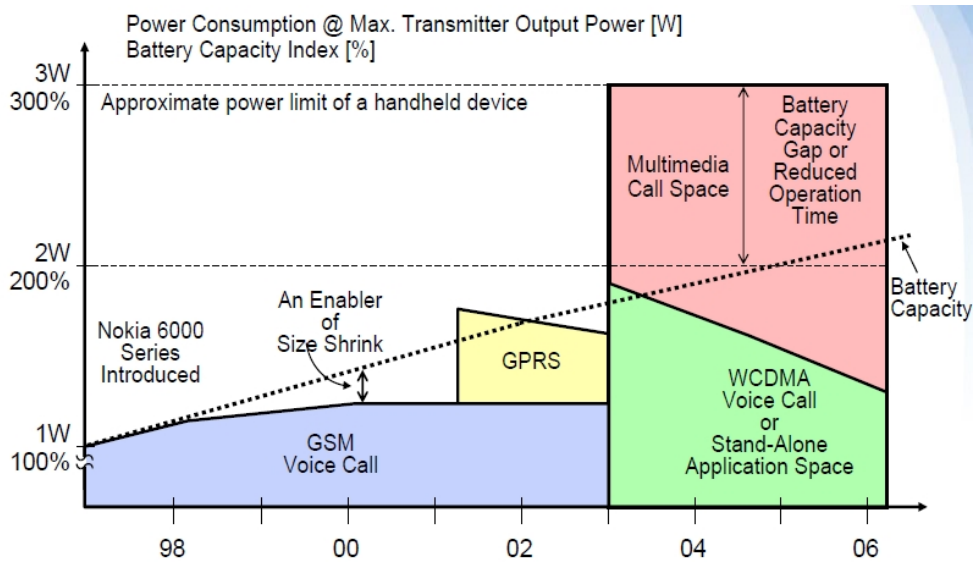


Figure 1.2: Power consumption trend in Embedded and Handheld Devices (courtesy, Nokia).

level for processors, caches, memories, power management of individual components such as hard drives, external memories, network interfaces. In addition, there has also been research in power-aware compiler optimizations, operating system optimizations for energy efficiency, etc. While power management has traditionally been thought to be for smaller/embedded devices, recently there has been a lot of work on power management in large systems like servers, data centers, etc. There has also been an increased focus toward platform-wide power management that aims to unite different power management techniques and capabilities.

Currently, platform power management broadly falls under two categories. At the one end are low power optimizations implemented by hardware designers like gate level and RTL optimizations, clock and power gating, low power microarchi-

tectural innovations, etc. Power-related constraints are now being imposed throughout the entire design flow in order to maximize the performance and reliability of devices. In the case of today's extremely large and complex designs, implementing a reliable power network and minimizing power dissipation have become major challenges for design teams. Creating optimal low-power designs also involves making trade-offs such as timing-versus-power and area-versus-power at different stages of the design flow. These optimizations are based on generic techniques in the sense that they are done quite independent of the rest of the platform components, and in most cases, with the assumption that hardware is the best judge of what power optimizations to use.

On the other end are the power management techniques that are available for the OS and devices to control. This includes OS control of processor C-states and P-states and device power states, with standardized interfaces like Advanced Configuration and Power Interface (ACPI) that regulate such state control based on the workload and configured policies. In recent years, the focus of power management has moved to more areas than the CPU alone given the trend of increasing power densities of all computing components on the platform. This includes caches, memory modules, buses, storage devices, etc. DRAMs, for instance, have multiple power states (deep and shallow self refresh, support for partial array self refresh, etc.). Both CPU and bus frequencies can be dynamically voltage- and frequency-scaled. In addition, embedded/handheld mobile devices also employ thermal throttling to maintain the entire platform and/or individual component under a thermal envelope.

To effectively reduce dynamic power, hardware designers must understand a multitude of low-power transformations and have the practical experience to know when they should be applied. The trade-off between power reduction and verification cost is not always clear which leads the architects and designers to be cautious and conservative in their implementation. Often aggressive low power transformations are not implemented in the design because the verification cost (typically more than 70% of the total time cost) is high enough to adversely impact the total time-to-market. The primary reason for prohibitive verification cost is the hardness of the verification problem. If the low power transformations are at the gate level, then equivalence checking methods can be used to automatically prove the correctness of the transformations. Although there are a few sequential equivalence checking algorithms, they are not widespread in the hardware industry due to their intractable, or hard-to-use nature.

With the emerging mobile internet devices market, the need for quick verification turn-around times on efficient and effective low power transformations is extremely important. The hitherto unexplored (in large measures) area of optimizations is RT-level low power transformations which can utilize the program design information of the RTL in its computation of where to lower dynamic power dissipation. These transformations definitely introduce new variables and state, and change the timing information of the design. Therefore verifying the correctness of these transformations becomes the problem of verifying the equivalence of two sequential circuits. Verifying the equivalence of two sequential circuits is an NP-hard problem. Rather than go after the holy grail, it might be more prudent to focus the

verification problem to equivalence of two sequential circuits that differ only because of the application of a low power transformation. By restricting the domain of difference, one can focus on algorithms that will work well with the domain in context. Therefore, what is of essence, ideally, is effective and efficient RT-level low power transformations, which can be, to the extent possible, automatically proved correct when applied to a particular hardware design.

Platform level optimizations are very coarse-grained and verification of any change in method essentially means a complete re-verification of the system. The optimizations at the platform level include the knowledge of the application and user-intent, and therefore are optimized for exact use-cases of the device. This knowledge is an extremely fine-grained information to achieve a fairly coarse-grained power optimization and has been the way the industry has been trending lately, particularly in the handheld device market.

Given the above challenges, it is very clear that neither hardware nor software in isolation can make the best decision about power and performance management, and neither can these be done for CPU alone, but must now be done at the entire platform level, in a holistic way. Such a holistic power management solution cannot ignore any of the platform components from a power and thermal perspective, neither can it be done in hardware or software alone. We need a synergistic bottom line across RTL, system level, OS level, compiler level, and application level specification of power intent. The kind of information available at a higher level of abstraction at the OS level or even at the application level, is just not visible to the RTL. We need all levels of abstraction of the design to be able to communicate their

power intent to get the most optimized solution. IEEE standard specification format UPF (Unified Power Format [60]) allows us to represent power intent at the RTL and lower levels. We require a method to represent power intent specifications at higher levels of abstraction as well, and a mechanism that can coordinate all such power management capabilities dynamically at runtime with constant feedback from the system about platform conditions (overall power consumption, thermal conditions, etc.).

The key contributions of this thesis are the following:

- We have introduced *Instruction-driven slicing* an effective and efficient RT-level low power transformation in microprocessors and microcontrollers. *Instruction-driven slicing* automatically introduces RTL annotations to achieve lower switching power dissipation.
- We have demonstrated the effectiveness of *instruction-driven slicing* on OR1200 an in-order RISC pipelined microprocessor and on PUMA, a more complicated out-of-order PowerPC core.
- We have proposed *Dedicated Rewriting*, a proof framework and a methodology to automatically (after some initial training) prove the correctness of RTL low power transformations.
- We have demonstrated the proof methodology on complicated arithmetic circuits, proving correctness of 128-bit Booth multipliers.

- We have demonstrated *dedicated rewriting* on a non-trivial System-on-Chip, the Viterbi decoder that is part of the Digital Radio Mondiale.
- We have given a correctness proof of the *instruction-driven slicing* algorithm as applied to a sample microprocessor. Given that the algorithm is correct by construction, this is sufficient for all microprocessors.
- We have applied *dedicated rewriting* in the context of dynamic power management of SoCs for mobile and hand-held devices in order to achieve communication between power constraints across different levels of design abstraction and to verify consistency of those constraints.
- The dynamic power manager is also a runtime engine which can on-the-fly drive the device to certain power states, based on the rule engine outcome to manage user-specified power intent for each workload.
- We demonstrate the holistic dynamic power constraint checker and manager on SoCs built on state-of-the-art Intel smartphone platforms.

The thesis is organized as follows. Chapter 2 gives a survey of low power transformations. We describe our *instruction-driven slicing* algorithm, and its effectiveness in full detail in Chapter 3. We then describe our *dedicated rewriting* system and its application to arithmetic circuits and an SoC in Chapter 4. We further apply our *dedicated rewriting* system in the context of holistic power management of SoCs, in Chapter 5. In Chapter 6 we give a proof of *instruction-driven slicing* in our *dedicated rewriting* system as well as in a general purpose theorem prover, ACL2.

Finally, we discuss some of the relevance and merits of this thesis, and potential future work and conclude in Chapter 7.

Chapter 2

Low Power Techniques, Transformations and Verification

Power Management Methodology as an architectural paradigm is still in its infancy. Most techniques in today's designs still follow ad-hoc methodologies. However, most systems are typically amenable to energy efficient design. Figure 2.1 addresses in detail the principles behind energy efficient design. The two key principles are that: (a) apply the lowest voltage to each block at any given time; and (b) keep a block on only when required. There is also a tradeoff between performance and power. In a *race to idle* strategy, the functional block tries to complete its task as fast as possible (high performance) and then shuts off. In a *crawl to idle* strategy, the functional block tries to delay the task as long as possible, while consuming as little power as possible during that process. The power-delay product in any system is conserved. An intelligent management system will choose correctly when to *race* or *crawl* to idle [73].

Sources of power dissipation in CMOS devices are summarized by the following expression:

$$P = \underbrace{\frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot f \cdot N}_{\text{Switching}} + \underbrace{Q_{SC} \cdot V_{DD} \cdot f \cdot N}_{\text{Short Circuit}} + \underbrace{I_{leak} \cdot V_{DD}}_{\text{Leakage Current}} \quad (2.1)$$

Most today's systems are amenable to energy efficient design:

- Systems do not have the same performance needs at all times, and operating at worst case design corner all the time is wasting power.
- The principle of Lowest Active Power is to *apply the lowest possible V_{dd} to each functional block at each instant of time.*
- Systems also do not need all functions to be running at the same time.
- The principle of Zero Idle Power is to *turn on a block only when necessary; turn it off once it is no longer needed.*
- What consumers want in today's complex designs are:
 1. High Performance
 2. Multi-media experience,
 3. Lowest Active Power
 4. Zero Idle Power.

Figure 2.1: Energy efficient design

where P denotes the total power, V_{DD} is the supply voltage, and f is the frequency of operation [47], [27].

The first term represents the power required to charge and discharge circuit nodes. Node capacitances are represented by C . The factor N is the switching activity, *i.e.*, the number of gate output transitions per clock cycle. Figure 2.2(a) shows the dynamic power dissipation at a switching gate.

Node capacitance C , depends largely on wire lengths of on-chip structures, and is therefore an architectural metric for determining the trade-offs - for example, between single monolithic large processor cores or smaller processor cores since the latter option is likely to reduce average wire lengths. Similarly, smaller cache mem-

ories or independent banks of cache are likely to reduce wire lengths since many address and data lines will only need to span across each bank array individually.

Supply voltage V_{DD} is one of the most important aspects of power-aware design given its quadratic influence on dynamic power. We will talk more about this subsequently.

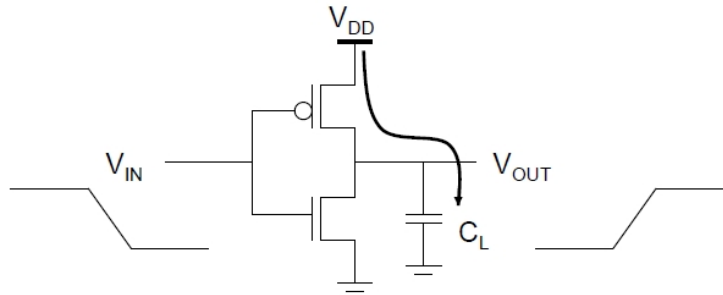
Switching Activity factor N , refers to how often wires actually transition from 0 to 1, or 1 to 0. Techniques such as clock gating are used to save energy by reducing activity factors during a hardware units idle periods.

The clock frequency f , in addition to influencing power dissipation, also influences the supply voltage. Typically, higher clock frequencies will mean maintaining a higher supply voltage. Thus, the combined $V^2 f$ portion of the dynamic power equation has a cubic impact on power dissipation.

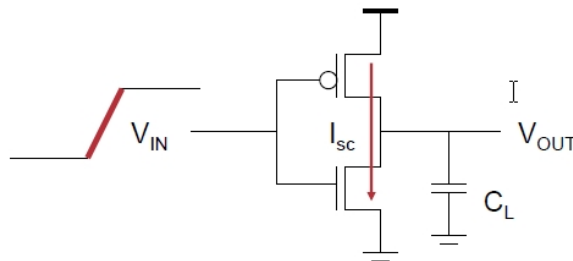
Strategies such as dynamic voltage and frequency scaling (DVFS) try to exploit this relationship to reduce (V, f) accordingly.

The second term in (Equation 2.1) represents power dissipation during output transitions due to current flowing from the supply to ground. This current is often called short-circuit current. The factor Q_{SC} represents the quantity of charge carried by the short-circuit current per transition. Figure 2.2(b) shows the power dissipation due to short circuit current. As the output load capacitance increases, the voltage transition time also increases.

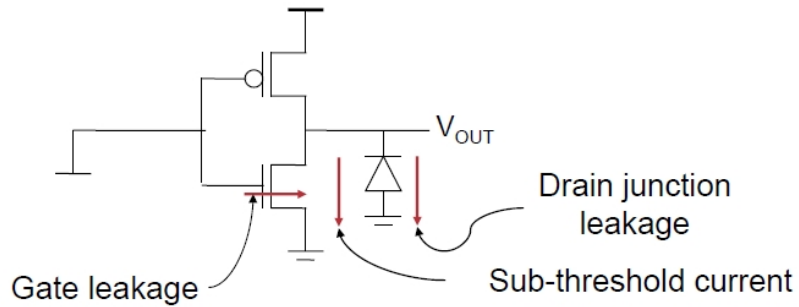
The third term in (Equation 2.1) represents static power dissipation due to leakage current I_{leak} . Devices source and drain diffusions from parasitic diodes with



(a) Dynamic Power Dissipation due to switching activity.



(b) Power Dissipation due to Short-Circuit current.



(c) Static Power Dissipation due to leakage current.

Figure 2.2: Power dissipation in CMOS devices.

How do we design for Lowest Active Power and Zero Idle Power?

- Lowest Active Power
 1. Clock Gating, Device sizing (*old methods*)
 2. Multi-Vdd (*spatial voltage control*)
 3. DVFS (*temporal voltage control*)
- Zero Idle Power
 1. Multi-Vt (*old method*)
 2. Power-gating/retention
 3. Low Vdd standby, back bias

Figure 2.3: Designing for Low Power

bulk regions. Reverse bias currents in these diodes dissipate power; sub-threshold transistor currents also dissipate power. Figure 2.2(c) shows the leakage power dissipation even when devices are not switching.

Sub-threshold leakage power represents the power dissipated by a transistor whose gate is intended to be off. The main reason behind this leakage is that transistors do not have ideal switching characteristics, and thereby leak a non-zero amount of current even for voltages lower than the threshold voltage.

Figure 2.3 details a list of techniques on how to design based on the principles identified in Figure 2.1. With process technologies below 100 nm, static power consumption has become a prominent and, in many cases, dominant design constraint. Due to the physics of the smaller process nodes, power is leaked from transistors even when the circuitry is quiescent*. New design techniques have been developed

*No toggling of nodes from 0 to 1 or vice versa

to manage static power consumption. Power gating [151], [154] or power shut-off turns off power for a set of logic elements; back bias techniques [31] are used to raise the voltage threshold at which a transistor can change its state. While back bias slows the performance of the transistor, it greatly reduces leakage. These techniques are often combined with multi-voltages [108] and require additional functionality: power management controllers [33], isolation cells [116] that logically and/or electrically isolate a shutdown power domain from powered-up domains, level-shifters [116], [32] that translate signal voltages from one domain to another, and retention registers [116] to facilitate fast transition from a power-off state to a power-on state for a domain.

In the rest of this thesis, we will refer to the three terms above as switching activity power, short-circuit power and leakage current power. A detailed survey of many of these optimization methods is given in [41]. Most of the optimizations described in this survey concentrate on minimizing switching activity power at various levels of abstraction.

One obvious way to reduce switching activity power is to reduce V_{DD} (because it appears as a squared term in (Equation 2.1)), and to reduce C , amount of capacitance we have to switch. Reduced V_{DD} implies reduced threshold voltage, which in turn implies increased leakage current I_{leak} . Static power dissipation due to leakage current increases exponentially as voltages are scaled down. Since scaling down of voltages is a natural by-product of scaling down the dimensions (width and length of transistors), static power dissipation is a huge problem in nano-scaled CMOS circuits. However, it is not clear how to model or reason about static power dissipation

due to leakage current at the RT-level.

In this survey, we first survey state-of-the-art optimization methods that target low power dissipation in VLSI circuits. Design optimizations for low power at the circuit, logic, architectural and system level are considered in Section 2.1. Section 2.2 details various state-of-the-art attempts at verification of low power transformations, techniques and optimizations. [147] carries more details.

2.1 Low Power Transformations and Techniques

2.1.1 Circuit and Design Optimizations and Considerations for Power

2.1.1.1 Transistor Level Optimizations

Complex gates are designed such that the late-arriving signals are placed closer to the output to minimize gate propagation delay. However, the average power dissipation is dependent on the transition probabilities of the gate inputs and the internal node capacitances (parasitic drain and source, as well as interconnect capacitance). Therefore, ordering of gate inputs will affect both power and delay. Methods to optimize the power and/or delay of logic gates by transistor re-ordering is given in [104],[125].

Transistor size of any given gate is inversely proportional to the delay of the gate, and directly proportional to the power dissipated in the gate. An increase in transistor size will also increase the delay of the fan-in gates feeding this gate because of increased load capacitance for the fan-in gates. The problem to be tackled here is that given a delay constraint, we have to size the transistor to minimize power dissipation. There are some approaches which treat this problem as a combinato-

rial problem and try to solve it using genetic algorithms, or simulated annealing. Another approach is to compute the slack at each gate of the circuit (how much the gate can be slowed down without affecting the critical delay) [125], [11], [37]. An approach using nonlinear optimization formulation, by introducing a notion of transition density is expostulated by [112]. In particular, [112] also does a good survey of other approaches which have taken power into consideration during transistor sizing.

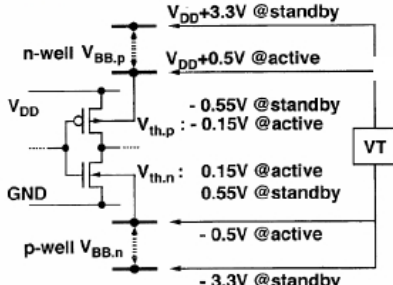
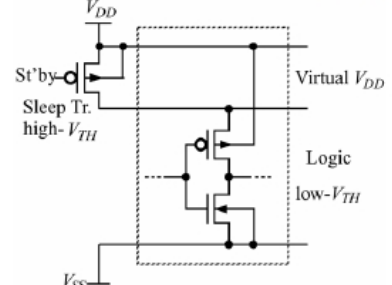
Body Bias (VTCMOS) V_{TH} Control With Substrate Bias	MTCMOS On/Off Control of Internal V_{DD}/V_{SS}
	
<ul style="list-style-type: none"> Needs Circuit Development Compensate ΔV_{TH} Fluctuation IDDQ Test No Serial MOSFET Conventional EDA Tools Re-Use of Existing Design Triple Well Desirable 	<ul style="list-style-type: none"> Conceptually Simpler Compensate ΔV_{TH} Fluctuation IDDQ Test Large Serial MOSFET Conventional EDA Tools Retention Registers Conventional Well structure

Figure 2.4: Body Bias vs. Internal Vdd On/Off [17].

Figure 2.4 contrasts two key transistor level changes to accommodate power management. Described on the left hand side is body bias. Adaptive Body Biasing (ABB) [45] is a popularly used technique to mitigate the increasing impact of manufacturing process variations on leakage power dissipation. The efficacy of the

ABB technique can be improved by partitioning a design into a number of *body-bias islands*, each with its individual body-bias voltage.

On the right hand side of Figure 2.4 is described a multi-threshold CMOS technology (MTCMOS) [49] which allows for easy switching between ON and OFF states. Such transistors are very useful to build retention registers [10], etc.

2.1.1.2 Combinational Logic Optimizations

A comprehensive treatment of combinational logic optimization for area and delay is given in [40]. In this subsection we focus on power related optimizations.

Methods to reduce circuit switching activity (and hence, power dissipation), by using controllability-don't-care-set[†] and observability-don't-care-set[‡] was presented in [61] and further improved in [12].

Delays of paths converging at any gate are typically balanced to avoid spurious transitions. Path balancing is done by adding unit-delay buffers on the faster paths, without affecting the timing. However, adding buffers can affect the reduction in switching activity. [74] discuss how to balance paths by reducing instead of completely eliminating spurious switching activity, by adding a minimum number of unit-delay buffers.

Factorization involves finding common sub-expressions and reusing them to reduce transistor count. In kernel extraction algorithms for factorization (discussed

[†]The controllability don't-care set corresponds to collections of input combinations that never occur at the gate inputs.

[‡]The observability don't-care set corresponds to collections of input combinations which produce the same values at the circuit outputs.

in [110]), the kernels of given expressions are generated and kernels that maximally reduce switching activity are selected.

2.1.1.3 Sequential Logic Optimization

Methods to encode state transition graphs to produce two-level and multi-level implementations with minimal power are described in [110] and [130]. An algorithm to re-encode logic level sequential circuits to minimize power dissipation is presented in [52].

Encoding to reduce switching activity in datapath logic has also been a subject of attention. A method to minimize switching on buses is proposed in [123]. In this method, an extra line is added which indicates whether the value on the bus is true or complemented. For example, if the previous value on the bus is 0000 and the value to be transmitted is 1101, then we can reduce switching activity by transmitting 0010 and setting the extra line to denote the complement.

Retiming is a well-researched optimization technique to reposition flip-flops in a synchronous sequential circuit to minimize the clock period. There isn't much work done in terms of reducing power dissipation using retiming. Although [89] exploit the fact that due to spurious transitions switching activity at the inputs of flops are substantially more than at the outputs, in order to retime for low power.

Gated Clocks is the most used power optimization technique. This involves "turning off" parts of the circuit governed by a clock which are not used on any given cycle. The cost is some additional circuitry which will decide whether to clock some flops on certain clock cycles or not.

Precomputation [4] builds on gated clocks. Idling sub-circuits are detected and "turned off". In sequential circuits, the output logic values are selectively precomputed one cycle before they are needed, and these precomputed values are used to reduce internal switching activity in the succeeding cycle.

[90] and [129] give algorithms to determine the sub-circuits to be turned off, and the logic required to perform the disabling.

2.1.1.4 Survey of Gate Level Optimizations

[76] presents a very good estimation of the power consumption in CMOS VLSI chips. It presents different ways of estimating power consumption in logic, memories, interconnects, clock distribution, and off chip components. This work also presents a method to estimate the power consumption of a chip based on gate count, memory size, logic, layout styles, etc. [94] presents a power optimization and synthesis system that can optimize power at the gate level and also perform area and timing optimizations. [107] presents a model that predicts not only the cycle-by-cycle power consumption of a module, but also the moving average of power consumption and the power profile of the module over time. [119] gives an excellent perspective on power-aware CAD tools and methodologies. [99] presents a very detailed survey of the important areas of hardware power optimizations. Some of the broad power-aware design methodologies described here include power-aware algorithm and system design (for example - a given task may be partitioned between various hardware modules or programmable processors or both to reduce the system-level power consumption), clock gating, memory power reduction by

segmenting/partitioning, power-aware behavioral and logic synthesis, etc.

2.1.2 Behavioral Level (RTL) Optimizations

An important aspect of optimizing power at the RTL level is to first develop a framework for analyzing the power dissipation at an architectural level. The traditional method has been to translate the given high level architecture description to gates (netlists) and then use reasonably accurate low-level power analysis engines. This method is infeasible if we want to evaluate a large number of design choices.

Most initial work in this area is focused on power analysis and reduction in caches. This is chiefly because embedded microprocessors, historically the reason for low power design, devote nearly 40% of their power budget to caches. Besides, caches are regular structures and are more easily modeled than other circuits. [29] discuss power reduction by reducing unnecessary speculation in branch predictors. [19] discusses gated clocks in integer ALUs. More recently, [20] presents a framework for analyzing power dissipation at the architecture level and [55] discusses optimizations at the microarchitectural level.

[97] propose a multiple clocking scheme for low power RTL design. The basis of this technique is: (a) to use a multiple clocking scheme of n non-overlapping clocks, by dividing the frequency of a single clock into n cycles; (b) to partition the circuit into n disjoint modules and assign each module to a distinct clock; and (c) to operate each module only during its corresponding duty cycle. The overall frequency remains the same, and at best $\frac{1}{n}$ of the original power is dissipated.

Sequential optimization seeks to replace a given sequential circuit with another

one optimized for some criterion – area, power, or performance, in a way such that the environment of the circuit cannot detect the replacement.

[86] computes logical redundancies in the circuit by a method of recursive learning [72] and eliminates the redundant gates. Identifying and eliminating redundant latches was first studied in [106] and was further explored in [121]. [120] explores safe replacement of sequential circuits. Sequential logic transformation integrating retiming with logic transformations at the technology-independent level is explored in [16].

2.1.3 Micro Architectural Techniques for Low Power

Power management for microprocessors can be done over the whole processor, or in specific areas. CPUs, for example, may have their execution suspended simply by stopping the issuance of instructions or by turning off their clock circuitry. Deeper power states however, might successively remove power from the processor’s caches, translation lookaside buffers (TLBs), memory controllers, and so on. There is a corresponding latency of deeper power states, and therefore extra energy is required to save and restore the hardware contents, or restart it, or both. Most modern processors support multiple low power states that can be exploited either independently by hardware (hardware idle detection) or through hints from the Operating system. Some examples are Intel’s SpeedStep [93], AMD’s Cool’n’Quiet, Transmeta’s LongRun technologies [44].

Utilizing the program structure or the data flow information available at the architectural and RTL levels can lead to many interesting and complicated low

power transformations [97], [20], [29], [145]; yet most power transformations in today's designs are at the gate-level [104], [4], [129], [90]. The primary reason for this is the hardness of the verification problem. If the low power transformations are at the gate level, then equivalence checking methods [85] can be used to automatically prove the correctness of the transformation. Although there are a few sequential equivalence checking algorithms [138], [59], [148], [149], they are not widespread in the hardware industry. In a typical industry scenario, an RTL or architectural low power transformation implies a full cost of dynamic validation, which can extend to many months, and require a lot of resources. Standardizing power intent specification will largely aid in reducing design and verification cost.

Another aspect of optimization for low power at the microarchitectural level is focused on power analysis and reduction in caches [111], [157]. Current implementations are limited only to *smart sizing* caches, which is done by the micro code in the core. [137] defines application specific cache partitions, called cache molecules, and resizing them to address performance targets for applications. None of these are visible or controllable from software/OS.

2.1.4 Dynamic Voltage and Frequency Scaling

The main premise in Dynamic Voltage and Frequency Scaling (DVFS) is that a system, a task, or a program can be slowed down with a small impact on its performance (presumably under acceptable limits), while at the same time obtaining significant savings in power consumption by voltage scaling. Power consumption has a linear dependence on frequency and a quadratic dependence on voltage (Equation

2.1). Power savings can therefore be achieved by intelligently reducing frequency while concurrently reducing the supply voltage. Reducing the frequency reduces the idle time in the system (slack in CPU execution, or instruction slack due to memory accesses in memory-bound program phases, etc). Many commercial implementations are now available for DVFS - Intel's SpeedStep, AMD's PowerNow, for example.

DVFS has been applied at both hardware and operating system/platform level. The main idea is to scale the supply voltage as low as possible for a given frequency while still maintaining correct operation. The voltage can be dropped only up to a certain critical level, beyond which timing faults occur. Some of the hardware mechanisms for DVFS [152], [43] implemented timing fault detection in hardware itself using special, "safe" flip flops that detect timing violations. While DVFS methods are effective in addressing the dynamic power consumption, they are significantly less effective in reducing the leakage power. As minimum feature sizes shrink, supply voltage scaling requires a reduction in the threshold voltage which results in an exponential increase in leakage current with each new technology generation. In [83], the authors show how the simultaneous use of adaptive body biasing (ABB) and DVFS can be used to reduce power in high-performance processors. ABB was previously used to control leakage during standby mode, and has the advantage that it reduces the leakage current exponentially, whereas dynamic voltage scaling reduces leakage current linearly. [132] and [156] look at similar other aspects of DVFS.

At the operating system level, several OSes now deploy some form of DVFS.

For example, Linux uses a very standard infrastructure called *cpufreq* to implement DVFS. *cpufreq* is the subsystem of the Linux kernel that allows frequency to be explicitly set on processors. *cpufreq* provides a modularized set of interfaces to manage the CPU frequency changes - it exposes common interface to the various low-level, CPU-specific frequency control technologies and high level CPU frequency controlling policies. *cpufreq* decouples the CPU frequency controlling mechanisms and policies and helps in independent development of the two. The actual policies are implemented as “governors”, the most popular one being the *ondemand governor*. There have been many variations of these governors that have been proposed for different kinds of systems, that have varying requirements/constraints with respect to power and performance [96].

Gurun et al [51] and Simunic et al [118] look specifically at handheld, portable, and embedded systems and propose different techniques for implementing DVFS in such battery constrained devices. For example, in [51], the authors present AutoDVS, a dynamic voltage scaling (DVS) system for handheld computers. AutoDVS distinguishes common, coarse-grained, program behavior and couples forecasting techniques to make accurate predictions of future behavior. AutoDVS estimates periods of user interactivity, user non-interactivity (think time), and computation, per-program and system wide to ensure quality of service while reducing energy consumption.

Pouwelse et al [102] look at application-directed DVFS with the observation that it is difficult to achieve good results using only statistics from the OS level when applications show bursty (unpredictable) behavior. The authors here take

the approach that such applications must be made power-aware and specify their Average Execution Time (AET) and the deadline to the scheduler controlling the clock speed and processor voltage. They implement an Energy Priority Scheduling (EPS) algorithm supporting power aware applications - EPS orders tasks according to how tight their deadlines are and how often tasks overlap.

DVFS for Multi-core Processors is another interesting and challenging area [42]. One major design decision concerns whether to apply DVFS at the chip level or at the per-core level. Per-core DVFS is considered more expensive since it requires more than one power/clock domain per chip, and other circuitry to synchronize between the chips. Several researchers have explored the benefits of per-core versus per-chip DVFS for CMPs - one research reports that a per-core approach had 2.5 times better throughput than a per-chip approach. This is because the per-chip approach must scale down the entire chips (V, f) when even a single core is nearing overheating. With per-core control, only the core with a hot spot must scale (V, f) downward; other cores can maintain high speed unless they themselves encounter thermal problems. Managing power when running parallel/multi-threaded programs especially with the onset of heterogeneous many core architectures is yet another active area of research. [7] considers independent DVFS for each core, while a mixture of chip-wide DVFS and core allocation is considered in [75].

2.1.5 Operating System Power Management

The most widely implemented architecture for power management is the Advanced Configuration and Power Interface (ACPI) [62]. It has evolved together

with Intel ®architecture, the hardware platforms based on the most widely available commodity CPUs and related components. ACPI defines the following power states: seven S-states (S0-S6) at the whole-system; and four D-states (D0-D3) at the per-device level. The zero-numbered state (S0 for the system, or D0 for each device) indicates the running (or active) state, while the higher-numbered ones are non-running (inactive) states with successively lower power and correspondingly decreasing levels of availability (run-readiness), and increasing latency for entry and exit. ACPI also defines performance states, called P-states (P0-P15, allowing a maximum of 16 per device), which affect the component's operational performance while running. Both power states and performance states affect power consumption.

ACPI is a specification and therefore different systems can implement different aspects of the specification in varying degrees of granularity.

Almost all processors in the marketplace today support the concept of multiple processor idle states with varying amounts of power consumed in those idle states. Each such state will have an entry-exit latency associated with it. [133] introduced the now highly popular *cpu-idle* framework, an effort toward a generic processor idle management framework in Linux kernel. The framework introduced *idle drivers*, which implement processor specific mechanisms to enter/exit sleep states, and *idle governors*, which decide which sleep state a processor should enter based on different criteria (current CPU activity, next expected interrupt time, etc.). The goal of these frameworks was to have a clean interface for any processor hardware to make use of different processor idle levels and also provide abstraction between idle-drivers and idle-governors allowing independent development of

drivers and governors.

Polling within the operating system drivers or applications) is one of the biggest source of wakeups, but the use of high-frequency clock-tick interrupts as the basis for timer events, time-keeping, and thread-scheduling became noticeably problematic from a power consumption point of view. The tickless kernel project [127] in Linux introduced the notion of dynamic ticks - by reprogramming the per-CPU periodic timer interrupt to eliminate clock ticks during idle, the average amount of time that a CPU stays in its idle state after each idle state entry can be improved by a factor of 10 or more.

2.1.6 Memory Power Management

Memory power management is another area of active research, both from hardware and software points of view. Memory technology itself has been evolving with the recent emergence of triple-channel DIMMs/DDR3 SDRAMs, which have enabled different levels of power management (increasing/decreasing clock frequency, varying degrees of shallow/deep self refresh, etc. One technology which is finding its way into some systems is called “partial array self refresh” or PASR [98], [88]. On a PASR-enabled system, memory is divided into banks, each of which can be powered down independently. If any of those banks of memory is not needed, that memory (and its self-refresh mechanism) can be turned off; the result is a reduction in power use, but also the loss of any data stored in the affected banks. The amount of power actually saved is a bit unclear; estimates fall in the range of 5-15% of the total power used by the memory subsystem. Correspondingly, there is

work going on to enable support for PASR and related memory power management technologies from an operating system point of view.

Main memory, because of its relatively low power requirement (say, 2 watts per DIMM), seems at first glance to be of even less concern than disks. Its average size on contemporary hardware platforms, however, may be poised to grow more rapidly. With hardware system manufacturers' focus primarily on performance levels (to keep up with the corresponding performance demands of multicore CPUs), maintaining full CPU-to-memory bandwidth is critical. The consequence has been an evolution from single to dual-channel and now triple-channel DIMMs along with the corresponding DDR, DDR2, and DDR3 SDRAM technologies. Although reductions in the process feature size (DDR3 is now on 50-nanometer technology) have enabled clock frequency to go up and power per DIMM to go down somewhat, the desire for even greater performance via an increase in DIMMs per memory channel is still increasing the total power consumed by the memory system.

2.1.7 Compiler Techniques for Low Power

While hardware and operating system work to provide power management, applications could do their bit to aiding the runtime system by providing hints about their behavior allowing the runtime system to identify application behavior and idle times. Other mechanisms include power aware code transformations and optimizations which can provide reduction in power consumption. Compiler techniques for low power have been explored in different domains. [155] discusses some memory and compiler techniques for low power. Memory compaction to allow contiguous

access (and corresponding powering off memory) is discussed. The authors also propose the use of DVFS during phases of program when the compiler can predict non-CPU bound phases; a similar approach is proposed in [58]. [56] propose interesting code transformations specifically directed toward devices by looking at the effect of transforming explicit I/O-based applications to increase their device idle times. A solution is implemented by having applications specifically indicate their “run-length” which is a hint to the runtime system as to when the application expects to be idle. With this information, the operating system can apply more effective device control policies.

[69] discusses Dynamic Voltage and Frequency Scaling, Resource Hibernation and Remote Task Mapping optimization mechanisms. [87] proposes to use register labeling techniques during compilation to reduce energy consumption. Power aware instruction scheduling has been largely targeted to VLIW and Super Scalar processors. [134] develops a technique to combine static and dynamic scheduling to reduce power in super scalar processors. Here the authors propose a technique that uniquely combines the advantages of static scheduling and dynamic scheduling to reduce the energy consumed in modern superscalar processors with out-of-order issue logic. In this Hybrid-Scheduling paradigm, regions of the application containing large amounts of parallelism visible at compile-time completely bypass the dynamic scheduling logic and execute in a low power static mode.

Power-aware compilation and code-generation are recent topics of research. Some of these possibilities are explored in [78]. Automating power optimizations is a very hard problem in this domain.

2.1.8 Surveys

There are several exhaustive survey papers that provide in-depth analysis of different areas power management techniques. [109] specifically looks at survey of techniques for energy-efficient on-chip communication. [77] surveys power management only in high-performance systems. An exhaustive survey of dynamic power management techniques is provided in [14] and [15]. Power modeling, estimation and optimizations are covered in [81]. An overall survey of power management techniques is presented in [25]. Most of these papers are focused on specific domains, while we are trying to unify the overall space of power management across four related yet orthogonal axes of specification, modeling, techniques, and verification.

2.2 Low Power Verification

With extensive power management comes the problem of extremely complicated verification. It is essentially required that low power specific design elements be implemented correctly. Power management brings a host of new types of bugs which are not in the class of traditional functional bugs. Figure 2.5 gives a list of such newly introduced types of bugs in the system. Figure 2.6 gives a good list of the additional tasks on the Verification teams in such a low power methodology. It is quite clear that without a carefully planned rigorous methodology in place, verification teams will be hard-pressed to provide correctness guarantees.

In this section we survey some verification methods at different levels of de-

Power Management introduces new bug types which are not like the traditional functional bugs:

- Isolation and Level Shifting Bugs
- Control Sequencing bugs
- Retention scheme and control errors
- Retention selection errors
- Electrical Problems like memory corruption
- Power Sequencing and Voltage Scheduling errors
- Hardware-Software deadlock
- Power Gating collapse or dysfunction
- Power On Reset and bring up problems
- Thermal runaway or Overheating
- Cooling inefficiencies because of thermal hotspots
- Sub-optimal workload placement, resulting in thermal hotspots
- Repeated shutdown/power-on of servers
- Failures due to concurrent access from different IPs when full end-to-end power use cases are enabled
- Failures to meet the Quality of Services in terms of interrupt service time & device access time

Figure 2.5: New type of bugs introduced in a low power methodology [153].

sign abstraction. We do not survey assertion based methodologies in detail here, although given the kind of changes in the design due to power management techniques, assertion based verification is expected to play a very large role in any generic or unified methodology.

Power Management requires Verification teams to perform new functions:

- Verify connection, placement, type of isolation/level shifting
- Include new power intent files such as UPF
- Formulate test plan for architecture correctly
- Reach good power state coverage
- Verify design works in all states, transitions and sequences
- Address firmware control of power management
- Address power-on reset issues
- Address verification at each stage of design, not just RTL
 - Verify netlist at each handoff
 - Verify Power Switch and rail connectivity
- Migrate existing testbenches, assertions, monitors to be low power aware
- Think about exhaustive constrained random and asynchronous logic testing
- Verify Quality of Service (QoS) when servers are turned off
- Verify thermal conditions that can lead to thermal hotspots, which in turn can result in processor throttling
- Complete end to end verification of each power use cases of the device/server to uncover any potential concurrency issues in real time
- Verify multiple power use case scenarios, to uncover any limiting factors to peak current or thermal related factors

Figure 2.6: New verification tasks introduced in a low power methodology [153].

2.2.1 Pre-Silicon Verification

Low Power Transformations historically reside at gate-level (for eg., *clock-gating*) for multiple reasons. Chief among them is the fact that verification of the low power transformation at higher levels of design hierarchy is very expensive. At the gate-level, low power transformations usually do not introduce new state. Hence formal

combinational equivalence checking between the design before the transformation and after will ensure the functionality preserving property of the transformation. However, high-level (RTL or architectural or system level) low power transformations will change the gate level design substantially more than low-level transformations. Figure 2.7 details this phenomenon.

Let oRTL be the original RTL design and oGates the gate-level design of the original RTL (obtained by synthesis). Assume the functionality of oRTL is checked by traditional validation methods. Now, we contrast the following two scenarios:

- [1] Suppose we have a low-level (gate-level) low power transformation on oGates to give us a new gate-level design llpGates . Since this transformation at the gate-level typically does not introduce new state, we can use formal Boolean combinational equivalence checking to ensure that the functionality of the design is not altered by the transformation. The cost of this equivalence checking is minimal, since the same methodology and tool-flow is used to verify other gate-level transformations for circuit optimization, reducing timing, reducing area, etc.
- [2] Suppose we have a high-level (RTL) low power transformation on oRTL to give us a new RTL design hlpRTL . Let the gate-level design (after synthesis) of this new RTL be hlpGates . Since the transformation is at a higher level of design hierarchy, it can potentially introduce extra state into the design. Therefore, using formal Boolean combinational equivalence checking to show equivalence between oGates and hlpGates is not an option. The

only verification option in this scenario is using traditional validation methods on the post-transformation RTL design hlpRTL.

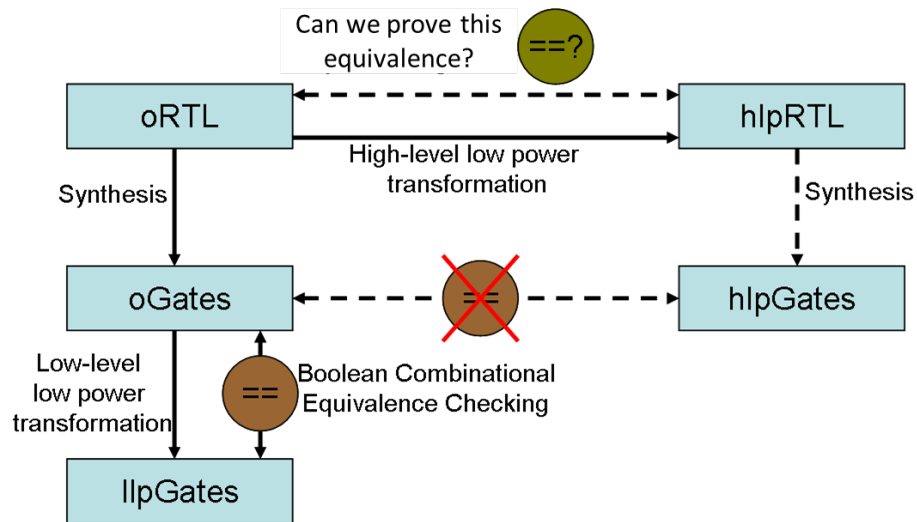


Figure 2.7: Verification cost of low power transformations.

In the second scenario above, the tradeoff between power reduction and verification cost is not always clear which leads the architects and designers being cautious and conservative in their implementation. Often aggressive high-level low power transformations are not implemented in the design because the verification cost (typically 70%+ of the total time cost) is too high. An automatic proof of the correctness of the high-level transformations would be extremely desirable in order to implement aggressive power reduction schemas.

Formal verification, especially equivalence checking, has achieved considerable success in the context of low power verification. These techniques are generic techniques which can also be effectively used for low power design verification.

Combinational equivalence checking checks two acyclic, gate-level circuits. Combinational equivalence checkers can also be used to check equivalence of two sequential designs, provided the state encodings of the two designs are the same. Although this technique has widespread use in many commercial tools, the real challenge of sequential verification is in verifying two designs with different state encodings. Sequential satisfiability engines [79], [103] and sequential ATPG engines [59], [3] solve this problem to a large extent by unrolling the circuit until a given time frame. Considerable research has been done to find compare points for latch mapping [23], [5], [135]. However, these techniques operate at the gate level, where they reason in the Boolean domain. More sequential simplification and equivalence algorithms can be found in [86], [64], [92], [136]. An overview of various methods are given in [71].

Fewer attempts have been made to apply sequential equivalence checking to the behavioral RTL descriptions of designs. In [122] a methodology for checking the combinational equivalence between C and RTL is described. The C source code is converted to a Hardware Description Language (HDL) and commercial RTL to RTL equivalence checkers are used thereafter. The C code is very similar to the RTL, in order for the translation to be achieved, which might not be a scalable solution.

Clarke and Kroenig [34],[70] proposed a solution with CBMC, a C-based bounded model checking engine that takes a C program and a Verilog implementation. The two programs are unwound together, and converted into a Boolean satisfiability checking problem. The Verilog code is converted to Boolean formulas by a synthesis-like procedure, and an innovative technique is described to convert the C-code into

Boolean formulas, including pointers and nested loops. However, the capacity of CBMC is limited by space and time considerations. This is because the reasoning done by this tool is entirely in the Boolean domain. On the other hand, our technique reasons at the system and register transfer level, splitting the equivalence checking problem into smaller problems that can be handled by the lower level engines. This static analysis of the source code, before running the problem through Boolean level engines, is the principal contribution of our technique.

Another approach to equivalence checking between C descriptions, that could be extensible to C vs RTL descriptions, is described in [84]. This approach detects and extracts the textual differences in the two target programs, and then performs a dependence analysis using program slicing, to check for the actual differences in the two programs. It then symbolically simulates this difference and reports the equivalence checking results. This technique, however, is most effective when the two target programs being compared are very similar to each other, in function as well as structure. Since this process uses syntactic information entirely, the similarity of the target descriptions is very essential to its application. Our technique does a semantic comparison of the two target programs, with respect to their functionality, and is therefore wider in its scope.

A few commercial tool vendors [1] also aim at solving the sequential equivalence checking problem between system level model (SLM) and RTL. However, this area still presents a major opportunity for further research.

In [142] we present a technique for RTL to RTL equivalence checking of complex combinational circuits including multipliers. We have extended the same tech-

nique to sequential equivalence checking, and addressed the problem in the realm of SLM vs RTL in [138],[141].

The technique involves the efficient decomposition of the equivalence checking problem, in order to make it more tractable. The authors present an automatic technique to compute high level *sequential compare points*, to compare variables of interest (observables) in the candidate design descriptions. The compare points are defined as co-ordinates on the space-time axis of the design, denoted by their relative position with respect to the time domain (clock cycles), and their position in the space domain (data variables). This aligns with the sequential behavior of the designs being compared, and provides an easy, intuitive abstraction of the equivalence checking problem space. The proof starts with the two design state machines at the same initial (or reset) state, and steps the machines through every cycle, until you reach a sequential compare point.

At the sequential compare points, one constructs symbolic expressions for the observables that encapsulate the sequential behavior of the designs, until the cycle of comparison. At each sequential compare point, the equivalence of the two state machines is proved using a lower (Boolean) level engine, which in this work, is a Boolean satisfiability (SAT) solver. The principal gain of this technique is that it leverages the expressiveness and information available at the register transfer and system levels. Although significant amount of research has been done on compare points for gate level equivalence checkers, these algorithms and heuristics are limited by their domain. On the other hand, since we operate at the higher, source code level, the sequential compare points are more intuitive and easier to detect.

Also, they capture the notion of design progression through time, which is useful in meaningful decomposition of the equivalence checking state space.

Most of the equivalence checking methods described so far in this section are generic methods, which can also be applied to the context of low power design. We next look at a methodology that has been carefully fine-tuned explicitly for the context of low power design.

Given the nature of power management, more verification will be focused on RTL and higher levels of abstraction. [117] describe methods to verify RTL power gating through transaction level models. The rest of this section will explore verification strategies at the Platform level and for servers and data centers.

2.2.2 Platform Verification

Platform level validation of power and/or thermals is a very hard and complex problem. Given the complexity of today's systems - whether they are many core systems, or SoCs for phones/tablets, validating all aspects of power management and thermals is very hard. While this is interesting (and hard) from a purely academic point of view, industrial designs rely very heavily toward ensuring that once the silicon arrives, power management can be validated as soon as possible, and thermal solutions can be built accurately for the specific form factor(s) in consideration.

In order to accomplish this, typically companies use complex and costly FPGAs to emulate the entire chip/SoC RTL, and build platform level validation/verification tools that can include the ability to boot entire operating system on such FPGA

complexes. SoftSDV from Intel [131] is a pre silicon functional verification tool. However, this does not allow for detailed power estimation/modeling/verification. Several such internal, proprietary (and costly) validation systems are used typically for validation power management features.

Thermal validation at platform level is an equally hard problem. Typically form factor devices are built early on in the platform bring up stage, and these form factors are analyzed for heat flows in heat chambers. Based on the thermal hot spots, appropriate thermal control algorithms are fine tuned. This is a costly, but accurate way of ensuring that thermal management on the devices are validated effectively.

2.3 Conclusions

We have surveyed many optimizations for lower power dissipation at various levels of hardware circuit abstractions. Most of the power optimizations are at the gate and transistor level. Techniques which try to analyze and optimize power consumption at a higher level of abstraction, *viz.* architecture or RTL level, translate the given high level architecture/RTL description to gates (netlists) and then use reasonably accurate low-level power analysis engines. This method is infeasible if we want to evaluate a large number of design choices.

Most initial work in the area of power optimizations at the RTL and architecture levels suffer from modeling power dissipation for very specific hardware structures (*eg.*, caches, branch predictors, etc.). It would be useful to be able to introduce optimizations for lower power dissipation for any arbitrary hardware circuit. In this

context, one might consider making use of the program structure or the dataflow information (statically or dynamically) available at the architecture and RTL levels.

There have been a variety of techniques proposed and implemented for hardware level power management. However, the same cannot be said of software; software is limited to use the ACPI interfaces exposed by hardware. Further, many hardware techniques are not even visible to the OS.

The techniques used at each level of design abstraction and hierarchy are quite different from each other. There is also no single scalable power management specification mechanism which is applicable from RTL to operating systems. What we need is a generalizable specification that encompasses silicon and software allowing standard interfaces to be exposed at each level, while simultaneously providing for esoteric techniques and abstracting such techniques from those used at other layers. In such a generalized power management methodology, there will be an intelligent optimal way to choose between techniques at different levels of abstraction, to maximize the global power intent.

Chapter 3

Instruction-driven Slicing: Automatic Insertion of Low Power RTL Annotations

3.1 Introduction

The tradeoff between power reduction transformation and its verification cost is not always clear which leads the architects and designers to be cautious and conservative in their implementation of such design transformations. Often aggressive low power transformations are not implemented in the design because the verification cost (typically more than 70% of the total time cost) is high enough to adversely impact the total time-to-market.

Utilizing the program structure or the dataflow information available at the architectural and RTL levels can lead to many interesting and complicated low power transformations [97], [20], [29], [145], [144]. Yet most power transformations in today's designs are at the gate-level [104], [4], [129], [90]. The primary reason for this is the hardness of the verification problem. If the low power transformations are at the gate level, then equivalence checking methods [85] can be used to automatically prove the correctness of the transformation. Although there are a few sequential equivalence checking algorithms [138], [59], they are not widespread in the hardware industry. In a typical industry scenario, an RTL or architectural low

power transformation implies a full cost of dynamic validation, which can extend to many months, and require a lot of resources.

In this chapter, we propose a new technique for low power microprocessor design. Our technique can be thought of as a fine-grained clock gating scheme implemented at the RTL or the architectural level which utilizes the program structure of the model. Our algorithm automatically identifies fine-grained blocks of circuit which are not used on any given cycle during the execution of a particular instruction, and shuts them down. This scheme of slicing the circuit based on the instruction being executed is termed *instruction-driven slicing*. The algorithm automatically inserts annotations in the RTL to implement the shut-down circuitry for the unused blocks during the execution of any instruction. We also propose a methodology to prove that these automatically inserted annotations preserve the functionality of the original processor RTL.

All prior approaches toward analyzing and optimizing RTL and architectural models for lower power dissipation suffer from modeling the power dissipation for very specific hardware structures. Besides, they do not make use of the program structure or the dataflow information (statically or dynamically) available at the architecture and RT-levels.

3.1.1 Contributions of this work

We propose a new technique for low power microprocessor design. For any given instruction, when it is decoded, we have sufficient information to recognize what resources are required to execute that instruction. We introduce the concept of

an *instruction-driven slice*. An *instruction-driven slice* of a microprocessor design, is all the relevant circuitry of the design (a slice of the RTL program) required to take the life cycle of the instruction to completion (execute, writeback *etc.*).

The primary idea is: given a microprocessor design, depending on the instruction, we identify the instruction-driven slice, and shut off the rest of the circuitry. This might include gating out fine-grained parts of various processor blocks depending on the instruction, or gating out the floating point execution units during integer ALU execution, or gating out the memory units during ALU operation, or turning off certain FSMs in various control blocks because the instruction-driven slice provides exact value constraints on their inputs, and so on.

One way of implementing this idea is to add to the RTL code, the instruction-driven slice identification and turning off operations as annotations. At the netlist level, we do not have sufficient information to identify an instruction-driven slice. The advantage of annotating the RTL is that the circuitry relevant to perform these tasks is automatically generated by the synthesis tool along with the rest of the netlist.

We have implemented this on OR1200, a Verilog implementation of the OpenRISC [38] architecture. This technique and the same annotations can also be inserted at the architectural level. We have implemented an architectural model of the same OR1200 processor and simulated it with and without the annotations in the SimpleScalar tool set [24] and estimated the power dissipation using SimWattch [30].

We have proved the correctness of these annotations on the OR1200 processor. We use the ACL2 [67] theorem prover to show that the RTL model before instruction-driven slicing and the RTL model after instruction-driven slicing are precisely equivalent in terms of their functionality.

We have also implemented instruction-driven slicing in PUMA [105], a PowerPC fixpoint core. As with the in-order OR1200 case, the technique and the same annotations have been applied at the architectural and RT levels. We have also implemented an architectural model of the same PUMA processor and as with the OR1200 model, simulated it with and without the annotations and estimated the power consumption.

An important aspect of optimizing power at the RT-level is to first develop a framework for analyzing the power dissipation at an architectural level. The traditional way has been to translate the given high level architecture description to gates (netlists) and then use reasonably accurate low-level power analysis engines. This method is infeasible if we want to evaluate a large number of design choices. Brooks *et. al* in [20] present a framework for analyzing power dissipation at the architecture level. We use this framework to estimate the efficacy of our optimizations at the architectural level.

Most initial work in optimization for low power at the RT-level is focused on power analysis and reduction in caches [111],[157]. This is chiefly because embedded microprocessors, historically the reason for low power design, used to devote nearly 40% of their power budget to caches. Besides caches are regular structures and are more easily modeled than other circuits. Other attempts include power re-

duction by reducing needless speculation in branch predictors [29], gated clocks in integer ALUs [19], *etc.*

Inserting annotations using instruction-driven slicing is explained in detail with examples in Section 3.2. Our overall methodology and integrated tool flow is detailed in Section 3.3. We also give an algorithm for adding the annotations automatically. Section 3.4 explains instruction-driven slicing in the RTL as well as the architectural models of OR1200, a pipelined implementation of OpenRISC. Results from running SPECINT2000 benchmarks on these models and some comparison metrics are also presented. Section 3.5 explains instruction-driven slicing in the RTL as well as the architectural models of PUMA, a PowerPC fixpoint core. Again, results from running SPECINT2000 benchmarks on these models and the same comparison metrics as for the in-order pipelined machine are presented. Some conclusions and future directions are discussed in Section 3.6.

3.2 Instruction-Driven Slicing

Program slicing has been well studied in the context of software engineering [80], programming languages [128], and more recently, in the context of slicing hardware description languages [35],[36],[143]. We define a new notion of program slicing for microprocessor descriptions, *viz.*, slicing based on the instruction which is being executed. Given a microprocessor design and an instruction, *instruction-driven slicing* identifies a slice of the abstract program graph of the microprocessor design corresponding to all the relevant circuitry needed to execute that instruction.

The cone of influence of a variable in a program is the set of all program state-

```

Algorithm instruction-driven-slicing (input: vRTL, insts; output: aRTL).
[1] Parse vRTL to obtain the ASPG (Abstract Syntax Program Graph).
[2] For each instruction i in insts repeat
    Begin loop
[3]   Slice the ASPG for instruction i
[4]   Traverse the ASPG (finish when done traversing all nodes)
[5]   Add annotation variables if such a block is found
[6]   If a particular flop is already gated by a previous annotation, then
        add the current annotation as an additional signal
[7]   Return the annotated ASPG
    End loop
[8] Generate Verilog code for the annotated ASPG (aRTL).
End.

```

Figure 3.1: Overview of the Instruction-driven Slicing Algorithm for RTL.

ments which depend on the variable. Any hardware design written in Verilog (at the RT-level) can be thought of as a control flow graph, henceforth referred to as program graph. Traditional program slicing on a variable can be thought of as reducing the program to retain only the slice of the program graph which is within the cone of influence of the variable. Instruction-driven slicing on the other hand, identifies the slice of the program graph which is within the cone of influence of an instruction. The cone of influence of an instruction is the slice of the microprocessor circuitry required to execute that instruction from start to finish. In terms of the RTL program graph, it is a slice of the program which is in the cone of influence of the semantics of an instruction. More specifically, it is the the union of the cone of influence of each of the variables affected by the instruction.

There are two parts to instruction-driven slicing. First, we need to identify the instruction-driven slice. Next, we need to isolate the rest of the circuit by identifying the flops governing the rest of the circuit and gating them out. Since we are slicing based on an instruction or a type of instruction (for example, an ALU instruction, or an LSU instruction, *etc.*), we can obtain a slice both at the RTL and the architectural level models.

Turning off sub-circuits is a well-researched topic [90], [129], and in addition to instruction-driven slicing, we can implement more sophisticated algorithms to determine the sub-circuits to be turned off and the logic required to perform the disabling. The innovation is to automatically identify the sub-circuits in the context of the execution of a particular instruction by leveraging available high level information about instructions and functional units at the RT-level, which is not available to the traditional transistor level optimizing tools. In fact, it turns out that our algorithm to automatically identify an instruction-driven slice introduces much more fine grained gated clocks than prior art automatic methods. We report these findings in Section 3.4.

Instruction-driven slicing at the architectural level is carried out exactly the same way as at the RT-level. The overall structure available at the architectural level is the same as the RTL model. The key difference is that the architectural model is more abstract than the RTL model. This in turn means that the clock gating due to instruction-driven slicing is more coarse-grained in the architectural model than in the RTL model.

3.3 Our Technique

3.3.1 Instruction-Driven Slicing Algorithm

We give an algorithm for automatically identifying instruction-driven slices, given a set of instructions and a microprocessor design. The instruction-driven slicing algorithm for RTL models is given in Figure 3.1.

The inputs to the algorithm are an RTL model (vRTL) of the microprocessor, and a set of instructions `insts` which will be executed on that model (given by the ISA of the microprocessor). First, we parse the vRTL model to generate an abstract graph of the program, called the **Abstract Syntax Program Graph (ASPG)**. The nodes of an ASPG are the data computing/modifying statements of the design, whereas the edges of the ASPG define the control flow of the RTL program. We have modified the Verilog parser from `vl2mv` code distributed with VIS-2.0 [126] to generate our ASPGs.

Now, we traverse the ASPG for each instruction and slice the ASPG. The graph traversal algorithm is a two-pass algorithm. In the first pass we identify variables affected by the instruction driving the slicing and the cone of influence of those variables. Along with this, we compute the condition predicates that are true for every pipeline stage. In the second pass, we identify parts of the ASPG governed by flops* which are outside the identified cone of the first pass. These parts of the ASPG are gated out†. If there is already gating logic on any of these flops, then the

*We use the term *flop* loosely to mean a single-bit storage element with an enable signal.

†We implement the gating out, not by preventing clock from switching, but setting and unsetting the enable on every flop. We assume that all flops have an enable signal. Also, because of such

algorithm adds to the existing logic in an optimized fashion.

Lastly, we reverse the process of the parser, and generate Verilog code for this annotated ASPG (aRTL).

The time complexity of our algorithm is linear in the size of the program graph. A point of note is that the algorithm may not be able to identify every flop outside the slice. The computing the cone of influence part of our algorithm is based on prior algorithms with guaranteed correctness [143]. The correctness result guarantees that the generated slice is always an over-approximation, *i.e.*, the annotation insertion is guaranteed to be a functionality preserving transformation.

We have implemented our instruction-driven slicing algorithm on the ASPGs. The advantage of this is that without any loss of generality, we can apply the algorithm on any ASPG, irrespective of whether the ASPG was generated from Verilog RTL or from SimpleScalar architectural C models. The algorithm for instruction-driven slicing on the architectural model remains the same, except for parsing the model into ASPGs and generating annotated models from ASPGs. We have implemented our instruction-driven slicing algorithm in C.

The advantage of decoupling the algorithm from the model is that the algorithm can now be treated as a transform engine, which is a part of the tool chain.

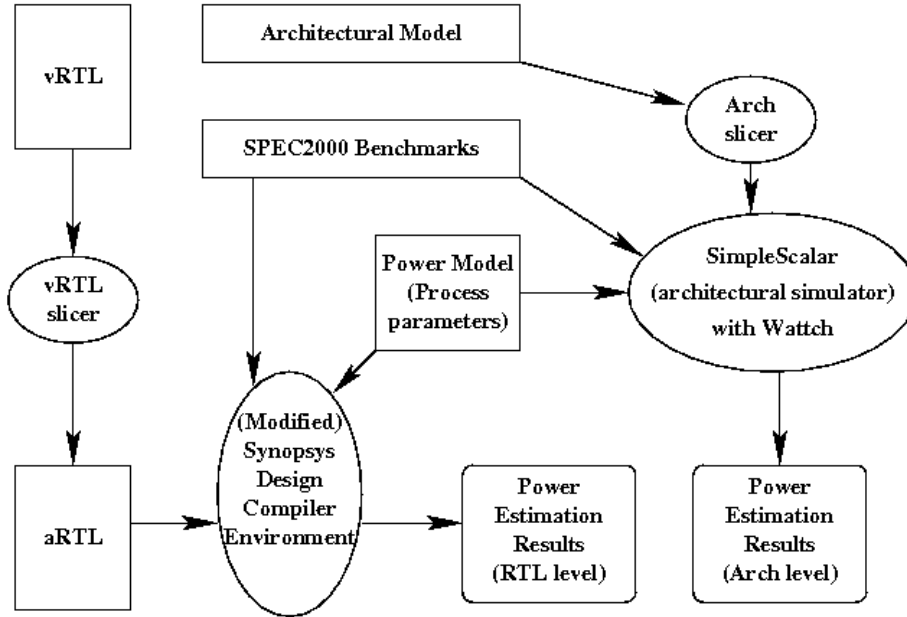


Figure 3.2: Incorporating *Instruction-driven slicing* into the design flow.

3.3.2 Methodology

We have implemented a methodology to incorporate instruction-driven slicing into the design flow. Figure 3.2 describes the overall implementation strategy of our technique. We have designed the tool-flow in order to incorporate instruction-driven slicing as a part of the traditional design flow.

In order to demonstrate our technique we have built the following tool-chain. We start with the Verilog RTL (ν RTL) and the architectural models. The RTL code

a gating out mechanism, there is no added clock skew, and at the same time, there is no dynamic power saved in the clock distribution network. The extra power consumed by this is accounted in the experiments

is annotated with instruction-driven slicing annotations to obtain the aRTL, by the previously described algorithm (Figure 3.1). The aRTL code, process parameters for power estimation, as well as the benchmark SPECINT2000 files are fed to the Synopsys Design Compiler Environment. There is a lot of work done on what regions of the SPEC benchmarks are to be used for simulation [100], [53]. The SPEC benchmarks we run are forwarded to 80 Million instructions, before we collect power dissipation numbers. We have modified and setup the Synopsys Design Compiler Environment as an integrated tool which can take SPEC benchmarks and RTL code, synthesize the RTL code and determine the power consumption due to switching activity.

Along a parallel path, we start with the architectural model of the design. Our model is written compatible with the SimpleScalar Tool Set [24]. The model is annotated with instruction-driven slicing annotations and fed as input to the SimpleScalar with Wattch environment. Wattch is an architecture level power estimator [20], [30]. We also modified the `power.h` file in this environment to reflect the same process parameters as used for the RTL power estimation.

Our aim in building this parallel power estimating setup at two levels of design hierarchy is two-fold. First, we wish to show that the dataflow and structure information available at these levels can be usefully exploited to optimize the design for lower power consumption. Second, our technique of automatically adding annotations is scalable to many levels of design hierarchy. The only caveat is that since the architecture model is more abstract than the RTL model, the slicing induced clock gating is coarser for the architecture model than the RTL model.

We have used this tool-chain to test our technique on OR1200, an in-order pipelined OpenRISC processor and PUMA, a Power PC fixed point unit out-of-order super-scalar core.

3.4 OR1200 - a Pipelined OpenRISC Implementation

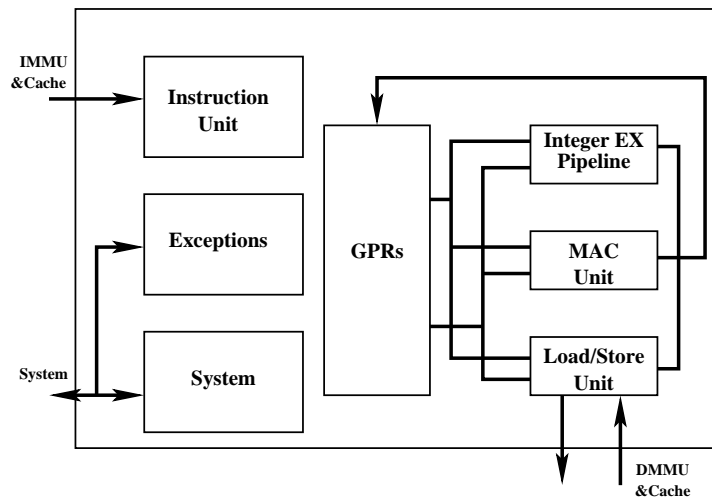


Figure 3.3: OR1200 Processor Block Diagram.

In order to demonstrate the efficacy of our technique, we have chosen a state-of-the-art in-order microprocessor as our example. OR1200 is a pipelined microprocessor implementing the OpenRISC instruction set architecture. We have implemented the architectural model of OR1200 compatible with SimpleScalar (`sim-or1200`). In the rest of this section, we first give a description of the processor itself, and then give our results from running our technique on these models.

3.4.1 OR1200

We use the OR1200, a publicly available processor for our experiments. The specification manual of the OR1200 is at [38] and the source code of its implementation in Verilog RTL can be obtained from [95]. The OR1200 is a 32-bit scalar RISC with Harvard microarchitecture, 4 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities. OR1200 is intended for embedded, portable and networking applications.

Figure 3.3 shows the block diagram of the CPU of the OR1200 processor. The instruction unit implements the basic instruction pipeline, fetches instructions from the memory subsystem, dispatches them to available execution units, and maintains a state history to ensure a precise exception model and that operations finish in order. The execution unit must discern whether source data is available and has to ensure that no other instruction is targeting the same destination register. OpenRISC 1200 implements 32 general-purpose 32-bit registers. The load/store unit (LSU) transfers all data between the general purpose registers and the CPU's internal bus. We have implemented all single-bit storage elements with an enable signal.

In this experiment we use TSMC CLO18G [91], a $0.18\mu\text{m}$ generic process technology to estimate the power dissipation.

3.4.2 Results for OR1200-RTL

The RTL annotations were automatically generated and inserted in the OR1200 RTL in this experiment. An example of this is shown in Figure 3.4. The results are shown in Figure 3.5. It is important to note that these numbers are on models

```

// Instruction selection in load/store unit

always @(posedge clk or posedge rst) begin
  case (id_insn[31:26])
    `OR1200_OR32_SB: lsu_op <= #1 `OR1200LSUOP_SB;
    `OR1200_OR32_SW: lsu_op <= #1 `OR1200LSUOP_SW;
    `OR1200_OR32_LBZ: lsu_op <= #1 `OR1200LSUOP_LBZ;
    `OR1200_OR32_LWZ: lsu_op <= #1 `OR1200LSUOP_LWZ;
    default: begin lsu_op <= #1 `OR1200LSUOP_NOP;
    endcase
  end
end

```

(a) Verilog RTL code for the always block assigning the lsu_op before instruction-driven slicing transformation.

```

// Instruction selection in load/store unit sliced on
// instruction l.addc

always @(posedge clk or posedge rst) begin
  if (iADDC_id)
    lsu_op <= #1 `OR1200LSUOP_NOP;
  else
    case (id_insn[31:26])
      `OR1200_OR32_SB: lsu_op <= #1 `OR1200LSUOP_SB;
      `OR1200_OR32_SW: lsu_op <= #1 `OR1200LSUOP_SW;
      `OR1200_OR32_LBZ: lsu_op <= #1 `OR1200LSUOP_LBZ;
      `OR1200_OR32_LWZ: lsu_op <= #1 `OR1200LSUOP_LWZ;
      default: begin lsu_op <= #1 `OR1200LSUOP_NOP;
      endcase
    end
end

```

(b) Transformed Verilog RTL code after applying instruction-driven slicing on instruction l.addc.

Figure 3.4: Instruction-driven slicing example.

SPECINT2000 Benchmarks	Unsliced Power Dissipation	1-Sliced Power Dissipation	%-age Gain
gcc	1.89 mW	1.72 mW	8.99%
gzip	1.44 mW	1.38 mW	4.17%
parser	2.12 mW	1.84 mW	13.21%
vortex	2.33 mW	2.02 mW	13.30%
Average	1.95 mW	1.74 mW	10.54%

SPECINT2000 Benchmarks	Unsliced Power Dissipation	4-Sliced Power Dissipation	%-age Gain
gcc	1.89 mW	1.69 mW	10.58%
gzip	1.44 mW	1.31 mW	9.03%
parser	2.12 mW	1.82 mW	14.15%
vortex	2.33 mW	1.98 mW	15.02%
Average	1.95 mW	1.70 mW	12.60%

SPECINT2000 Benchmarks	Unsliced Power Dissipation	10-Sliced Power Dissipation	%-age Gain
gcc	1.89 mW	1.53 mW	19.05%
gzip	1.44 mW	1.27 mW	11.81%
parser	2.12 mW	1.63 mW	23.11%
vortex	2.33 mW	1.81 mW	22.32%
Average	1.95 mW	1.56 mW	19.79%

Figure 3.5: OR1200-RTL Power dissipation results after slicing on 1, 4 and 10 instructions. These results are for SPECINT2000 benchmarks **with** Synopsys clock gating

of the processor, and were not originally designed to be power-efficient. The key result therefore, is the percentage reduction in power dissipation. The results are summarized in Figure 3.7(a). In the best case, we see a 25% reduction of dynamic

SPECINT2000 Benchmarks	Unsliced No Synopsys clockgating	1-Sliced No Synopsys clockgating	%-age Gain
gcc	1.94 mW	1.83 mW	5.67%
gzip	1.73 mW	1.57 mW	9.25%
parser	2.47 mW	2.17 mW	12.15%
vortex	2.51 mW	2.14 mW	14.74%
Average	2.16 mW	1.93 mW	10.87%

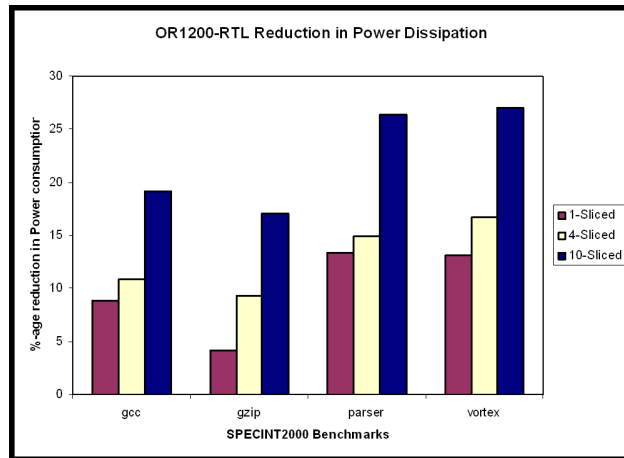
SPECINT2000 Benchmarks	Unsliced No Synopsys clockgating	4-Sliced No Synopsys clockgating	%-age Gain
gcc	1.94 mW	1.72 mW	11.34%
gzip	1.73 mW	1.51 mW	12.72%
parser	2.47 mW	2.14 mW	13.36%
vortex	2.51 mW	2.18 mW	13.15%
Average	2.16 mW	1.89 mW	12.72%

SPECINT2000 Benchmarks	Unsliced No Synopsys clockgating	10-Sliced No Synopsys clockgating	%-age Gain
gcc	1.94 mW	1.65 mW	14.95%
gzip	1.73 mW	1.51 mW	12.72%
parser	2.47 mW	1.89 mW	23.48%
vortex	2.51 mW	1.88 mW	25.10%
Average	2.16 mW	1.73 mW	19.88%

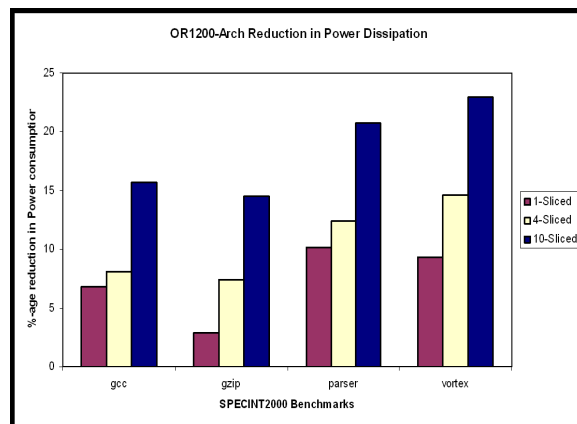
Figure 3.6: OR1200-RTL Power dissipation results after slicing on 1, 4 and 10 instructions. These results are for SPECINT2000 benchmarks **without** Synopsys clock gating

power dissipation and 20% on an average.

Our power estimation tool (Synopsys power compiler) also automatically gates



(a) OR1200-RTL



(b) OR1200-Arch

Figure 3.7: OR1200 reduction in power dissipation for SPECINT2000 benchmarks.

the clock. We also show results of turning off the default clock-gating provided by Synopsys Power Compiler in Figure 3.6. Instruction-driven slicing power gains are very similar even when the auto-power reduction mechanism of the measuring tool is turned off. Primarily, we have less or no overlapping gating because our flop

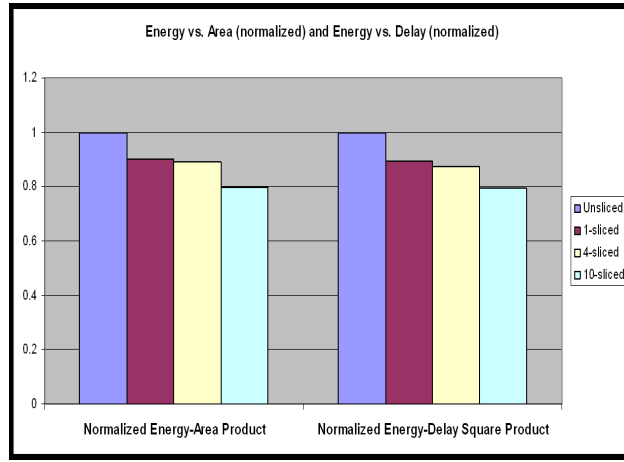
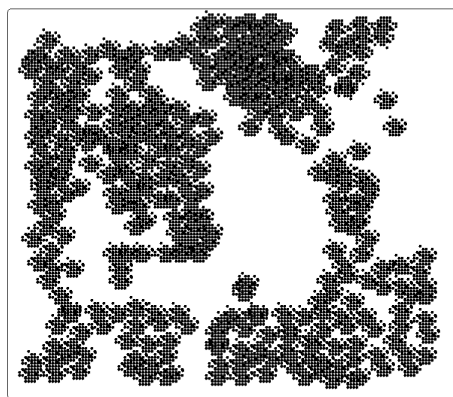


Figure 3.8: OR1200-RTL Power reduction compared to increase in area and delay due to slicing. The first set of comparisons depicts the normalized $Energy - Area$ product. The second set depicts the normalized $Energy - Delay^2$ product.

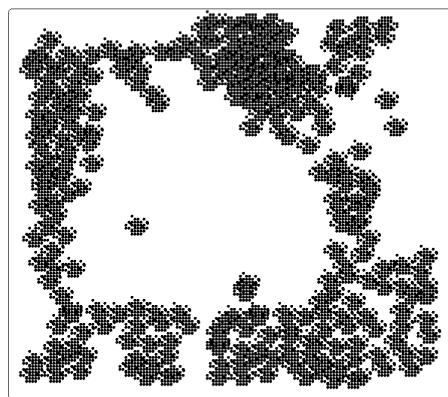
disable logic is extremely fine grained and is not on the clock distribution network. In our experiment, both the unsliced and the sliced RTL go through the same additional clock gating and hence the percentage reduction we obtain is in addition to what was automatically added by the synthesis tool.

Figure 3.8 depicts the power-vs-timing and power-vs-area tradeoff. The normalized $Energy - Area$ product decreases consistently with increased slicing. This means that as far as increase in area is considered because of additional logic, it is not a problem since we are gaining substantially in terms of reduced power. The same result shows up from the $Energy - Delay^2$ product as well. [82] introduced $Energy - Delay^2$ product as an efficient measure of energy-vs-delay tradeoff since it represents a voltage independent metric. Therefore, independent of device supply voltage, gains because of lower power dissipation consistently offset the increased

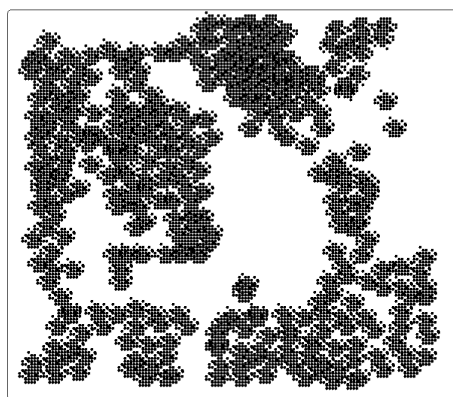
area and delay. Also, in certain timing critical blocks, our algorithm can be tuned to slice more coarsely to meet the timing requirements of that block.



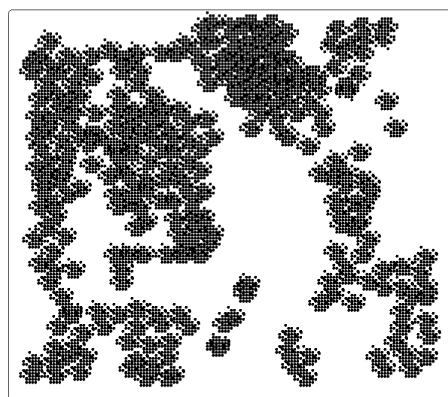
(a) Un sliced flop distribution
(3287 flops enabled)



(b) After slicing on `l.add`
(1874 flops enabled)



(c) Un sliced flop distribution
(3287 flops enabled)



(d) After slicing on `l.lw`
(2456 flops enabled)

Figure 3.9: Flop distribution effect of instruction-driven slicing on `l.add` and `l.lw` in the OR1200 RTL.

We also synthesized our design and ran it through a place-and-route tool [26],

both before and after the slicing. The design contained 3287 flops before slicing. In the unsliced version, all 3287 enables are treated as on as shown in Figure 3.9(a). This is a visualization of flop positions after place-and-route. After instruction-driven slicing on `l.add`, 1413 flops are disabled during the course of execution of the `l.add`. Figure 3.9(b) shows the flop distribution after slicing on the `l.add` instruction. The parts of the chip that are lit are all the enables on the flops which are on during the execution of the `l.add`. In the unsliced layout, the entire chip is on, as opposed to the sliced layout where we can clearly see the fine-grained clock gating induced by our algorithm. Figure 3.9(d) shows the same comparison for a load (`l.lw`) instruction (831 flops are disabled in this case). The flop distribution in figures Figure 3.9(c), Figure 3.9(b), and Figure 3.9(d) is based on preliminary floor plan estimate, whereas, the number of enables in each case is accurate.

The inserted annotations introduce additional flops into the design. For the OR1200 RTL we found that the number of additional flops was less than 1% of the total number of flops. On the same count, the additional switching power due to the additional logic is also less than 1% of the total power dissipated. The additional logic will also cause increased leakage power. Since we have no model to measure the static power dissipation, we do not have a measure of this. However, since the percentage of additional logic is so low compared to the overall power reduction, we do not expect this to be a problem.

SPECINT2000 Benchmarks	Unsliced Power Dissipation	1-Sliced Power Dissipation	%-age Gain
gcc	2.04 mW	1.90 mW	6.86%
gzip	1.67 mW	1.62 mW	2.99%
parser	2.32 mW	2.08 mW	10.34%
vortex	2.51 mW	2.28 mW	9.16%
Average	2.14 mW	1.97 mW	7.73%

SPECINT2000 Benchmarks	Unsliced Power Dissipation	4-Sliced Power Dissipation	%-age Gain
gcc	2.04 mW	1.87 mW	8.33%
gzip	1.67 mW	1.55 mW	7.19%
parser	2.32 mW	2.03 mW	12.50%
vortex	2.51 mW	2.14 mW	14.74%
Average	2.14 mW	1.90 mW	11.12%

SPECINT2000 Benchmarks	Unsliced Power Dissipation	10-Sliced Power Dissipation	%-age Gain
gcc	2.04 mW	1.72 mW	15.69%
gzip	1.67 mW	1.43 mW	14.37%
parser	2.32 mW	1.84 mW	20.69%
vortex	2.51 mW	1.94 mW	22.71%
Average	2.14 mW	1.73 mW	18.85%

Figure 3.10: OR1200-Arch Power dissipation results for SPECINT2000 benchmarks after slicing on 1, 4 and 10 instructions.

3.4.3 Results for OR1200-Arch

We ran the same benchmarks on our architectural model `sim-or1200`. The results are shown in Figure 3.10. `sim-or1200` absolute power dissipation estima-

tions were more than the RTL estimations. The percentage improvement observed was also less than in the RTL model. We believe this behavior is a direct correlation of how fine-grained the clock gating is. Also, since the architectural model is more abstract than the RTL model, it is natural to expect lesser gains on the architectural model. The results are summarized in Figure 3.7(b).

Although our technique is automatic, it is key to note the importance of the initial setup before the technique can be implemented. The instruction set architecture of the target microprocessor is the input to our technique. This needs to be described in complete detail if fine order differences between opcodes should be picked up by the technique. On the other hand, if the input opcode description can abstract (and therefore combine) certain opcodes, then the technique will give you a transformation that includes that abstraction.

Instruction-driven slicing is unique because it tries to enforce a semantics (the semantics of the instruction being executed, as given by the program graph) on the flops one is trying to shut-off. This does not preclude the use of netlist level power optimization techniques. To what extent a netlist level optimization is anticipated by our method is not clear. On the contrary, it is clear that our algorithm by virtue of operating at the RTL and architectural levels employs a host of optimizations which are not visible at the netlist level.

3.5 Instruction-Driven Slicing in the PowerPC Microprocessor

In order to demonstrate the versatility of our technique, we have chosen a very complicated, multiple instruction in flight, out-of-order super-scalar microprocessor

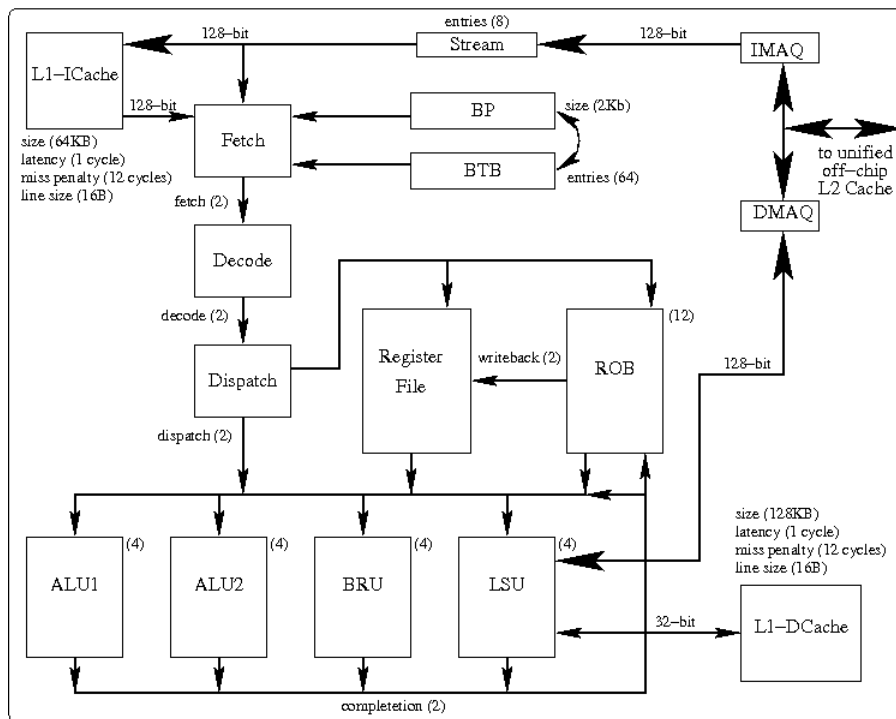


Figure 3.11: PUMA Fixed Point Unit Processor Block Diagram.

as our next example. PUMA is a fixed point unit PowerPC microprocessor. We have implemented the architectural model of PUMA compatible with SimpleScalar (`sim-puma`). In the rest of this section as before, we first give a description of the processor itself, and then give our results from running our technique on these models.

3.5.1 PUMA

PUMA [105] is a dual-issue, out-of-order super scalar fixed-point unit (FXU) based on the PowerPC instruction set. The processor implements a majority of the

SPECINT2000 Benchmarks	Unsliced Power Dissipation	1-Sliced Power Dissipation	%-age Gain
gcc	382.00 mW	365.00 mW	4.45%
gzip	370.00 mW	362.00 mW	2.16%
parser	412.00 mW	384.00 mW	6.80%
vortex	420.00 mW	392.00 mW	6.67%
Average	396.00 mW	375.75 mW	5.11%

SPECINT2000 Benchmarks	Unsliced Power Dissipation	4-Sliced Power Dissipation	%-age Gain
gcc	382.00 mW	361.00 mW	5.50%
gzip	370.00 mW	353.00 mW	4.59%
parser	412.00 mW	380.00 mW	7.77%
vortex	420.00 mW	384.00 mW	8.57%
Average	396.00 mW	369.50 mW	6.69%

SPECINT2000 Benchmarks	Unsliced Power Dissipation	10-Sliced Power Dissipation	%-age Gain
gcc	382.00 mW	347.00 mW	9.16%
gzip	370.00 mW	339.00 mW	8.38%
parser	412.00 mW	356.00 mW	13.59%
vortex	420.00 mW	363.00 mW	13.57%
Average	396.00 mW	351.25 mW	11.30%

Figure 3.12: PUMA-RTL Power dissipation results after slicing on 1, 4 and 10 instructions. These results are for SPECINT2000 benchmarks **with** Synopsys clock gating

SPECINT2000 Benchmarks	Unsliced No Synopsys clockgating	1-Sliced No Synopsys clockgating	%-age Gain
gcc	398.00 mW	375.00 mW	5.78%
gzip	389.00 mW	379.00 mW	2.57%
parser	441.00 mW	413.00 mW	6.35%
vortex	447.00 mW	417.00 mW	6.71%
Average	418.75 mW	396.00 mW	5.43%

SPECINT2000 Benchmarks	Unsliced No Synopsys clockgating	4-Sliced No Synopsys clockgating	%-age Gain
gcc	398.00 mW	371.00 mW	6.78%
gzip	389.00 mW	368.00 mW	5.40%
parser	441.00 mW	410.00 mW	7.03%
vortex	447.00 mW	413.00 mW	7.61%
Average	418.75 mW	390.50 mW	6.75%

SPECINT2000 Benchmarks	Unsliced No Synopsys clockgating	10-Sliced No Synopsys clockgating	%-age Gain
gcc	398.00 mW	357.00 mW	10.30%
gzip	389.00 mW	353.00 mW	9.25%
parser	441.00 mW	384.00 mW	12.93%
vortex	447.00 mW	389.00 mW	12.98%
Average	418.75 mW	370.75 mW	11.46%

Figure 3.13: PUMA-RTL Power dissipation results after slicing on 1, 4 and 10 instructions. These results are for SPECINT2000 benchmarks **without** Synopsys clock gating

integer instructions of the PowerPC ISA. The FXU was originally designed to find the optimal number of execution units, issue-width, branch prediction, *etc.* while reducing the total transistor count.

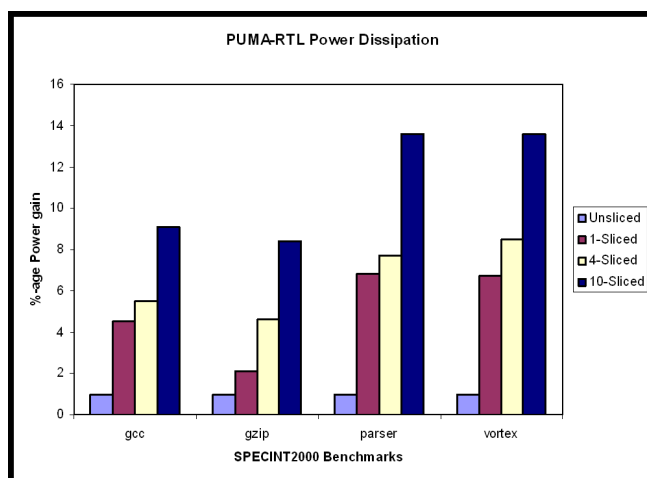
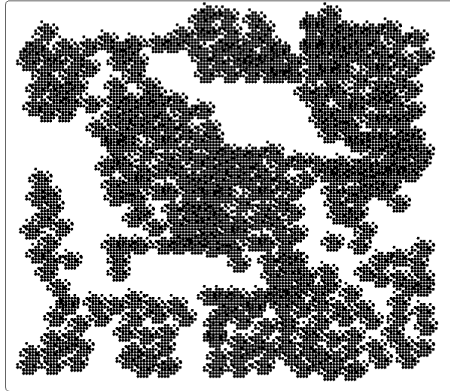


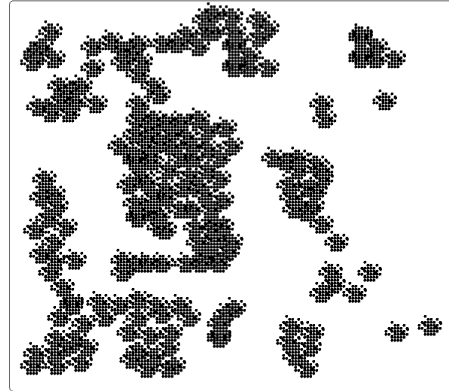
Figure 3.14: PUMA-RTL Power gains for SPECINT2000 benchmarks.

The processor block diagram is shown in Figure 3.11. The processor has a split level-1 cache and unified off-chip level-2 cache. The chip interfaces the level-2 cache through a 128-bit data bus and a 32-bit address bus. The address bus sends the requested load or store address to the second level memory management unit. Data is written on the 32-bit bus and read across the 128-bit bus. The cache line is 128-bits, so a full line is read for each second-level access. The data memory access queue (DMAQ) is the portal for the 128-bit bus and routes data and instructions to the respective on-chip cache. The instruction/control portion of the machine is composed of an instruction cache, fetch unit, decoder, and branch predictor. The decoded instructions are issued to the execution core through the dispatch unit and written back to the register file or re-order buffer. Four functional units have been implemented in the processor: one branch unit, two ALUs, and one load-store unit.

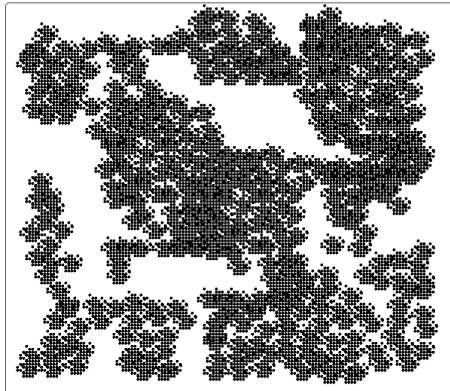
In this experiment (as with OR1200), we continue to use the same TSMC CLO18G



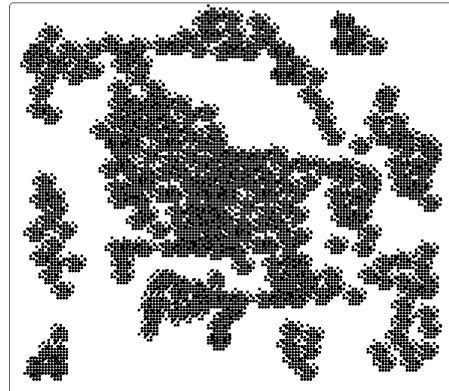
(a) Un sliced flop distribution
(6244 flops enabled)



(b) After slicing on 1 . add
(3431 flops enabled)



(c) Un sliced flop distribution
(6244 flops enabled)



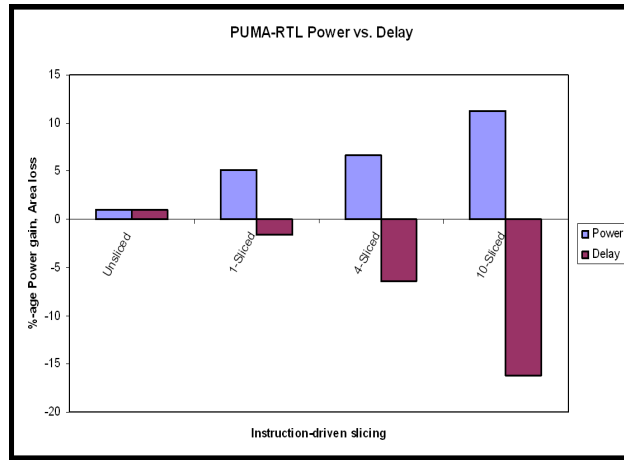
(d) After slicing on 1 . 1w
(4077 flops enabled)

Figure 3.15: Flop distribution effect of instruction-driven slicing on 1 . add and 1 . 1w in the PUMA RTL.

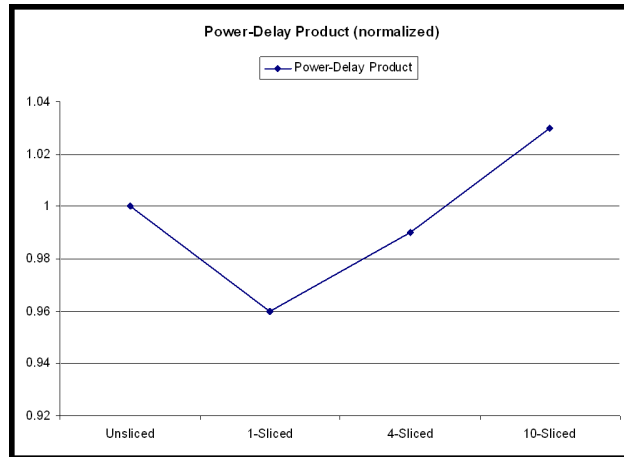
[91] process, a $0.18\mu\text{m}$ generic process technology, to estimate the power dissipation.

Component	Un sliced	1-sliced	4-sliced	10-sliced
Power (mW)	396	375.75	369.5	351.25
Delay (ns)	2.47	2.51	2.63	2.87
Area (mm ²)	15.26	15.37	15.59	15.93

Figure 3.16: Time delay and Area estimate for the PUMA RTL.

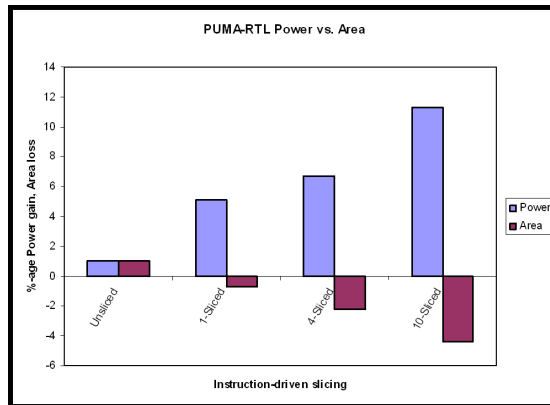


(a) Power gain vs. Increased delay

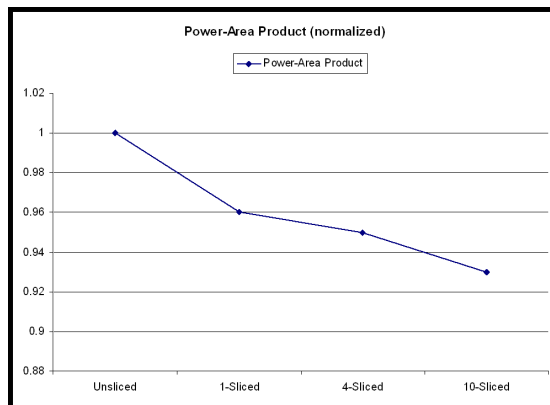


(b) Power-Delay product

Figure 3.17: Power vs. Delay for the PUMA-RTL.



(a) Power gain vs. Increased area



(b) Power-Area product

Figure 3.18: Power vs. Area for the PUMA-RTL.

3.5.2 Results for PUMA-RTL

We used our tool-chain to test our methodology on this processor. The RTL annotations were automatically generated and added to the PUMA RTL in this experiment. The results are shown in Figure 3.12. It is important to note that these numbers are on models of the processor, and were not originally designed to be

power-efficient. The key result therefore, is the percentage improvement in power dissipation. The results are summarized in Figure 3.14. As with the OR1200 case, we also show results of turning off the default clock-gating provided by Synopsys Power Compiler in Figure 3.13. Instruction-driven slicing power gains are very similar even when the auto-power reduction mechanism of the measuring tool is turned off.

We also synthesized our design and ran it through a place-and-route tool [26], both before and after the slicing. The design contained 6244 flops before slicing. In the unsliced version, all 6244 enables are treated as ON, as shown in Figure 3.15(a). After instruction-driven slicing on `add`, 2813 flops are disabled during the course of execution of the `add`. Figure 3.15(b) shows the flop distribution after slicing on the `add` instruction. The parts of the chip that are lit are all the enables on the flops which are on during the execution of the `add`. In the unsliced layout, the entire chip is on, as opposed to the sliced layout where we can clearly see the fine-grained clock gating induced by our algorithm. Figure 3.15(d) shows the same comparison for a `load` instruction (2167 flops are disabled in this case). The flop distribution in Figure 3.15 is based on preliminary floor plan estimate, whereas, the number of enables in each case is accurate.

We measured the change in timing and area induced by instruction-driven slicing. Figure 3.16 tabulates the results. As expected, there is an increased cost of area and delay (max delay along the critical path reported) because of the slicing. Figure 3.17(a) shows the increase in delay with respect to decrease in power dissipation, and Figure 3.17(b) shows the power-delay product. Based on these graphs

we can make an informed decision on the extent of instruction-driven slicing to employ, depending on the extent of slack available in the timing and area constraints. Similar results for area are shown in Figure 3.18.

Instruction-driven slicing has a larger detrimental effect on the timing rather than the area. As we increase the number of instructions we are slicing on, the normalized power-delay product initially decreases (for 1 and 4 instructions), but increases for the 10-sliced case. The point of inflection indicates the limit at which the power gains from slicing are overridden by the timing loss. However, the increase in area is marginal and as shown in Figure 3.18, the power-area product just keeps decreasing with more instructions. The over-approximation built into our slicing algorithm is revealed in the non-monotonicity of these graphs.

3.5.3 Results for PUMA-ARCH

We ran the same benchmarks on our architectural model `sim-puma`. The results are shown in Figure 3.19. `sim-puma` absolute power dissipation estimations were more than the RTL estimations. The percentage improvement observed was also lesser than in the RTL model. We believe this behavior is a direct correlation of how fine-grained the clock gating is. Also, since the architectural model is more abstract than the RTL model, it is natural to expect lesser gains on the architectural model. The results are summarized in Figure 3.20.

We have obtained positive results on the PUMA which is strictly a fixed point unit. The gains would be predictably much larger in a processor which has two floating point units, an on chip L2, *etc.* Also, PUMA, unlike OR1200, having

SPECINT2000 Benchmarks	Unsliced Power Dissipation	1-Sliced Power Dissipation	%-age Gain
gcc	412.00 mW	396.00 mW	3.88%
gzip	411.00 mW	403.00 mW	1.95%
parser	443.00 mW	429.00 mW	3.16%
vortex	477.00 mW	456.00 mW	4.40%
Average	435.75 mW	421.00 mW	3.38%

SPECINT2000 Benchmarks	Unsliced Power Dissipation	4-Sliced Power Dissipation	%-age Gain
gcc	412.00 mW	396.00 mW	3.88%
gzip	411.00 mW	391.00 mW	4.87%
parser	443.00 mW	416.00 mW	6.09%
vortex	477.00 mW	447.00 mW	6.29%
Average	435.75 mW	412.50 mW	5.34%

SPECINT2000 Benchmarks	Unsliced Power Dissipation	10-Sliced Power Dissipation	%-age Gain
gcc	412.00 mW	379.00 mW	8.01%
gzip	411.00 mW	381.00 mW	7.30%
parser	443.00 mW	397.00 mW	10.38%
vortex	477.00 mW	429.00 mW	10.06%
Average	435.75 mW	396.50 mW	9.01%

Figure 3.19: PUMA-Arch Power dissipation results for SPECINT2000 benchmarks after slicing on 1, 4 and 10 instructions.

multiple instructions in-flight on any given clock cycle also has less opportunity to fully utilize our technique.

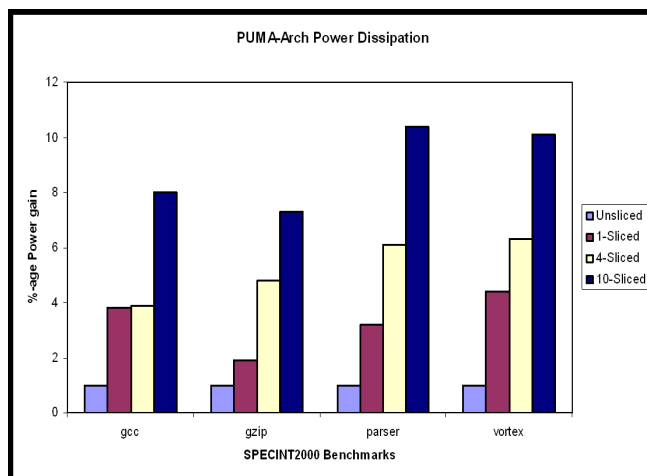


Figure 3.20: PUMA-Arch Power gains for SPECINT2000 benchmarks.

3.6 Conclusions

In this chapter, we have proposed *instruction-driven slicing*, a new technique to automatically annotate RTL for reducing power dissipation by switching activity. We have implemented the *instruction-driven slicing* algorithm and have incorporated it into the design flow tool-chain. We have automatically sliced the RTL and architectural models for OR1200, a pipelined implementation of the OpenRISC instruction set architecture and for PUMA, a PowerPC dual-issue, out-of-order, superscalar fixed point unit. We have used our tool-chain to test our methodology on this processor and have obtained encouraging results.

Our algorithm is particularly suited for in-order pipelined processor designs. It can be applied to out-of-order superscalar processors too. However the reduction in power in the case of PUMA was expectedly substantially lesser than the OR1200 case since there might be multiple instructions in flight in any pipeline stage of the

PUMA, thereby reducing the amount of logic we can actually shut off.

Although our algorithm is conservative, it automatically identifies a close-to optimal set of flops. Our instruction-driven slicing algorithm can be thought of as a wrapper to implement more sophisticated methods of identifying flops which control the circuitry outside the slice.

All previous program slicing algorithms slice the program graph syntactically. The key innovative idea in this technique which separates it from its priors is that it computes a cone of semantic influence of an instruction. It is noteworthy that this information is available only at the RTL and architectural level of description, but is lost at the netlist level where most prior power optimization techniques reside.

Chapter 4

Dedicated Rewriting: Correctness of Low Power Transformations in RTL

4.1 Introduction

In this chapter we introduce a methodology for proving correctness of low power transformations in RTL. Formal verification of digital hardware can be classified broadly into two categories – state-space based techniques and deductive techniques. State-space based techniques like model checking [64], BDD-based verification [21],[22] etc., reason with the state space of the entire system at the Boolean level. Although there have been dramatic improvements in model checking in recent years with bounded model checking, Satisfiability-based techniques, design space partitioning, hierarchical verification etc., the methods still remain intractable to be applied to increasingly growing large low power designs of today’s SoCs. In contrast, deductive verification techniques like theorem proving [67],[113], rewriting [65] etc., try to solve the verification problem by equational reasoning. Due to the high computational complexity of the verification problem, computer aided verification methods are all partial or incomplete. In the case of automatic state-space based methods, this incompleteness manifests as *state-space explosion*, leading to practical time and space limitations. In the case of deductive methods that circumvent state space explosion and are size independent, the incompleteness manifests

itself as a lower degree of automation, requiring manual intervention during the verification process. Therefore, while state-space based approaches cannot handle circuits of even reasonable sizes, deductive verification approaches involve a significant manual component. Despite their incompleteness, theorem provers are used for verification of complex hardware systems due to their efficiency in handling real designs, and the high degree of confidence they provide.

We present *dedicated rewriting*, a rewriting methodology to automatically prove the correctness of low power transformations at the RT-level. We propose a highly automated deductive verification technique which is fine-tuned for low power transformations. We first describe our work in the context of arithmetic circuits [142] and then extend it into time-domain and sequential circuits. We present a dedicated low-power transformation prover, as opposed to a generic rewriting engine which would involve considerable user interaction. We prove the equivalence of two Verilog RTL designs, one derived from the other after the application of a low power transformation. Our notion of equivalence is defined with respect to an observable. An observable is a variable in the Verilog RTL at a particular time, that, by specification, is expected to have the same function in both RTLs, at possibly different times. The inputs to our system are the two RTLs and a set of observables.

The computation model in our technique involves Term Rewriting Systems (TRSs) [68]. A TRS is defined as a tuple of terms and rules. Our technique starts by automatically deriving a TRS from a given Verilog RTL. The variables in the Verilog RTL form the terms of the TRS. The rules of the TRS represent the hardware design and are both time-preserving and atomic. The time-preserving nature of the

rules guarantees that both explicit and implicit timing dependencies in Verilog RTL are captured in the rules. Essentially, this means that a given variable at different times is treated as separate terms, with the corresponding time annotation. Atomic transactions ensure that a given rule is executed in its entirety without interruption or conflict with rest of the system. We derive TRSs from both Verilog RTLs, before and after applying the low power transformation. We have defined a notion of equivalence of two TRSs with respect to an observable (a term in both TRSs). We now proceed to prove the equivalence of the two TRSs with respect to the observable. In the process we create a dedicated database of low power transformation rules. If a proof cannot be established, then we either add a new rule to the database, or we have found a bug. This process is repeated for all observables.

We present the result of our technique on different low power transformations applied to a Verilog RTL implementation of Viterbi decoder [150] module that is a part of the Digital Radio Mondiale (DRM) SoC [2]. The main contributions of this work are the following.

- We present a methodology to automatically verify correctness of applying a low power transformation on existing hardware, thereby drastically reducing design cycle time.
- We present dedicated rewriting, an automatic and dedicated prover for low power transformations in RTL. There is minimum overhead of providing environment and additional lemmas as opposed to a general purpose rewriting engine.

- We present a novel notion of capturing the functional and timing description of RTL in the form of atomic transactions called rules.
- We define and use a novel notion of decomposed TRS equivalence.
- We have created a dedicated database of low power transformation rules.
- We leverage the expressive power and relative simplicity of high-level designs by reasoning entirely at that level.
- We demonstrate our technique by proving the correctness of multiple low power transformations on a real life SoC RTL.

Term Rewriting Systems have been used in the past for program verification [8], [9], [46]. In the context of hardware, rewriting strategies have been used in the past to design correct circuits [57], [115], [114], [13]. Term Rewriting Systems were first proposed for hardware verification in [28]. Subsequently, they have been used for checking functional correctness of hardware [158], [92]. More recently, a System Verilog based rewriting system for RTL abstractions was introduced by [54] in the context of Pentium processor. We presented a previous version of dedicated rewriting, in the context of automatic verification of combinational arithmetic circuits [142]. This work is a more generalized version in a completely different hardware context of low power transformations at the RT-level.

We explain our notion of rules in full detail in Section 4.2. Section 4.3 describes our dedicated rewriting methodology. In Section 4.4 we present dedicated rewriting as applied to multiplier verification. In Section 4.5 we present a case study of using

our technique on multiple low power transformations on the Viterbi decoder. We discuss the merits of our technique and conclude in Section 4.6.

4.2 Rules

We briefly review several definitions and concepts about term rewriting in Figure 4.1 and Figure 4.2. See [68], [39] for a detailed treatment of Term Rewriting Systems.

1. A Term Rewriting System (TRS) is defined as a tuple $\langle S, R, S_0 \rangle$, where S is a set of terms, R is a set of rewriting rules, and S_0 is the set of initial terms ($S_0 \in S$).
2. The state of a system is represented as a TRS term, while the state transitions are represented as TRS rules.
3. The general structure of rewriting rules as an ordered pair of terms is as follows:
Rule: $s_1 \longrightarrow s_2$ if $p(s_1)$
where s_1 and s_2 are terms and p is a predicate.
4. If $s_1, s_2 \in S$, and $\alpha \in R$, then $s_1 \xrightarrow{\alpha} s_2$ denotes that the term s_1 can be rewritten to term s_2 by the rule α .
5. If the left-hand-side pattern of a rule matches a term or one of its sub-terms, and the corresponding predicate of the rule is true, then the rule can be used to rewrite the term. The new term is generated in accordance with the rule's right-hand side. If several rules apply, then any one of them can be applied. If no rule applies, the term cannot be rewritten any further and is said to be in *normal form*.

Figure 4.1: Term Rewriting Systems: Definitions and concept

6. We say term s_1 can be rewritten to s_2 in zero or more rewriting steps ($s_1 \longrightarrow s_2$), if $s_1 = s_2$, or there exists a term s_3 such that, $s_1 \longrightarrow s_3$ and $s_3 \longrightarrow s_2$.
7. A term s is *legal* if there exists $s_0 \in S_0$ such that $s_0 \longrightarrow s$.
8. A TRS is *terminating* if there are no infinite rewrite sequences $s_1 \rightarrow s_2 \rightarrow \dots$.
9. A TRS is *confluent* if, for any term s_1 , if $s_1 \longrightarrow s_2$ and $s_1 \longrightarrow s_3$, then there exists a term s_4 such that $s_2 \longrightarrow s_4$ and $s_3 \longrightarrow s_4$, *i.e.*, any divergence in rewriting is eventually joined.
10. A *normal form* is a term which cannot be rewritten any further.
11. A TRS is *strongly terminating* if, for any term, it can always be rewritten to a normal form using any rewriting strategy.
12. Termination ensures the existence of normal forms, while confluence ensures their uniqueness.

Figure 4.2: Term Rewriting Systems: Definitions and concept (continued)

Applying a rule is also called executing or firing. When a rule is applied, the state on the left-hand-side is read at the beginning of the clock cycle and updated at the end of the clock cycle. This single cycle notion automatically enforces the atomicity constraint of each rule. All enabled rules fire in parallel (or in no particular order). If two rules modify the same state element, then we have a race condition. We expect the Verilog RTL to be race-free and combinational-loop-free at the input. This is an easy constraint to impose on the input Verilog, since it can be checked by standard Verilog linting tools.

We have two kinds of rules within our system, structural rules and logical rules.

```

// Instruction selection in load/store unit
assign lsu_op_next = lsu_op;

always @(posedge clk or posedge rst) begin
  case (id_insn[31:26])
    `OR1200_OR32_SB: lsu_op <= lsu_preop_SB;
    `OR1200_OR32_SW: lsu_op <= lsu_preop_SW;
    `OR1200_OR32_LBZ: lsu_op <= lsu_preop_LBZ;
    `OR1200_OR32_LWZ: lsu_op <= lsu_preop_LWZ;
    default: begin lsu_op <= `OR1200_LSUOP_NOP;
  endcase
endcase
end

```

(a) Example Verilog RTL code assigning lsu_op.

```

// Instruction selection in load/store unit
Rule1: lsu_op_next(t) → lsu_op(t)
      if (T)
Rule2: lsu_op(t) → lsu_preop_SB(t-1)
      if ( (id_insn[31:26](t) == `OR1200_OR32_SB)
          and (posedge_clk or posedge_rst) )
Rule3: lsu_op(t) → lsu_preop_SW(t-1)
      if ( (id_insn[31:26](t) == `OR1200_OR32_SW)
          and (posedge_clk or posedge_rst) )
Rule4: lsu_op(t) → lsu_preop_LBZ(t-1)
      if ( (id_insn[31:26](t) == `OR1200_OR32_LBZ)
          and (posedge_clk or posedge_rst) )
Rule5: lsu_op(t) → lsu_preop_LWZ(t-1)
      if ( (id_insn[31:26](t) == `OR1200_OR32_LWZ)
          and (posedge_clk or posedge_rst) )
Rule6: lsu_op(t) → `OR1200_LSUOP_NOP
      if ( (id_insn[31:26](t) != (`OR1200_OR32_SB
          or `OR1200_OR32_SW or `OR1200_OR32_LBZ
          or `OR1200_OR32_LWZ))
          and (posedge_clk or posedge_rst) )

```

(b) Rules derived from the Verilog RTL code.

Figure 4.3: Sample Verilog RTL and the TRS Rules derived from it.

```

input clk;
input pgate_signal;

always @(posedge clk) begin
  case (sw)
    0: butterfly_1 = fifo[11:8] + gsm[23:20];
    1: butterfly_1 = fifo[7:4] + gsm[19:16];
    default: butterfly_1 = fifo[3:0] + gsm[15:12];
  endcase
end

```

(a) Pre-transformation Verilog RTL.

```

input clk;
input pgate_signal;
assign gated_clk = clk & ~pgate_signal;
always @(posedge gated_clk) begin
  if (c3_state) butterfly_1 = fifo[3:0] + gsm[15:12];
  else
    case (sw)
      0: butterfly_1 = fifo[11:8] + gsm[23:20];
      1: butterfly_1 = fifo[7:4] + gsm[19:16];
      default: butterfly_1 = fifo[3:0] + gsm[15:12];
    endcase
end

```

(b) Post-transformation Verilog RTL.

Figure 4.4: Clock gating for lower switching activity power dissipation (Verilog RTL).

Structural rules are timing preserving atomic transactions representing a state update in hardware. These are derived directly from the Verilog RTL. Logical rules represent identities about the RTL operators and carry information about the low

power transformations. We explain these further in the next two subsections.

4.2.1 Structural Rules

Consider the example in Figure 4.3. Verilog RTL for a module that is selecting an instruction in the load/store unit of a microprocessor is shown in Figure 4.3(a). When we derive the TRS from the Verilog RTL, we arrive at the structural rules as shown in Figure 4.3(b). The structural rules are a syntactic translation with timing information at the same level of abstraction as the Verilog RTL. The resulting set of rules can now be used to compute the symbolic term of any signal at a particular time t in terms of other signals at different times $k, k < t$ and/or primary inputs.

Each hierarchical signal in Verilog is represented by a new function symbol (*signal function*), thereby creating a “flattened” TRS. Structural rewrite rules rewrite each signal function into an expression consisting of RTL operators and other signal functions. The cycle-accuracy of the RTL semantics is maintained in the TRS by each signal function being a function of time t . The notion of time is relative. A combinational logic assignment in the Verilog is a rewrite rule with all terms being a function of the same time t (the rule Rule1 in Figure 4.3(b)). Whereas, a sequential logic assignment in the Verilog is a rewrite rule with the assigned term being a function of time t and all other terms relatively 1 cycle before the assigned term, as a function of time $(t - 1)$ (rules Rule2 through Rule5 in Figure 4.3(b)).

```

Rule: butterfly_1(t) →
      fifo[11:8](t-1) + gsm[23:20](t-1)
      if (sw(t-1) == 0) and (posedge_clk(t-1))
Rule: butterfly_1(t) →
      fifo[7:4](t-1) + gsm[19:16](t-1)
      if (sw(t-1) == 1) and (posedge_clk(t-1))
Rule: butterfly_1(t) →
      fifo[3:0](t-1) + gsm[15:12](t-1)
      if ((sw(t-1) != 0) and (sw(t-1) != 1))
          and (posedge_clk(t-1))

```

(a) Pre-transformation TRS Rules.

```

Rule: gated_clk(t) →
      (clk(t) & ~pgate_signal(t))
      if (T)
Rule: butterfly_1(t) →
      fifo[3:0](t-1) + gsm[15:12](t-1)
      if (c3_state(t-1) == T)
Rule: butterfly_1(t) →
      fifo[11:8](t-1) + gsm[23:20](t-1)
      if (sw(t-1) == 0) and (c3_state(t-1) != T)
          and (posedge_gated_clk(t-1))
Rule: butterfly_1(t) →
      fifo[7:4](t-1) + gsm[19:16](t-1)
      if (sw(t-1) == 1) and (c3_state(t-1) != T)
          and (posedge_gated_clk(t-1))
Rule: butterfly_1(t) →
      fifo[3:0](t-1) + gsm[15:12](t-1)
      if ((sw(t-1) != 0) and (sw(t-1) != 1))
          and (c3_state(t-1) != T)
          and (posedge_gated_clk(t-1))

```

(b) Post-transformation TRS Rules.

Figure 4.5: Clock gating for lower switching activity power dissipation (after translation to TRS).

```

Rule:  ( x & x ) → ( x ) if (T)
Rule:  ( x & (y & z) ) → ( ( x & y) & z) if (T)
Rule:  ( (x << 1) - x ) → ( x ) if (T)
Rule:  ( x + y ) → ( y + x ) if (T)
Rule:  ( x << 1 ) → ( x * 2 ) if (T)
Rule:  ( x | T ) → ( T ) if (T)
Rule:  ( (x << 2) - x ) → ( x * 3 ) if (T)

```

Figure 4.6: Sample logical rules in the dedicated rule database.

4.2.2 Logical Rules

The logical rules codify identities about RTL operators and various low power transformations. This is an independent database that is part of our dedicated rewriting system. These rules can be generic or low power transformation specific. Some examples of the generic rules in this database are shown in Figure 4.6.

Generic rules define RTL operator identities and are helpful in simplification of terms during the equivalence proof. We also do not restrict the level of abstraction of the input RTL. This structure in our system allows us to, for example, use a new (or non-synthesizable) RTL operator in the input Verilogs, and define the identities of that operator within this rule database to allow for simplification during the proof process. Typically, the predicate function of these generic logical rules is always true (T). This also allows for using abstract (uninterpreted) functions in the RTL, while the interpretation can be formulated in the form of logical rules within the database.

The low power transformation specific rules define the identities associated with the transformation. These will be different for each transformation. However, these

are independent of the RTL on which the transformation will be applied, and as such need to be incorporated as part of this database exactly once, for each transformation.

Consider the clock gating example in Figure 4.4 and Figure 4.5. It shows the Verilog RTL and the derived TRS rules before and after the application of the transformation to achieve lower switching activity power dissipation. In the above example, the transformation creates a new clock gated by the signal `pgate_signal`. As shown in Figure 4.5(b) two new rules corresponding to the creation of the gated clock are added to the list of rules. In this example, apart from these structural rules, two transformation specific rules are added to the logical rule database:

- An external assumption when to enable power clock gating decides the value of the signal `pgate_signal`. The enabling assumption needs to be codified into the transformation specific rules database as follows:

Rule: (`pgate_signal`) \longrightarrow T if (T)

If we were to run our proof with power clock gating disabled, then we would add the corresponding rule.

- The algorithm of power gating has the hardware assume a special state (`c3_state`) when the transformation is enabled. This information is captured in a transformation specific rule as follows: corresponding to which is

Rule: (c3_state) \rightarrow T if (pgate_signal==T)

Assumptions of this nature which are very specific to the low power transformation need to be specially encoded as rules in order to assist the proof system.

Rules are powerful representations of always blocks. The active rules, where the guards are true, can be applied in parallel, but each rule operates as an atomic transaction, *i.e.*, each rule observes and ensures a consistent state relative to all other rules in the system.

4.3 Dedicated Rewriting

We propose a refinement based rewriting methodology to automatically generate proofs for low power transformations in RTL. Figure 4.7 gives a flow chart representation of our proof methodology.

The input to the system are two RTLs, an original RTL and a transformed RTL (after the application of the low power transformation), and a set of observables. The proof methodology works in three primary steps. First, we derive the structural rules of the TRS from the Verilog RTL for both models. Next, we execute the rules to derive expressions for all observables in each model. This is guaranteed to complete since the terms represent Verilog variables of finite width. Finally, for each observable, we go through an iterative, mostly automated proof process. These are labeled as stages 1, 2, and 3 in Figure 4.7. Figure 4.8 gives the algorithm for

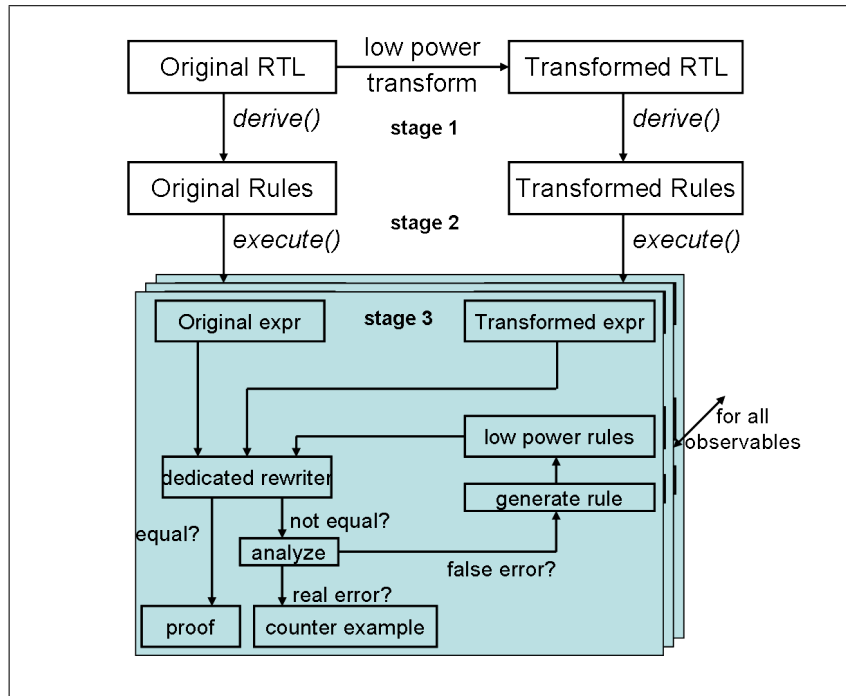


Figure 4.7: Dedicated rewriting proof system flow chart.

the dedicated rewriting procedure. We describe the procedure stage by stage by elaborating on the functions involved in each stage in the rest of this section.

4.3.1 *derive ()*: Verilog RTL to TRS rules

This function translates Verilog RTL to TRS rules. As we described in Section 4.2, rules are atomic transactions carrying timing information. The *derive ()* function will generate the structural rules of the TRS. We have fully automated this translation process. Examples of this automated translation in different contexts are shown in Figure 4.3 and Figure 4.5. We had described a Verilog RTL to TRS translation in our previous work on automatic verification of arithmetic circuits [142].

This function is an extension of that process, with the primary difference being the notion of time annotation in the current process. Combinational logic statements in the RTL get translated to rules with all terms at the same relative time t , whereas, sequential logic statements get translated to different relative times. We do this translation on both models.

The fully automatic nature of this translation allows us to use our methodology on any existing Verilog RTL design. This is particularly useful in the context of verification of low power transformations since most of these transformations tend to be RTL changes late in the design stage in order meet the power requirements of the landing zone.

4.3.2 *execute ()*: TRS rules to expressions

This function computes the expression of a particular observable by rewriting the structural TRS rules. The process of rewriting is based on firing the rules that are ready to fire. If more than one rule can fire at any given time (clock cycle), then all the rules fire in no particular order. Since our rules are strictly atomic transactions this maintains the correctness of the Verilog semantics. At the end of this process, we have essentially computed the symbolic expression of the observable. We do this in tandem on both models for each observable in the input set of observables. The next subsection explains how the equivalence of the two expressions in the two models, for each observable is proved. Every such point where the two expressions are proved equivalent (can be at different times in each model) is called a *Compare point*.

Our compare points are defined as co-ordinates on the space-time axis of the design, denoted by their relative position with respect to the time domain (clock cycles), and their position in the space domain (data variables). This aligns with the sequential behavior of the designs being compared, and provides an easy, intuitive abstraction of the equivalence checking problem space.

Depending on the design we are working with, these expressions can be arbitrarily large, thereby making the equivalence proof arbitrarily hard. In order to mitigate that problem, we use some heuristics to reduce the complexity of the expressions. One heuristic we have successfully used in the context of arithmetic circuits is a bit-wise partitioning of assignments to a particular RTL signal. If we have matching bit partitions aligned in both models, then the expression corresponding to that bit partition is treated as the basis for the equivalence proof between the models. We have employed this heuristic with great success in the context of arithmetic circuits [142], and we detail the heuristic here.

4.3.2.1 Reassignments

If all the bits of an observable variable are assigned together (in one time step) in a model, the variable is included in the list of observables as it is. However, if the bits of the variable are assigned separately (in different time steps), there will be more than one observable, corresponding to the same variable. We clarify this with an example.

Consider an example from the universe of arithmetic circuits, a 32-bit multiplier that we would like to verify, which has `mul_result[31:0]` as an output,

and therefore an observable. If the multiplier RTL model has only one assignment statement assigning the entire value `mul_result[31:0]`, then there will be a single observable, namely, `mul_result` added to the list of observables.

Assume the multiplier's RTL is modeled such that 8 bits of the output are assigned a value together, *i.e* at the same time. All the 32 bits of the output are, therefore, assigned values after 4 such assignments. Each assignment generates an observable for `mul_result`. Hence, there will be 4 observables that correspond to `mul_result[7:0]`, `mul_result[15:8]`, `mul_result[23:16]`, and `mul_result[31:24]`.

Every subset of bits assigned, therefore, has a corresponding observable. We call such assignments (to different subsets of bits of the same variable), *reassignments*, as in [142]. Thus, a reassignment for a variable defines a partition of the bits for the variable. In our example, the 4 *reassignments* define the partition $\{[31:24], [23:16], [15:8], [7:0]\}$ on the 32 bits of the output signal `mul_result`.

In order to illustrate the reassignment process, let us now assume the specification for the multiplier is modeled as a shift-and-add design which has `spec_mul_result` as an output variable. This model assigns a value to the output 1 bit at a time. Therefore, in the specification model, there will be 32 *reassignments* defining the partition $\{31, 30, \dots, 2, 1, 0\}$ on the bits of the signal `spec_mul_result`.

Observables are computed for every major variable in our design as follows:

1. The reassignment bit partitions in the design are computed. In our example, the original design partition is $\{31, 30, \dots, 2, 1, 0\}$ and the transformed

design partition is $\{[31:24], [23:16], [15:8], [7:0]\}$.

2. A new variable is defined for every set of bits in the pairwise intersection of these two partitions. In our example, the pairwise intersection groups entries of the original partition together. The new observables will be $mS1$, $mS2$, $mS3$, and $mS4$ corresponding to the bit sets $\{7, 6, \dots, 1, 0\}$, $\{15, 14, \dots, 9, 8\}$, $\{23, 22, \dots, 17, 16\}$, and $\{31, 30, \dots, 25, 24\}$ respectively. The new variables in the transformed design will be $mV1$, $mV2$, $mV3$, and $mV4$ corresponding to the bit sets $\{[7:0], [15:8], [23:16], [31:24]\}$ respectively.
3. The new observables obtained are mapped to establish their correspondence, and added to the list of observables. In our example, the variables `spec_mul_result` and `mul_result` are mapped into four pairs of observables, namely, $\{(mS1, mV1), (mS2, mV2), (mS3, mV3), (mS4, mV4)\}$.

We thus compute a partition of the bits for a particular output defined by the reassignments in both original and transformed models. This is a simple heuristic that appears to work well for models with common outputs and possibly some common internal points.

Our *execute()* procedure is also completely automated and includes the above decomposition heuristic. We have structured our methodology such that it can accommodate any decomposition strategy. Our tool is designed such that we can easily and seamlessly incorporate any library of decomposition heuristics in our routine that calculates expressions by rewriting structural rules.

Algorithm Dedicated Rewriting.

```
main ( $V_o$ : Original RTL model,  $V_p$ : Transformed RTL model,  
       $O$ : Set of observables,  $DRdb$ : dedicated rule database)  
begin  
  proved = T  
   $T_o$  = derive ( $V_o$ )  
   $T_p$  = derive ( $V_p$ )  
  for every observable  $o \in O$   
  begin  
     $\{\langle expr_1, expr_2 \rangle\}$  = execute ( $T_o$ ,  $T_p$ ,  $o$ )  
  end  
  for every pair of expressions  $\langle expr_1, expr_2 \rangle$   
  begin  
    proved = proved && (prove ( $expr_1, expr_2, DRdb$ ))  
  end  
  return (proved)  
end  
prove ( $expr_1$ : Original expression,  $expr_2$ : Transformed expression,  
       $DRdb$ : dedicated rule database)  
begin  
  do  
  begin  
    out = dedicated_rewrite ( $expr_1, expr_2, DRdb$ )  
    if (out == T) return (T)  
    else  
    begin  
      true_error = analyze ()  
      if (true_error == T)  
        return (counter_example)  
      else  
        generate_and_add ( $DRdb$ )  
    end  
  end  
  while (out==F && true_error==F)  
end
```

Figure 4.8: Dedicated rewriting algorithm

4.3.3 *prove ()*: Equivalence of expressions

In this function we check the equivalence of two expressions by rewriting based on simplification using the logical rules from the dedicated rule database. As described in Subsection 4.2.2 these logical rules codify various identities about RTL operators, as well as rules specific to the low power transformation.

These rules can be generic or design specific or transformation specific. While trying to prove the equivalence of two expressions, we select the set of rules ready to fire from the rule database and apply them in some arbitrary order. If the resulting simplifications fail to establish the equivalence, then the rules are applied in a different order. These *proof iterations* continue until equivalence is established or no more rules can be applied in any order. This step is executed in the function *dedicated_rewrite ()*.

This function is repeated for every compare point, and if the expressions at every compare point turn out equivalent, then the two designs are declared equal.

When two designs are declared unequal by this technique, the eponymous function *analyze ()* analyzes the result for each output expression and discovers one of two possibilities. In the first case, the two designs are truly not equal, in which case we have caught a “bug” in the transformation. In this case, our technique provides a detailed proof trace and also picks a counterexample starting from the inputs at specific times. In the second case, we may not have been able to establish the equivalence due to insufficient rules in the dedicated rule database. In this case we allow for user intervention to create and add the required rules to the database

(this is handled in the function *generate_and_add()*), and the entire process can be repeated again. One other possibility is that our previously added rule could be wrong, and this step allows for fixing that as well, although it does not prevent false positives due to wrong rules. The shaded part of Figure 4.7 describes this process. This entire process is repeated for every pair of expressions that need to be proved equivalent.

Figure 4.8 gives the full algorithm for the dedicated rewriting process we described in this section. Once we establish the equivalence of two TRSs with respect to a given set of observables, in effect, we have proved the correctness of the low power transformation which created the transformed RTL from the original RTL. In this process, we have established the logical transformation specific rewrite rules as part of our dedicated rule database. Therefore, all further proofs of application of the same transformation on any other RTL design should be possible with minimal user intervention or changes to the dedicated database. This is what separates dedicated rewriting from a general purpose rewriting engine or a theorem prover, and makes the technique a highly automatable approach.

4.3.4 Our notion of equivalence

Many notions of sequential equivalence have been proposed in the literature. Most of them adhere to the broad classification of equivalence with respect to a set of initial states [136],[71] or alignability equivalence that can demonstrate resetability across all states [101]. All these notions of sequential equivalence are at the gate level, and deal with retiming and synthesis based optimizations. They also build the

state-transition relation in order to reason about sequential equivalence. Although we are dealing with a different level of abstraction, our notion of equivalence is more along the lines of [136] than the alignability notion of equivalence. We state our theories of correctness with respect to a set of initial states. This paradigm has higher scalability [48], due to the potential leveraging of many existing algorithms. Since we perceive our technique to act synergistically with the existing Boolean level algorithms, we prefer to use this notion of equivalence.

4.3.5 Error Detection

An inherent limitation of selecting observables is that the information about the cycle of comparison is obtained from the RTL implementation model itself. The state-transition graph or the simulation of the RTL provide accurate information about the time at which an observable is available for comparison according to the design. Since our technique attempts to capture the design progression in time as well as in data space, we present a brief discussion about the functional and temporal error scenarios in our domain, and how our technique performs in these scenarios.

There are four possible outcomes of the *prove()* function in Figure 4.8 when comparing the two expressions of an observable at any compare point $C = (t, v)$, such that t is the time at which the observable v is computed in the RTL model.

- Functionally and temporally correct.

In this case, *prove()* returns **true**, t is the correct cycle of computation, and t is the time of comparison. This is the case when the algorithm will return a

true value. This means that the symbolic function of v in the both the RTLs is equivalent, as well as compared at the right time.

- Functionally incorrect and temporally correct.

In this case, *prove()* returns **false**, t is the correct cycle of computation and t is the time of comparison. This is the case when the algorithm will return a **false** value. This means that the symbolic function of v in the transformed design does not match with the function in the original design. This scenario is detected by our technique. An error trace is provided between the past compare point and the current compare point. This scenario could also be indicative of inadequate logical rules. That part of the process is taken care of in the *analyze()* and *generate_and_add()* functions.

- Functionally incorrect and temporally incorrect.

In this case, the *prove()* procedure returns a **false**, t is not the correct cycle of computation, and t is the time of comparison. Incorrect cycle of computation refers to a time when temporally the data is not yet stable. The algorithm now provides a functional error trace, but not a temporal error trace. In other words, there is a possibility of obtaining false negatives in this scenario, since a mismatch does not indicate if there is an error in the functionality or timing. Again, the functional incorrectness could be indicative of inadequate logical rules.

- Functionally correct and temporally incorrect.

In this case, the *prove()* procedure returns a **true**, t is not the correct cycle

of computation, and t is the time of comparison. If the design is flawed with respect to time of computation, and if the comparison point is not at the “flawed” cycle, but another cycle, the designs will not match in functionality. However, in the case where the design itself has a timing bug, and we check at the (incorrect) cycle that the design computes its (correct) data, we will not be able to find the bug, and it can result in a false positive. This situation cannot be avoided, due to the inherent limitation of a sequential equivalence technique that uses the timing information from the implementation itself. However, in the case where the specification details the timing, or we have an external timing specification (supposedly reliable), this rare case of errors can be avoided.

4.4 Dedicated Rewriting for Combinational Equivalence Checking: Multiplier Verification

We used an earlier version of *dedicated rewriting* in the context of arithmetic circuit verification, and proved the correctness of complicated multipliers at the RT-level. Our technique retains the efficiency and the size independence of deductive verification techniques, while sacrificing automation minimally. Arithmetic circuits have sufficient structural regularity to afford analysis by functional decomposition and are, therefore, ideal candidates for our verification by stepwise refinement [142].

We use a simple Shift-and-Add multiplier as the original design for multipliers. We present the experimental results that we have obtained from our tool. We pro-

duce three sets of results, on a radix 3 Booth multiplier, on a Wallace Tree multiplier and on a Dadda Tree multiplier. The Booth multiplier is an array-based multiplier, whereas the Wallace multiplier has a tree of carry save adders and a single carry lookahead adder as the last stage. The Dadda Tree multiplier uses the more regular redundant binary addition trees [124] instead of a tree of CSAs. The comparisons are made after the computation of 3 output bits at a time. We show the time taken by the tool for increasing sizes of these multipliers.

We have tried to compare our tool to state-of-the-art equivalence checkers. Since the equivalence checkers are most efficient when comparing two gate level designs, we provided gate level implementations of the Booth and Wallace Tree designs as inputs. Although our tool works at the RT level, we have compared the numbers obtained from the gate level verification by the equivalence checkers with our tool output, in order to provide a basis for comparison. It is seen from Figure 4.9(a), Figure 4.9(b) and Figure 4.9(c) that the verification of 8×8 multipliers are performed by both Commercial Equivalence Checker 1 and Commercial Equivalence Checker 2 in time comparable to our tool. However, in the case of 16×16 multipliers, both the equivalence checkers do not run to completion. Our tool, in comparison, verifies the design in 24 seconds. It can also be seen that as the sizes increase, the time taken by our tool scales linearly with the size of the design.

We mentioned in Subsection 4.3.3 that the *prove()* function proves term equivalence using proof iterations. Figure 4.10 shows the number of proof iterations that the tool required to prove the equivalence of the 64×64 multipliers at two sample compare points. The proof iterations for the Booth multiplier at (sample) compare

Booth Multiplier	Dedicated Rewriting	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	16s	12s	9s
$8b \times 8b$	19s	20s	16s
$16b \times 16b$	24s	not completed	not completed
$32b \times 32b$	37s	not completed	not completed
$64b \times 64b$	53s	-	-
$128b \times 128b$	93s	-	-

(a) Booth Multiplier

Wallace Multiplier	Dedicated Rewriting	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	14s	10s	9s
$8b \times 8b$	18s	18s	16s
$16b \times 16b$	25s	not completed	not completed
$32b \times 32b$	40s	not completed	not completed
$64b \times 64b$	60s	-	-

(b) Wallace Tree Multiplier

Dadda Tree Multiplier	Dedicated Rewriting	Commercial Tool 1	Commercial Tool 2
$4b \times 4b$	13s	11s	8s
$8b \times 8b$	17s	19s	17s
$16b \times 16b$	29s	not completed	not completed
$32b \times 32b$	51s	not completed	not completed
$64b \times 64b$	83s	-	-

(c) Dadda Tree Multiplier

Figure 4.9: Comparison of execution times of Dedicated Rewriting against two commercial equivalence checkers for Booth, Wallace Tree and Dadda Tree multipliers of varying sizes. In each case the golden model was a shift and add multiplier of the corresponding size.

points 3 and 21 and the Wallace Tree multiplier at compare points 3 and 7 have been illustrated. We observe that the proof iterations do not increase significantly as the proof progresses, *i.e.* with increasing comparison points. We also observe that

Multiplier	Compare point	Number of rules	Number of proof iterations
Booth	3	107	192
Booth	21	107	212
Wallace Tree	3	107	347
Wallace Tree	7	107	291
Dadda Tree	3	107	462
Dadda Tree	7	107	341

Figure 4.10: Number of proof iterations done by *reduce()* to prove equivalence at the given compare points. The numbers correspond to the 64×64 Booth, Wallace Tree, and Dadda Tree multiplier designs.

the number of compare points for the Wallace Tree are lesser than the Booth. This is because of the tree of carry save adders in the Wallace Tree design which delays assignment to the final output bits. Therefore, the terms are larger and the effect of this is seen in the increased number of proof iterations for the Wallace Tree than for the Booth multiplier.

Traditional combinational equivalence checkers routinely face the issue of not being able to reliably conclude that two designs are not equivalent. This is the problem of *false negatives*. In order to mitigate the verification complexity, equivalence checkers perform *hierarchical verification* that comprises isolated verification of each hierarchical block, under the assumption that exact functional equivalence at hierarchical boundaries is preserved. False negatives occur in such hierarchical verification when either (a) functional equivalence at hierarchical boundaries is not preserved, or (b) when the block of design is functionally equivalent only when is constrained by the environment, not with unconstrained variables as viewed by the

hierarchical verification process [6]. These issues are circumvented by our technique, since we “flatten” the hierarchy during translation of the design from Verilog to TRS.

Another noteworthy difference between our technique and traditional gate level equivalence checkers is that we do not view each internal register as a comparison point. Our comparison points are the assignments or reassignments to the output signals of the design. Consequently, between arithmetic designs whose (output) size and number of outputs are equivalent, a correspondence between the comparison points in the two designs can always be expected.

We have extended this work in the context of sequential circuits and sequential compare points [138], [140], [139] as described earlier in this chapter. The next section details a larger case study using the Viterbi decoder which is a part of the Digital Radio Mondiale SoC.

4.5 Case Study: Viterbi Decoder

We perform our experiments on a Viterbi decoder, that is a part of the Digital Radio Mondiale (DRM), implemented in Verilog RTL. The initial Viterbi decoder RTL is a basic model that implements the Viterbi decoding algorithm, but has no optimizations for power, area, or performance. This is shown in Figure 4.11. On this model, we perform certain low power transformations aimed at reducing the switching activity power dissipation. These transformations do not cross register boundaries. We then use our framework to prove that these transformations do not affect the functionality of the original design.

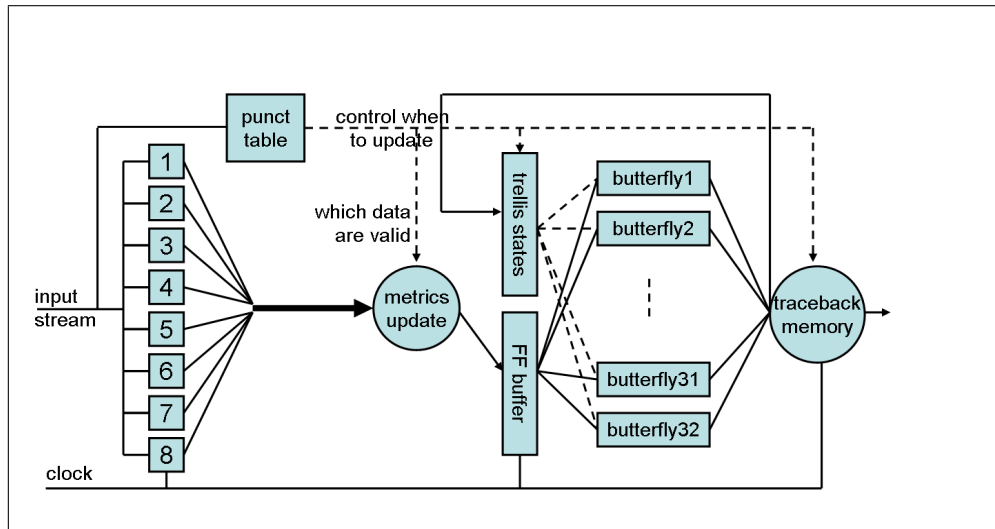


Figure 4.11: Basic Viterbi design.

Next, we use a more complicated Viterbi decoder design, also implemented in Verilog RTL, but one optimized for lower power dissipation. In contrast to the earlier transformations, this transformation crosses register boundaries. The new design is as shown in Figure 4.12. The butterfly network in this design is now pipelined into two stages. We estimate the power dissipation savings and also prove the equivalence of the two sequential designs.

Finally, we use a third Viterbi decoder design, also implemented in Verilog RTL, but optimized for a combination of power and timing. This is more realistic in today's industry where every functional unit block in the design has specific area, power, and timing budgets, and all optimizations have to meet an optimal landing zone satisfying all the three requirements. This new design is as shown in Figure 4.13. Again, we prove the equivalence of the two sequential designs, the optimized design and the original Viterbi design by dedicated rewriting.

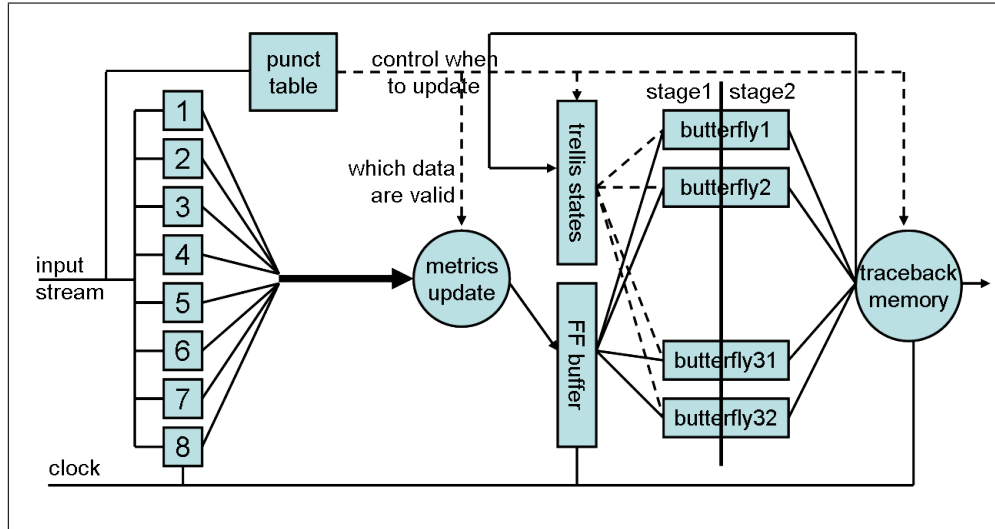


Figure 4.12: Viterbi design optimized for low power.

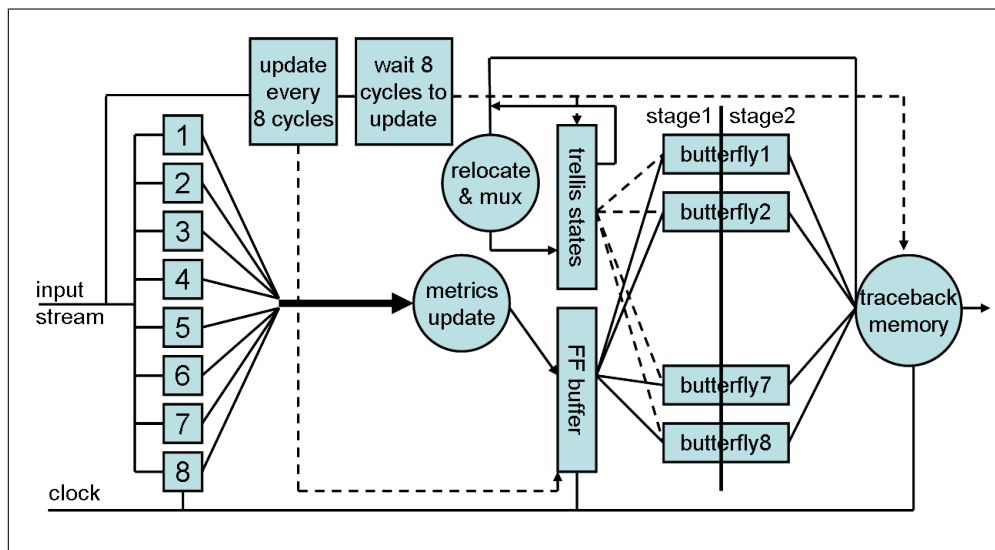


Figure 4.13: Viterbi design optimized for power and delay.

4.5.1 Combinational low power transformations

Figure 4.11 shows the basic block diagram of a Viterbi decoder [150]. There are two major stages to the functionality of the Viterbi decoder. One is collecting the inputs depending on the Puncture Pattern and storing them in a buffer (FF Buffer). The other stage is the Trellis computation. The next state values of the Trellis matrix are computed by a function (Butterfly network) of current state values of the Trellis matrix and the inputs stored in the FF Buffer.

We perform combinational low power optimization in the logic computing the values of the Trellis matrix. The low power transformations performed included common sub-expression elimination, movement of operations, constant propagation and commutativity and associativity based optimizations. All these optimizations were strictly between register boundaries of the design. Hence, the symbolic terms at every comparison point were at the same relative times in both models. We used the macro-modeling technique employed in [50] to estimate power at the RTL level of abstraction. This method involved characterizing the power consumed by the basic combinational blocks (macros) used in the design. We did the characterization for the all macros in the `GetMatrixSet` block of the Viterbi decoder design. All our optimizations were also in the same block.

The table in Figure 4.14 shows the power estimation results for these optimizations. The first column in the table denotes the transformations used to optimize the design. The entry “Nil” corresponds to the base design before optimization. The power estimated (not include glitch power) after applying each transformation sequentially is listed in the second column. This estimated power is only for the

Optimizations used	Estimated power (mW)
Nil	143
Common sub-expression reduction-1	112
Common sub-expression reduction-2	105
Constant propagation	104
Commutative rearrangement	127
Associative rearrangement	121

Figure 4.14: Results of power estimation due to combinational logic low power transformations in the Viterbi decoder. These estimates were based on the macro-modeling technique employed by Gupta *et al* [50]. These estimates are only over the trellis computation calculation function of the Viterbi decoder.

Design Configuration	Estimated power (mW)
Original Viterbi, clock_delay=10ns	416.37
Transformed Viterbi, clock_delay=10ns	354.33
Original Viterbi, clock_delay=20ns	208.41
Transformed Viterbi, clock_delay=20ns	177.17

Figure 4.15: Results of power estimation after sequential low power transformations in the Viterbi decoder.

GetMatrixSet block.

4.5.2 Sequential low-power optimizations

The sequential optimizations for low-power included clock-gating, some functional gating, and reorganizing pipeline registers. The optimized design is shown in Figure 4.12. The table in Figure 4.15 shows the power estimation before and after the low-power transformation. These numbers were obtained by using the Artisan TSMC 0.18um library. The Verilog designs were synthesized using Synopsys

Design Configuration	Estimated power (mW)
Original Viterbi, clock_delay=10ns	416.37
Transformed Viterbi, clock_delay=10ns	358.06
Original Viterbi, clock_delay=20ns	208.41
Transformed Viterbi, clock_delay=20ns	179.04

Figure 4.16: Results of power estimation after sequential optimization for low power and timing in the Viterbi decoder.

Design Compiler and the power estimated using Synopsys Power Compiler. The activity factors were kept as default, which is 1 per cycle for input signals and 2 per cycle for clock signal. We show the power estimation numbers for a clock delay of 10ns and 20ns.

4.5.3 Optimizations for power and timing

These optimizations were similar to the sequential lower power optimizations except that some aggressive power saving was sacrificed to lower the critical path delay time of the design. The optimized design is shown in Figure 4.13. The power estimation methodology was the same as in the case of the sequential optimizations and the estimated dynamic power dissipation is very close to the previous case. The table in Figure 4.16 shows the results.

4.5.4 Correctness of low power transformations on the Viterbi decoder

We start with the basic Viterbi decoder design, and three transformed designs, and construct the equivalence proof of each of the three transformed designs against

the basic design. In this subsection, we outline the proof of equivalence of the sequential low power transformation using our technique. We have four observables in this design:

- 8 FIFO entries, each 32-bits wide: $FF[7:0][31:0]$
- 64 Trellis Matrix entries, each 32-bits wide: $TM[63:0][31:0]$
- 2 entries in the MatDec, each 32-bits wide: $MD[1:0][31:0]$
- Decoded output, 32-bits wide: $Out[31:0]$

A pictorial representation of the proof of the first three observables is shown in Figure 4.17. For the sake of readability, we denote the observables in the original Viterbi design with a subscript o and the observables in the transformed design with a subscript p . The horizontal axis represents the data (observables), and the vertical axis shows the number of systems being compared (in our case, two). Time is represented along the axis normal to the plane of the paper.

We outline the proof methodology using our technique. In accordance with our algorithm in Figure 4.8 we follow the three stages. We first translate both designs into TRSs. The *execute()* function is represented by the horizontal arrows in the proof figure. We identify and obtain the expressions that need to be proved equivalent for each observable. Next, we use the dedicated rule database, and prove the equivalence of the expressions by rewriting. The vertical equivalence represents the *prove()* function. We had to add several rules to the dedicated rule database while

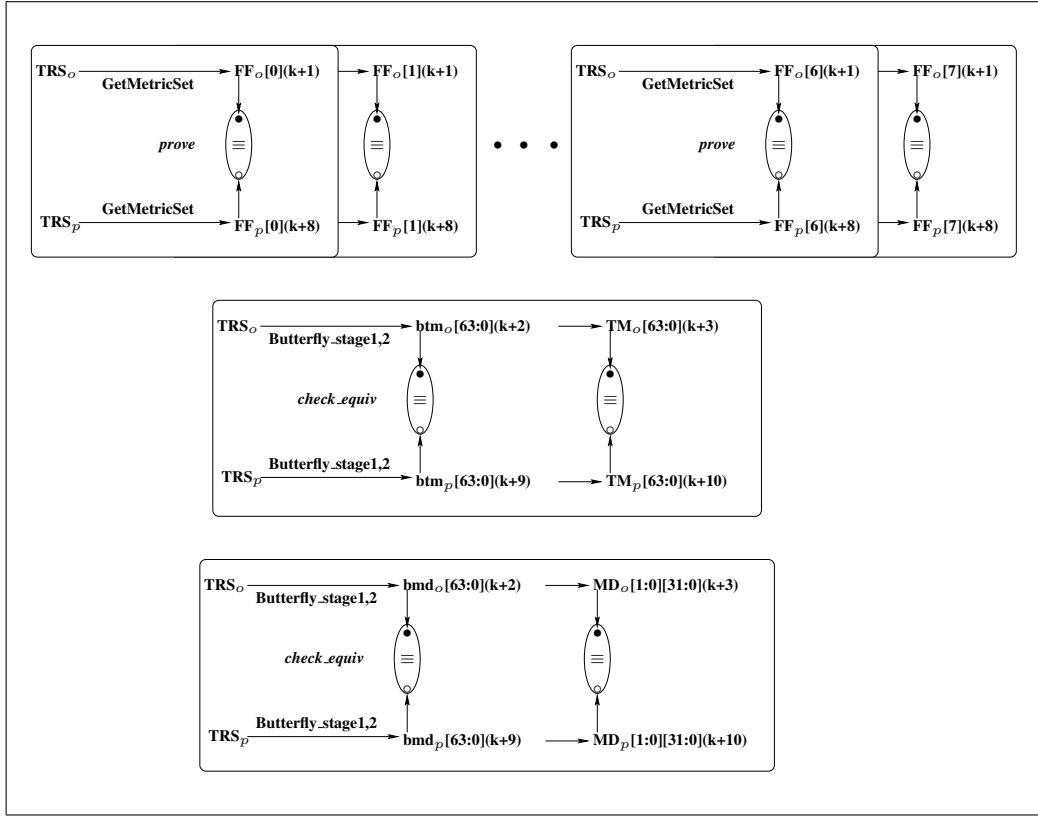


Figure 4.17: Picturization of sequential equivalence checking of transformed TRS_p against the original TRS_o of the Viterbi design. The first row is the proof of the FF buffer (over 8 time cycles). The center row shows the proof of the Trellis Computation and the bottom row shows the proof of the MatDec Decision Table.

we proved the correctness of this transformation. Similarly, we derive correctness proofs for the other transformations.

The first set of observables $FF[7:0][31:0]$ is available after 8 cycles, at the output of the FF Buffer. The first compare point, is therefore $C_1 = (t = 8, d = FF[7 : 0][31 : 0])$.

For each entry i in the FIFO buffer, the FIFO variables are $FF_o[i][31:0]$ and $FF_p[i][31:0]$. We call the *execute()* function at the compare point, and obtain the expressions for the FF variables.

In both the designs, the FF Buffer gets updated by the function *GetMetricSet()*. Therefore, the symbolic expressions correspond to an expansion using this function. The two symbolic expressions for $FF_o[i][31:0]$ and $FF_p[i][31:0]$ are checked by the *prove()* function. This procedure is repeated 8 times, for every entry in the FF Buffer, since each of them has a unique symbolic expression.

The next comparison point is obtained by stepping the two state machines of the designs after the 8^{th} cycle. The Verilog design assigns to the next observable at the 10^{th} cycle. The next observable is the Trellis Matrix, $TM[63:0][31:0]$. All the entries in this 64×32 matrix need to be checked, since the entire table is updated every 10^{th} cycle. The values of the MatDec decision table, $MD[1:0][31:0]$ is also updated in this cycle, as is the decoded output, $Out[31:0]$. The intermediate variable every 9^{th} cycle, b_{tm} which is not an observable is shown in lower case in Figure 4.17.

The second compare point, is therefore, $C_2 = (t = 10, d = TM[63 : 0][31 : 0], MD[1 : 0][31 : 0], Out[31 : 0])$.

The Trellis Matrix table gets its values from the 32 butterfly blocks in the design, each of which output 2 entries. The symbolic expression from the RTL, therefore, is a function of the butterfly blocks. For every 2 entries in the Trellis Matrix, the corresponding symbolic expression can be obtained from the butterfly. For instance,

$$TM_p[0], TM_p[1] = \text{Butterfly}(TM_p[0], TM_p[2], FF_p[0][31:0], FF_p[7][31:0])$$

and likewise in the original design,

$$TM_o[0], TM_o[1] = \text{Butterfly}(TM_o[0], TM_o[2], FF_o[0][31:0], FF_o[7][31:0])$$

Since $FF_p[0] = FF_o[0]$ from a previous comparison point C_1 , the symbolic expression for these signals are not expanded any further. The symbolic expressions for $TM_p[0]$, $TM_p[1]$ and $TM_o[0]$, $TM_o[1]$ are checked for equivalence again with the *prove()* function. This procedure is repeated 32 times, for every pair of entries in the Trellis Metric that need to be checked.

The other observables $MD[1:0][31:0]$ and $Out[31:0]$ are similarly checked for equivalence. The proof of $Out[31:0]$ is not shown in the figure.

4.6 Discussion and Conclusions

We have presented dedicated rewriting, a novel technique for proving correctness of low power transformations in RTL. Our technique uses Term Rewriting Systems and decomposes the problem into smaller and tractable proofs. The deductive nature of the rewriting system ensures that we do not encounter any size or capacity issues that are common in an algorithmic sequential formal verification engine.

It is key to note that our conversion of RTL to rules is not changing the abstraction level of the design description, it is merely a sideways representation change. Therefore the computational complexity of the general algorithm is still the same as sequential equivalence. However, what makes our technique computationally tractable is that the complexity of the verification problem which is size of the de-

sign in model checking methods, in our system is converted into the problem of incompleteness of the dedicated rule database. While this might lead to a highly interactive system in the general case, restricting the rule database to low power transformations, helps us leverage a higher degree of automation. Once the database captures a transformation (in the form of rules), then those rules work for any RTL and have high reuse value.

The primary advantage of this technique is that we reason with uninterpreted operators and bit abstractions at the RT-level, which is decidedly a more abstract (and therefore smaller) representation than reasoning at the gate-level. We have previously presented a specialized rewriter for arithmetic circuits, and this work largely generalizes the previous work in the context of low power transformations. Given the cost of re-validating hardware systems in a traditional design cycle, an automated technique of this nature is extremely desirable and can add immense value to the hardware design cycle.

Chapter 5

Holistic Power Management of SoCs using Dedicated Rewriting

5.1 Introduction

In today's systems, when SoCs are optimized for some applications and the optimizations are done in isolation without utilizing the knowledge of the workloads. Due to lack of hardware/software cooperation in power management, the platform as a whole cannot anticipate power requirements of the application ahead of time and instead, has to perform power management reactively.

Currently, platform power management broadly falls under two categories. At the one end are low power optimizations implemented by hardware designers like transistor level [31], gate level [94] and RTL optimizations [145], clock and power gating [151], optimizations to the processor pipeline, etc. These optimizations are done quite independent of the rest of the platform components, and in most cases, hardware is the best judge of what optimizations to use.

On the other end are the limited power management techniques that are available for the operating system (O/S) and devices to control. This includes O/S control of processor C-states and P-states and device power states [93], with standardized interfaces like ACPI (Advanced Configuration and Power Interface [62]) that reg-

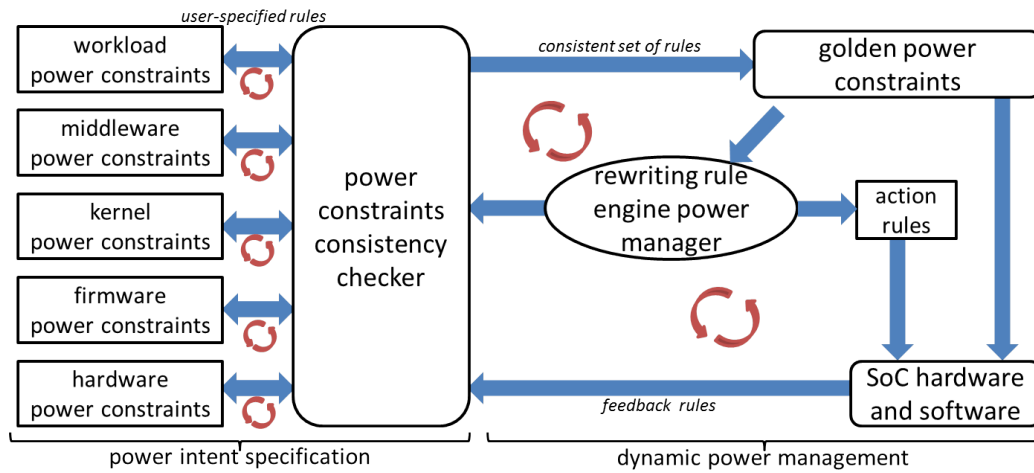


Figure 5.1: Rule-based formal Power Specification and Management

ulate such state control based on the workload and configured policies. In recent years, many platform level techniques for power management have emerged organically [14]. Both CPU and bus frequencies can be dynamically voltage- and frequency-scaled [156]. A key component of such a system which makes it most efficient as well as effective is the ability to control this dynamically, on-the-fly as it were, during application execution on the device.

It is imperative that a holistic platform level dynamic power management system be aware of (a) different power states supported by different components, both at architectural and micro-architectural level; (b) current power consumption of the platform as a whole, and at individual component level; (c) power requirements of applications and workloads; and, (d) continuous feedback from the platform on performance with respect to overall power constraints. The upshot of this is that we need a system which allows a common way to specify constraints and communicate

between them at different abstraction levels.

Figure 5.1 describes our system at a high level. We present a formal system to specify power constraints and power intent at every level of design hierarchy and abstraction. All specifications within this format are able to communicate with each other, and our power constraint consistency checker will also flag any conflicting constraints. Identifying a golden set of consistent power constraints is an iterative process through the checker. The power constraints at each level of design hierarchy captures the power intent along with any other constraints relevant at that level of design. Together the power constraints at each subsystem level capture a holistic power intent for the SoC.

We present a rewriting strategy and a Term Rewriting Systems based rule engine for dynamic power management. Given a power intent specification, and a well defined feedback from the system, our rewriting engine provides a way to dynamically resolve the feedback constraints against the input constraints, and instruct actions to the system for efficient power management.

The rest of the chapter is organized as follows. In Section 5.2 we define the notion of rules as relevant to dynamic power management. In Section 5.3 we explain our power constraints consistency checker which guarantees a golden set of consistent power rules. We give a detailed algorithm for dynamic power management in Section 5.4. In Section 5.5 we explain our experiments of dynamic power management on the state-of-the-art next generation Intel hand held device platform.

structural rules [feedback]:	
<code>_DB_Util</code>	<code>→ get_system_feedback (fabric, Util%) if (*)</code> (1)
<code>_fb_mem_BW</code>	<code>→ get_system_feedback (memory, BW%) if (*)</code> (2)
<code>_fb_dev_Util</code>	<code>→ get_system_feedback (soc , Util%) if (*)</code> (3)
...	
structural rules [user specified]:	
<code>_policy</code>	<code>→ performance if (*)</code> (4)
<code>_audio_device_state</code>	<code>→ less_than (D0, 25%)</code>
	<code>if (_policy = performance)</code> (5)
<code>_soc_device_state</code>	<code>→ more_than (S0ix, 80%)</code>
	<code>if (_policy = performance)</code> (6)
...	
structural rules [actions]:	
<code>_audio_device_state</code>	<code>→ ON if (_policy = performance)</code> (7)
<code>_display_device_state</code>	<code>→ OFF if (_policy = performance)</code> (8)
<code>_cpu_device_state</code>	<code>→ P_n if (_policy = performance)</code> (9)
...	
logical rules:	
<code>less_than (x, y)</code>	<code>→ T if (x_{numeric} < y_{numeric})</code> (10)
<code>less_than (x, y)</code>	<code>→ ⊥ if (x_{numeric} ≥ y_{numeric})</code> (11)
<code>more_than (x, y)</code>	<code>→ T if (x_{numeric} > y_{numeric})</code> (12)
<code>more_than (x, y)</code>	<code>→ ⊥ if (x_{numeric} ≤ y_{numeric})</code> (13)
...	

Figure 5.2: Audio player structural and logical rules

5.2 Rules

We have adapted our rule-based system *dedicated rewriting* system for specifying and managing power constraints as well as for dynamically managing system level power consumption (Figure 5.1). Our rules follow the general structure of rewriting rules in a Term Rewriting System (TRS). This helps us to leverage the formalism that goes with a TRS to uniformly and uniquely represent power specification across all levels of design hierarchy. For RTL-level and hardware constraints, we allow a way to convert UPF representations into TRS rules. We also capture

power policies and expected system behavior as rules. This sets up the stage for our two-stage rewriting process. The first stage checks consistency of the power constraints and flags any conflicts. This is an iterative user-driven process and the end result is a golden set of non-conflicting constraints. The next stage is the dynamic power manager, which outputs action rules to the system to maintain the system behavior within the expected power policy behavior. We describe both these stages of rewriting in subsequent sections. In this section, we take a more detailed look into the different types of rules within our system.

Applying a rule is also called executing or firing. When a rule is applied, the state on the left-hand-side is read at the beginning of a power tick and updated at the end of the power tick. We define a *power tick* as any power related event. This could be when an O/S call is triggered because of an interrupt, or a power state change kicks in, or any event that changes the normal state of the input constraints or system feedback constraints. A power tick is any change in power state, and the system run-time semantics will ensure there are no overlapping ticks. This notion of a single rule execution enforces an atomicity constraint on the system at each power tick. In the context of rules, this says that during the execution of a rule, no other events happen in the system, to invalidate the action described in that rule.

All enabled rules fire in parallel (or in no particular order). If two rules modify the same state or power element, then we have a race condition. We have a notion of user-specified priority ordering of rules. During a race condition, the rule with a higher priority is chosen. If no priority is specified, a random choice is made.

We have two kinds of rules within our system, structural rules and logical rules.

Structural rules are atomic transactions representing a power state update in the system. These are derived directly from the system architecture, from both hardware as well as software. Logical rules represent identities about the operators used in the structural rules and carry information about power specific transformations. Figure 5.2 shows the different kinds of rules in our system.

Structural rules can be feedback rules, user-specified ones, or actions. The user-specified rules define power management policies (see Rules (4)..(6) in Figure 5.2). These are static, and are defined prior to running the system. Feedback rules are what come from the system into the rewriting engine (see Rules (1)..(3) in Figure 5.2). These rules define the status of various system attributes that govern power policy management. These rules dynamically fire given system behavior during run-time. Action rules define outputs from the rule engine back into the system (see Rules (7)..(9) in Figure 5.2). These define how the system should behave based on the conclusions of the power management rule engine. Logical rules codify identities about operators and low power transformations and policies (see Rules (10)..(13) in Figure 5.2). These rules are independent of the target system. We can use logical rules to create abstractions, model existential/universal quantification and interpret uninterpreted functions, etc.

Rules are powerful representations of power policy and power consumption intent. Precedence/ordering and atomicity of rules together form a provably complete specification of power intent. The user provides more rules where they are insufficient, and precedence decides order of application. Once the “training” period is done, the set of rules forms a provably complete power intent specification. For

large SoCs, managing power constraints at all levels of design hierarchy can be a veritable nightmare. The *dedicated rewriting* system manages power constraints and does some automatic syntax and semantic consistency check of power constraints across all levels of design hierarchy in a user-driven iterative process (Section 5.3). Given a precise and conflict-free golden power intent and rules about power policies, our rewriting engine dynamically fires all triggered rules at each power tick, and generates actions for the system. We describe the adaptation of the *dedicated rewriting* algorithm in Section 5.4.

5.3 Power Constraints Consistency Checker

Managing power constraints is a hard enough problem just at the netlist level. In order to be able to manage power constraints at all levels of design abstraction and hierarchy, we need both a uniform representation, as well as an automated checking mechanism to keep the constraints syntactically correct and semantically consistent.

The left-hand-side of Figure 5.1 gives a flow chart of our constraint checking process. We bring in the constraints at different levels of hierarchies all within the same rule formats. In situations where hardware constraints are written in other representations like UPF, we provide a way to convert them into our rules. The constraint checker has a two-fold function. First, it does a *power-constraint-lint*, checking for syntactic correctness of the constraints. Next, it does a *power-constraint-consistency-check*, checking for semantic consistency across constraints, as detailed in Algorithm 1. The complexity of the power constraint checker lies in understanding the interactions between constraints at different levels of design hierarchy and

Algorithm 1: Power Constraints Consistency Checker

Input: power constraints at all levels of hierarchy $\$pcdb$,
Rule ordering in case of race condition RO

Output: Database of golden power constraints $\$pcdb_g$

$lintResult = \emptyset$; $consistencyResult = \emptyset$;

for each power constraint pc in $\$pcdb$ **do**
 $lintResult := power_constraint_lint(pc)$;

for each power constraint pc_i in $\$pcdb$ **do**
 if ($pc \neq pc_i$) **then**
 $consistencyResult := power_constraint_consistency_check$
 (pc, pc_i);
 end
 end
end

if ($lintResult == \emptyset \ \&\& \ consistencyResult == \emptyset$) **then**
 $\$pcdb_g = \$pcdb$; **return** $\$pcdb_g$;

end
else
 return ($lintResult \cup consistencyResult$);
end

proving that they are consistent with each other.

Checking for power constraint consistency is a user-driven iterative process. The worst-case runtime of this algorithm is $O(n^2)$, which can be prohibitively expensive if n , the number of constraints is very large. In practice, even for a 1 million gate design, the number of constraints is within manageable numbers, so that the iterative process of consistency checking has sufficiently small turn-around times. The end result of this process is a golden set of power constraints which are syntactically correct and semantically consistent with one another. This now forms an input to our dynamic power management rule engine.

5.4 Dedicated Rewriting as a Dynamic Power Management Rule Engine

We propose a refinement based rewriting strategy for dynamic power management, a working flow chart of which is show in the right-hand-side of Figure 5.1. This is a modified implementation of our *dedicated rewriting* system, applied to this context of holistic power management.

The database of logical as well pre-existing (from previous runs or projects) rules is statically available as an input. Dynamic power management is an outcome of which of these rules fire during run-time, based on system feedback. Also available as input, is a rule ordering, to decide precedence if more than one rule modifying the same element is triggered at the same power tick.

The process starts with loading the static list of rules into the rewrite engine. Each iteration of the flow is defined by a power tick and involves two key phases. In the first phase, the rule engine resolves the feedback rules and arrives at a set of action rules. If there are conflicts, then in the second phase, the user has to resolve them and help the engine move along. This learning process is done once to train the engine and obtain an exhaustive set of rules required to run the engine for this particular system architecture. Conflicts are resolved typically by adding further rules to help move along the rewriting process. The process is described in Algorithm 2.

Our rule rewriting engine, for the purpose of an automatic, dynamic power manager, is essentially a constraint solver. The *dedicated rewriting* system is generic

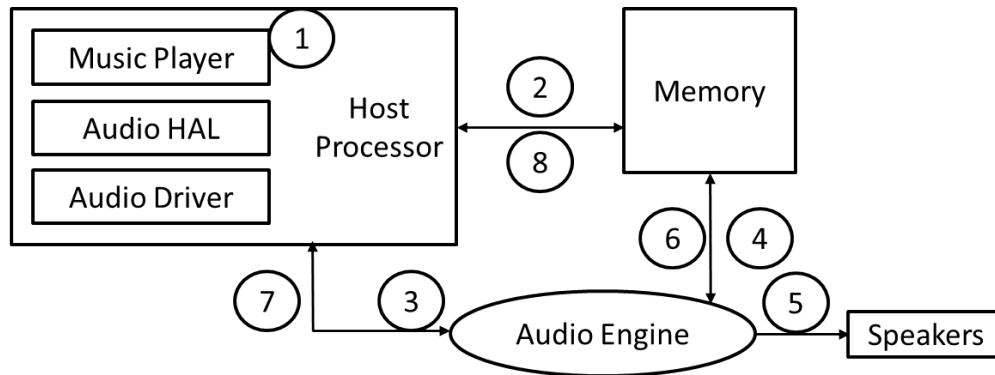
Algorithm 2: Power Management Rewriting Engine

Input: Database of logical and power rules $\$db$,
Rule ordering in case of race condition RO
Output: Action Rules driving the system $R_{actions}$
for *each power tick* /* change in any power state */ **do**
 $execResult = FAIL$;
 repeat
 $T_U = read_from_db (\$db, user_spec)$;
 $T_F = read_from_db (\$db, system_feedback)$;
 $T_L = read_from_db (\$db, logical)$;
 $T_P = read_from_db (\$db, power)$;
 $\langle execResult, T_A \rangle = execute (T_U, T_F, T_L, T_P)$;
 if ($execResult == FAIL$) **then**
 Add rules to $\{\$db:T_L, \$db:T_L, \$db:T_L, \$db:T_L\}$
 end
 if ($execResult == PASS$) **then**
 Add T_A to $R_{actions}$ based on order RO
 end
 until $execResult != PASS$;
end

enough to accommodate the change of context from low power hardware transformation rules, to these holistic power constraint rules.

5.5 Case Studies

We will now illustrate how both the constraint checker and the rule rewriting manager can be applied for a typical hand held mobile device in three distinct contexts: audio playback, video playback, and web browsing. We have implemented the rewriting rule engine on one of the latest Intel smartphone reference platforms. This smartphone SoC is a fully state of the art platform, built on the latest process



1. User selects a music list
2. Host Processor loads music to Memory
3. Host Processor notifies Audio Engine to process data in Memory
4. Audio Engine fetches music file from Memory
5. Plays music. Rest of platform is in DEEPSLEEP state
6. Audio Engine refills its buffer by partially waking Memory
7. Audio Engine wakes up Host Processor when Memory runs out of data
8. Host Processor loads next chunk of data in Memory

Steps 3—8 repeat as long as media playback continues or user interrupts the process.

Figure 5.3: Low power audio playback policy

technology and supports multiple power management hooks in hardware, firmware, and software.

Typically, in most smartphone and/or tablet platforms there is an application or host processor that is used for general purpose processing, while functions such as graphics, video encoding/decoding, audio playback, etc. are offloaded to highly specialized low power processor cores/accelerators.

5.5.1 Audio Playback

Figure 5.3 describes the Audio playback low power policy. We have implemented this policy as rules in our rewriting system. A subset of these rules can be found in the earlier example in Figure 5.2. The key underlying concept in this policy is that when the user is listening to music, we can buffer the audio data, keep the audio subsystem on, and put the rest of the SoC to sleep. We started off translating the policy specification into user-specified constraints, feedback rules from the system, and action rules to the system.

Constraints for audio playback are to describe the resource we would need to have this use case running and the quality of service the workload can tolerate. Given the above data flow, the audio playback use case can be represented in terms of constraints capturing

- Quality of service that we need to guarantee from the system.
- Host processor and Audio processor utilization time and minimum frequency they need to run at to meet the minimum quality of service.
- During playback, only the low power audio engine must be kept ON, while rest of the SoC should be in DEEPSLEEP state.
- Status of various subsystem while audio is playing.

Over a few iterations of the constraint checker we identified a conflict-free golden set of power constraints. Over a further few iterations of the rewriting engine we

figured out the relevant logical and power rules to embed in the database. Once these iterations are done, the system is ready for mainstream operation.

5.5.2 Panel Self Refresh

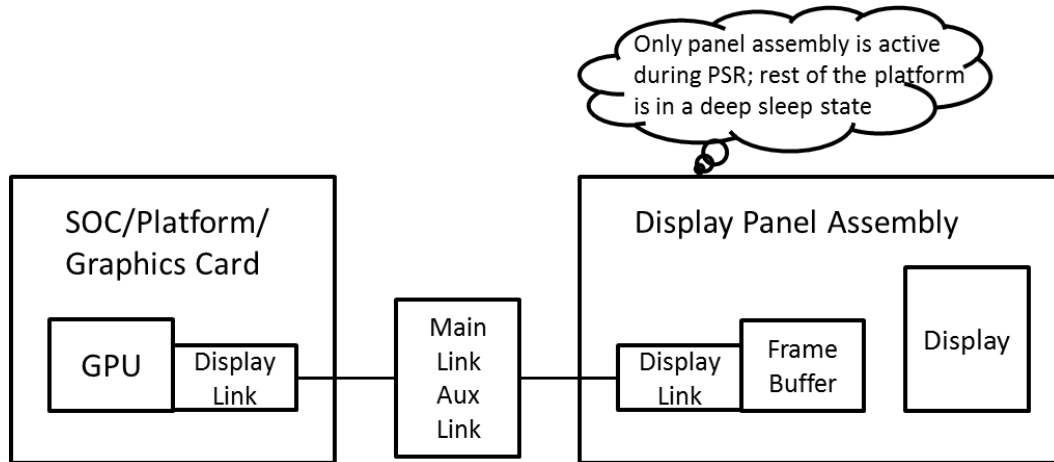


Figure 5.4: Panel self refresh policy for video playback and web browsing applications

Like the audio playback policy, a similar power policy called *panel self refresh* (PSR) is employed in the display subsystem. Figure 5.4 shows the low power policy of a PSR system. PSR is a combination of functionality on the chip, display, and software, which allows screens to save power by only refreshing the image on the screen when it changes. If the user is reading an eBook or browsing the web, odds are that most of the time, the screen is a set of static images for a few seconds at a time. Basic analysis [63] indicates that for most usages like web browsing, reading PDFs, document editing, etc., the idle frame percentages are in the range of 85-96%.

5.5.3 Results

We have implemented our rule rewriting engine in the above three contexts, *viz.*, audio playback, PSR for video playback, and PSR for web browsing. The power gains achieved by the automated dynamic power management of these policies is shown in Figure 5.5. For the audio playback case, when our rewriting engine is absent, the measured power is high primarily because the SoC and Display are both ON even when the only action going on is an audio playback. Our dynamic power manager generates the necessary actions to reduce that power consumption to attain a significant gain of $\sim 43\%$. The panel self refresh gains are somewhat similar in both the video and browser cases with the power consumption down by over a third. These measurements are done on the latest generation Intel smartphone platform.

We have also implemented the same rewriting rule engine system on a many-core platform, modeling the power constraints for an O/S scheduling policy. The considerations in this context are very different from an SoC context, and with our dynamic manager we achieve a 10% gain in CPU utilization, which translates to a linear gain in power consumption [146].

5.5.4 Discussion

The first stage of each experiment is to arrive at a golden set of conflict-free power constraints. We have used our automated checker to assist in each iteration of this user-driven process. As power optimizations increase at each level of design hierarchy, number of constraints dramatically increase, and a robust constraint manager is key to delivering a conflict-free golden power constraint set.

Experiment	Power Gain%
Audio Playback (SoC OFF; Display OFF)	43%
Browser Self-Refresh (only Display panel ON)	35%
Video Self-Refresh (only Display panel ON)	39%
O/S Scheduler (cores are shielded when offline)	10%

Figure 5.5: Summary of results without and with dynamic power management by the rewriting rule engine

In all these experiments, there is an initial cost of having the power management rewriting engine learn the policies iteratively and refine its rule database for each policy. However, addressing that incompleteness is a one-time cost, and in our experience, a small cost, because we are learning the rules for a small specific policy each time. In all subsequent runs, the dynamic rule execution will execute automatically and exactly as per the power intent specified in the policy.

These two engines, the checker and the power manager, are completely independent of each other, even though the former feeds into the latter. In a different power management system, one could easily imagine using the checker alone, and converting its output into constraints as understood by the power manager of that system. The checking is a static process done before actual execution, while the power manager is a real-time tool, working during run-time of the SoC.

Chapter 6

Correctness of Instruction-driven Slicing

6.1 Introduction

In Chapter 3 we introduced Instruction-driven slicing as a low power transformation at the RT-level to lower dynamic power dissipation in microprocessors. In Chapter 4 we introduced *Dedicated Rewriting* as an automatic technique to generate proofs of correctness of low power transformations in hardware, also at the RT-level. In this chapter we will discuss the application of *dedicated rewriting* to *instruction-driven slicing*. We will give an automatic proof of *instruction-driven slicing* in our framework.

We describe within our rewriting methodology, how we use our dedicated database of rules to automatically prove the correctness of low power transformations at the RT-level. We use the example of instruction-driven slicing [144] as applied to a 32-bit OpenRISC pipelined microprocessor (OR1200) [38] to explain our technique [149], [148]. We also give a proof for the same problem in a general purpose theorem prover, ACL2 [67], and contrast our dedicated rewriter against using a general purpose proof system.

The technique was already described in detail in Chapter 3. We present the proof of *instruction-driven slicing* on OR1200 in Section 6.2. Section 6.3 explains

the proof in the general purpose theorem prover ACL2 and contrasts our technique against a more generic approach. We discuss the merits of such a framework and conclude in Section 6.4.

6.2 Automatic Proof Technique

We described *dedicated rewriting* in Chapter 4. Here we give an instance of its use to prove the correctness of *instruction-driven slicing*. This is a one-time activity since *instruction-driven slicing* is correct by construction, and does not need to be repeated for each application of *instruction-driven slicing*.

Figure 6.1 shows the Verilog RTL before and after applying *instruction-driven slicing**. Figure 6.2 and Figure 6.3 show the rules derived from the Verilog RTL code of the OR1200 microprocessor corresponding to Figure 6.1. The block in Figure 6.2 is the rules defining the original RTL, and the block in Figure 6.3 is the rules of the transformed RTL after automatic insertion of low power annotation by instruction-driven slicing.

The *dedicated rewriter()* takes the two expressions and tries to rewrite them to show they are equal. In order to assist the rewriter, we provide a dedicated set of low power rules which are generic to all low power transforms. If the rewriter is not able to prove the equivalence (defined as, rewrite the XNOR of the two expressions to TRUE), then we analyze the result. The output of the dedicated rewriter could be one of two choices. If it is a false negative, then our rule database lacked the

*This figure is the same as Figure 3.4 and has been reproduced here for the convenience of the reader.

```

// Instruction selection in load/store unit

always @(posedge clk or posedge rst) begin
  case (id_insn[31:26])
    `OR1200_OR32_SB: lsu_op <= #1 `OR1200LSUOP_SB;
    `OR1200_OR32_SW: lsu_op <= #1 `OR1200LSUOP_SW;
    `OR1200_OR32_LBZ: lsu_op <= #1 `OR1200LSUOP_LBZ;
    `OR1200_OR32_LWZ: lsu_op <= #1 `OR1200LSUOP_LWZ;
    default: begin lsu_op <= #1 `OR1200LSUOP_NOP;
  endcase
end

```

(a) Verilog RTL code for the `always` block assigning the `lsu_op` before instruction-driven slicing transformation.

```

// Instruction selection in load/store unit sliced on
// instruction l.addc

always @(posedge clk or posedge rst) begin
  if (iADDC_id)
    lsu_op <= #1 `OR1200LSUOP_NOP;
  else
    case (id_insn[31:26])
      `OR1200_OR32_SB: lsu_op <= #1 `OR1200LSUOP_SB;
      `OR1200_OR32_SW: lsu_op <= #1 `OR1200LSUOP_SW;
      `OR1200_OR32_LBZ: lsu_op <= #1 `OR1200LSUOP_LBZ;
      `OR1200_OR32_LWZ: lsu_op <= #1 `OR1200LSUOP_LWZ;
      default: begin lsu_op <= #1 `OR1200LSUOP_NOP;
    endcase
end

```

(b) Transformed Verilog RTL code after applying instruction-driven slicing on instruction `l.addc`.

Figure 6.1: Instruction-driven slicing example from Figure 3.4 reproduced here for the reader's convenience.

```

// Instruction selection in load/store unit

Rule: lsu_op(t) -> `OR1200_LSUOP_SB
      if (id_insn[31:26](t) == `OR1200_OR32_SB)
Rule: lsu_op(t) -> `OR1200_LSUOP_SW
      if (id_insn[31:26](t) == `OR1200_OR32_SB)
Rule: lsu_op(t) -> `OR1200_LSUOP_LBZ
      if (id_insn[31:26](t) == `OR1200_OR32_SB)
Rule: lsu_op(t) -> `OR1200_LSUOP_LWZ
      if (id_insn[31:26](t) == `OR1200_OR32_SB)
Rule: lsu_op(t) -> `OR1200_LSUOP_NOP
      if (id_insn[31:26](t) !=
          (`OR1200_OR32_SB
           or `OR1200_OR32_SW
           or `OR1200_OR32_LBZ
           or `OR1200_OR32_LWZ))

```

Rules derived from Verilog RTL code for the `always` block assigning the `lsu_op` before instruction-driven slicing transformation.

Figure 6.2: Rules derived before instruction-driven slicing transformation.

required rules. In which case, we add that rule to the database, and repeat the rewriting step. If the output is a true error, then we output a counter example, a trace to reach the error starting from the inputs over a specific period of time.

In the example of Figure 6.3, one of the rules that is specific to the transformation, which we have to add to the rule database is

```

Rule: iADDC_id(t) -> T
      if (id_insn[31:26](t) ==
          `OR1200_OR32_ADDC)

```

```

// Instruction selection in load/store unit sliced
on instruction l.addc

Rule: lsu_op(t) -> `OR1200_LSUOP_NOP
      if (iADDC_id(t) == T)
Rule: lsu_op(t) -> `OR1200_LSUOP_SB
      if ((id_insn[31:26](t) == `OR1200_OR32_SB)
          and (iADDC_id(t) == F))
Rule: lsu_op(t) -> `OR1200_LSUOP_SW
      if ((id_insn[31:26](t) == `OR1200_OR32_SW)
          and (iADDC_id(t) == F))
Rule: lsu_op(t) -> `OR1200_LSUOP_LBZ
      if ((id_insn[31:26](t) == `OR1200_OR32_LBZ)
          and (iADDC_id(t) == F))
Rule: lsu_op(t) -> `OR1200_LSUOP_LWZ
      if ((id_insn[31:26](t) == `OR1200_OR32_LWZ)
          and (iADDC_id(t) == F))
Rule: lsu_op(t) -> `OR1200_LSUOP_NOP
      if ( (iADDC_id(t) == F) and
          (id_insn[31:26](t) != (`OR1200_OR32_SB
                                or `OR1200_OR32_SW
                                or `OR1200_OR32_LBZ
                                or `OR1200_OR32_LWZ)))

```

Rules derived from the transformed Verilog RTL code after applying instruction-driven slicing on instruction l.addc.

Figure 6.3: Rules derived after instruction-driven slicing transformation.

The low power rule database is a continuously growing entity by step-wise refinement with every proof attempted in this system.

Even though refining the low power rules database is an iterative process, we need to do it once for the transformation. Subsequently, any RTL using the same

transformation can be verified without adding any further rules. A possible design methodology could be to implement a new transform on a very small example RTL and obtain the proof of correctness using our technique. This exercise will populate the low power rule database with any rules that are specific to the current transformation in question. Subsequently the transform can be applied to the larger designs and this time around the proof will be automatic without requiring any refinement steps. Also, the refinement step in the first case is a very powerful way to understand the logical implications of the transform.

We have proved the correctness of the instruction-driven slicing algorithm on the OR1200 microprocessor Verilog RTL using our technique. In the next section we give another proof for the same using a generic theorem prover, and contrast that with the automatability of our technique.

6.3 Interactive Proof by Deductive Verification

We have proved the correctness of the instruction-driven slicing algorithm on the OR1200 example using the ACL2 theorem prover. We give a proof that the functionality of the OR1200 processor before and after the low power annotations is precisely the same. This guarantees the functionality preserving property of the low power annotations.

6.3.1 Proof using the ACL2 theorem prover

We use the ACL2 theorem prover to establish our proofs. ACL2 is both a first-order mathematical logic and a mechanical theorem prover to reason and prove the-

orems about functions in this logic. The language is based on Applicative Common Lisp and the theorem prover is an *industrial strength* version of the Boyer-Moore theorem prover, Nqthm [18].

The ACL2 theorem prover is a computer program which takes formulas written in first order logic and tries to find mathematical proofs. It uses rewriting, decision procedures, mathematical induction, and many other proof techniques to prove theorems in a first-order mathematical theory of recursively defined functions and inductively constructed objects [66].

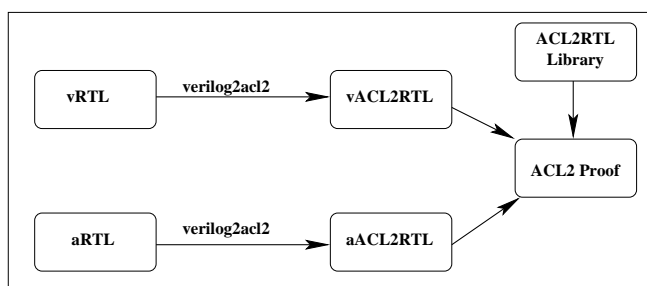


Figure 6.4: Proof methodology.

Figure 6.4 explains our proof methodology. We convert the two RTLs into ACL2 functions. We also create a large library of functions which are used to interpret the RTL functions. You can consider this library as an RTL (in our case, Verilog) definition library. Creating this library is a high-effort task. Admittedly though, once we create this library, it should work for any Verilog RTL. After this is an iterative and interactive procedure of working with the ACL2 prover system, adding and defining more rules and theorems, until we are able to obtain a proof.

We have implemented *verilog2acl2*, a compiler from Verilog RTL to ACL2 functions. The annotations are added to the original RTL (vRTL) as described in Subsection 3.3.1 to obtain the annotated RTL (aRTL). We then obtain the corresponding ACL2 models vACL2RTL and aACL2RTL.

Figure 6.5 shows an example function in ACL2 corresponding to an `always` block in Verilog. The ACL2 model of the RTL is an executable model with the following top function signature:

```
(defun or1200_cpu (n) ...)
```

Every ACL2 function is modeled with the clock cycle n as the input argument. We have created a large ACL2RTL library to interpret the automatically generated ACL2 RTL model. In Figure 6.5(b), the functions `bif`, `bv-and`, `cw-d` *etc.* are functions within our ACL2RTL library.

Figure 6.6 and Figure 6.7 show the converted ACL2 function for the same example in Figure 6.1.

Once we have our two ACL2 models and a library of functions to interpret these models, we can use ACL2 to reason about these two models and prove the theorem that they both are functionally equivalent. We do this by building up the proof block by block in a bottom-up fashion. The theorem we prove is:

```
(defthm iADDC_slicing_correct
  (equal
    (or1200_cpu n)
```

```

//Instruction latch in ex_insn
always @(posedge clk or posedge rst) begin
    if (rst)
        ex_insn <= #1 `OR1200_OR32_NOP, 26h041_0000;
    else if (!ex_freeze & id_freeze | flushpipe)
        // ex_insn[16] must be 1
        ex_insn <= #1 `OR1200_OR32_NOP, 26h041_0000;
    else if (!ex_freeze) begin
        ex_insn <= #1 id_insn;
    end
end

```

(a) Verilog RTL code for latching an instruction in the Execute stage.

```

(defun or1200_ctrl_ex_insn (n)
  (bif (or1200_ctrl_rst n)
    (concat (OR1200_OR32_NOP)
      (bv-<< (cw-d 26 65) (cw-d 6 16))))
  (bif (bv-and
    (logical-not (or1200_ctrl_ex_freeze n))
    (bv-or (or1200_ctrl_id_freeze n)
      (or1200_ctrl_flushpipe n))))
    (concat (OR1200_OR32_NOP)
      (bv-<< (cw-d 26 65) (cw-d 6 16)))
  (bif (logical-not (or1200_ctrl_ex_freeze n))
    (or1200_ctrl_id_insn (1- n))
    (concat (OR1200_OR32_NOP)
      (bv-<< (cw-d 26 65) (cw-d 6 16))))))

```

(b) ACL2 RTL function corresponding to the Verilog always assignment block.

Figure 6.5: *Verilog2ACL2* example.

```

(defun or1200_lsu_dcpu_sel_o (n)
  (cond ((equal (concat (or1200_lsu_lsu_op n)
    (getbits 1 0 (or1200_lsu_dcpu_adr_o n)))
    (concat (OR1200_LSUOP_SB)
      (cw-d 2 0))) (cw-d 4 8))
    ((equal (concat (or1200_lsu_lsu_op n)
    (getbits 1 0 (or1200_lsu_dcpu_adr_o n)))
    (concat (OR1200_LSUOP_SW)
      (cw-d 2 0))) (cw-d 4 15))
    ((equal (concat (or1200_lsu_lsu_op n)
    (getbits 1 0 (or1200_lsu_dcpu_adr_o n)))
    (concat (OR1200_LSUOP_LBZ)
      (cw-d 2 0))) (cw-d 4 8))
    ((equal (concat (or1200_lsu_lsu_op n)
    (getbits 1 0 (or1200_lsu_dcpu_adr_o n)))
    (concat (OR1200_LSUOP_LWZ)
      (cw-d 2 0))) (cw-d 4 15))
    (t (cw-d 4 0))))

```

ACL2RTL code for the `always` block assigning the `lsu_op` before instruction-driven slicing transformation.

Figure 6.6: Instruction-driven slicing example after *Verilog2ACL2* (pre-transformation).

```

(or1200_cpu_sliced_for_iADDC n))

```

This theorem guarantees that, for `l . addc` instruction, the original ACL2 model is functionally equivalent to the annotated ACL2 model. The variable `n` above is a free variable, quantified over all time. The model is combinational and the ACL2 proof is a simple equivalence checking proof and does not require any induction.

```

(defun or1200_lsu_dcpu_sel_o (n)
  (bif (iADDC_id n) (cw-d 4 0)
    (cond ((equal (concat (or1200_lsu_lsu_op n)
      (getbits 1 0 (or1200_lsu_dcpu_adr_o n)))
      (concat (OR1200_LSUOP_SB)
        (cw-d 2 0))) (cw-d 4 8))
      ((equal (concat (or1200_lsu_lsu_op n)
        (getbits 1 0 (or1200_lsu_dcpu_adr_o n)))
        (concat (OR1200_LSUOP_SW)
          (cw-d 2 0))) (cw-d 4 15))
      ((equal (concat (or1200_lsu_lsu_op n)
        (getbits 1 0 (or1200_lsu_dcpu_adr_o n)))
        (concat (OR1200_LSUOP_LBZ)
          (cw-d 2 0))) (cw-d 4 8))
      ((equal (concat (or1200_lsu_lsu_op n)
        (getbits 1 0 (or1200_lsu_dcpu_adr_o n)))
        (concat (OR1200_LSUOP_LWZ)
          (cw-d 2 0))) (cw-d 4 15))
      (t (cw-d 4 0))))))

```

Transformed ACL2RTL code after applying instruction-driven slicing on instruction `l.addc`.

Figure 6.7: Instruction-driven slicing example after *Verilog2ACL2* (post-transformation).

6.3.2 Comparing a dedicated rewrite system versus a generic theorem prover

A general purpose theorem prover has many uses. However, it requires a highly trained individual to interact with the program and help the deduction process. In our dedicated rewrite system, we have managed to focusedgenerate a sufficiently large database of rules that are on specific transformations. Now, for those transformations, we have an automatic solution. Besides, we also have the methodology

built into our system to enhance these rules to incorporate new transformations. The overall ease of having an automatic tool, and the fact that it works with Verilog and easily and seamlessly fits in with the design tools, makes our technique easy to incorporate in a hardware design cycle.

6.4 Discussion and Conclusions

We have proposed a step-wise refinement based dedicated rewriting engine to automatically prove the correctness of low power transformations at the RT-level. We have also proved the correctness of the instruction-driven slicing transformation on the OR1200 Verilog RTL. At the gate level, the low power transformations are not across flop boundaries, and combinational equivalence checking is sufficient to prove the correctness of the transformations. However, at the RT-level, the transformations are more behavioral and can utilize the semantic information available to reduce further power dissipation. We provide a way to verify the correctness of the transformations at higher levels of abstraction. Our rules and rewriting is not restricted to the RT-level either. Any architectural model can also be used instead of RTL, as long as a *derive()* function is also provided to interface the model into our rules system.

Chapter 7

Discussion and Conclusions

Designing for low power in the context of today's devices is a holistic exercise. There are optimizations in the hardware which are software agnostic, and vice versa, and there are a host of techniques which need the hardware to be exposed to what the software intent is. In order to achieve the most optimal set of techniques applied dynamically we need an overall power management framework, with its key components being the four aspects we've identified, *viz.* specification, modeling, techniques and verification.

The key problem in low power, in today's shrinking hardware designs, is that of coming up with effective high level low power transformations, and verifying the correctness of such transformations, as applied to a specific hardware, in minimal time. We have addressed this problem in this thesis in two parts.

First, we have proposed *instruction-driven slicing*, a new technique to automatically annotate RTL for reducing power dissipation by switching activity. We have implemented the *instruction-driven slicing* algorithm and have incorporated it into the design flow tool-chain. We have automatically sliced the RTL and architectural models for OR1200, a pipelined implementation of the OpenRISC instruction set architecture and for PUMA, a PowerPC dual-issue, out-of-order, superscalar fixed

point unit. We have used our tool-chain to test our methodology on this processor and have obtained encouraging results.

Second, we have presented dedicated rewriting, a novel technique for proving correctness of low power transformations in RTL. Our technique uses Term Rewriting Systems and decomposes the problem into smaller and tractable proofs. The deductive nature of the rewriting system ensures that we do not encounter any size or capacity issues that are common in an algorithmic sequential formal verification engine. However, the complexity of the verification problem in our system is converted into the problem of incompleteness of the dedicated rule database. While this might lead to a highly interactive system in the general case, restricting the rule database to low power transformations, helps us leverage a higher degree of automation. Once the database captures a transformation (in the form of rules), then those rules work for any RTL and have high reuse value.

Our *instruction-driven slicing* algorithm is particularly suited for in-order pipelined processor designs. It can be applied to out-of-order superscalar processors too. However the reduction in power in the case of PUMA was expectedly substantially less than the OR1200 case since there might be multiple instructions in flight in any pipeline stage of the PUMA, thereby reducing the amount of logic we can actually shut off. Although our algorithm is conservative, it automatically identifies a close-to optimal set of flops. Our instruction-driven slicing algorithm can be thought of as a wrapper to implement more sophisticated methods of identifying flops which control the circuitry outside the slice.

The primary advantage of the *dedicated rewriting* technique is that we reason

with uninterpreted operators and bit abstractions at the RT-level, which is decidedly a more abstract (and therefore smaller) representation than reasoning at the gate-level. We have presented a specialized rewriter for arithmetic circuits, and a more generalized version in the context of low power transformations. Given the cost of re-validating hardware systems in a traditional design cycle, an automated technique of this nature is extremely desirable and can add immense value to the hardware design cycle.

We have presented a logically sound formal engine as a backbone for managing power constraints at different levels of design hierarchy and abstraction; and for dynamic system level power management. We have presented our power constraint consistency checker as a way to holistically deal with constraints at different levels, and check for syntactic correctness and semantic consistency. Such a uniform and formalized representation is a big piece of the overall holistic power management system. As we build smaller and faster hand-held devices, power constraints will only get more complicated, and a user-driven power constraint manager with built-in automated checking capability will be central to any low-power SoC design.

We have also presented a rewriting strategy which localizes the problem to the domain of its action. This is a key insight into the power management technique. While the general complexity of the problem still remains NP-hard, in the localized context, the problem of resolving the rules and constraints is tractable in practice. We have also implemented some power optimization and management policies in our system on state-of-the-art Intel smartphone platform. We have shown power gains in the range of 40% for different experiments (policies). These are significant

gains, and to be able to do them automatically through a dynamic manager is a significant step further in delivering low power hand-held mobile devices.

Verification is the hardest piece of the power management puzzle. The current systems are quite ad-hoc and work in very specific contexts, while it is not clear how to generalize most of them. What we will need is a comprehensive methodology where we can define verification testplans in the context of power specification of the system, define coverage metrics and assertions, generate formal and simulation based guarantees about the asserted properties, etc. Further, a lot of these will need to be automated, given the extensive use of multiple voltages and multiple domains.

Low power designs are here to stay, and they are making a change to the design process at a very basic and fundamental level. We have partitioned the unified power based system design methodology into four key axes – specification, modeling, techniques, and verification, and have surveyed various state-of-the-art activities under each. While there may not be a single methodology that can be generalized across all contexts, for the most energy efficient designs of the future, we do need a unified methodology across all levels of design hierarchy and abstraction, from workload/application to transistors, for each context or system under design. An ideal methodology will include an executable specification, a synthesizable model, accurate techniques, and automatable verification methods.

Bibliography

- [1] Calypto Design Systems <http://www.calypto.com>. 38
- [2] Digital Radio Mondiale <http://www.drm.org/>. 79
- [3] J. A. Abraham, V. M. Vedula, and D. G. Saab. Verifying properties using sequential atpg. *International Test Conference*, pages 194–200, 2002. 37
- [4] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou. Precomputation-based sequential logic optimization for low power. *IEEE Trans. Very Large Scale Integr. Syst.*, 2(4):426–436, 1994. 21, 24, 43
- [5] D. Anastasakis, R. Damiano, H. K. T. Ma, and T. Stanion. A practical and efficient method for compare-point matching. In *Proceedings of the 39th conference on Design automation*, pages 305–310, 2002. 37
- [6] D. Anastasakis, L. McIlwain, and S. Pilarski. Efficient equivalence checking with partitions and hierarchical cut-points. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 539–542, New York, NY, USA, 2004. ACM Press. 104
- [7] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Proceedings of the 32nd annual international symposium on Computer Architecture, ISCA '05*, pages 298–309, Washington, DC, USA, 2005. IEEE Computer Society. 27
- [8] S. Antoy and J. Gannon. Using term rewriting to verify software. *IEEE Trans. Softw. Eng.*, 20(4):259–274, 1994. 80
- [9] T. Arts and J. Giesl. Applying rewriting techniques to the verification of erlang processes. In *CSL*, pages 96–110, 1999. 80

- [10] P. Babighian, L. Benini, A. Macii, and E. Macii. Low-overhead state-retaining elements for low-leakage mtcmos design. In *Proceedings of the 15th ACM Great Lakes symposium on VLSI, GLSVLSI '05*, pages 367–370, New York, NY, USA, 2005. ACM. 19
- [11] R. I. Bahar, G. D. Hachtel, E. Macii, and F. Somenzi. A symbolic method to reduce power consumption of circuits containing false paths. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 368–371. IEEE Computer Society Press, 1994. 18
- [12] R. I. Bahar, E. T. Lampe, and E. Macii. Power optimization of technology-dependent circuits based on symbolic computation of logic implications. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):267–293, 2000. 19
- [13] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 522–527, New York, NY, USA, 1998. ACM Press. 80
- [14] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, June 2000. 32, 116
- [15] L. Benini and G. De de Micheli. System-level power optimization: techniques and tools. *ACM Trans. Des. Autom. Electron. Syst.*, 5(2):115–192, 2000. 32
- [16] S. Bommur, N. O'Neill, and M. Ciesielski. Retiming-based factorization for sequential logic optimization. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):373–398, 2000. 23
- [17] S. Borkar, T. Karnik, and V. De. Design and reliability challenges in nanometer technologies. In *DAC*, page 75, 2004. xiv, 18

- [18] R. S. Boyer and J. S. Moore. Program verification. *Journal of Automated Reasoning*, 1(1):17–23, 1985. 136
- [19] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, page 171. IEEE Computer Society, 2001. 22, 47
- [20] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94. ACM Press, 2000. 22, 24, 43, 46, 53
- [21] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986. 77
- [22] J. R. Burch. Using bdds to verify multipliers. In *Proceedings of the 28th conference on ACM/IEEE design automation conference*, pages 408–412. ACM Press, 1991. 77
- [23] J. R. Burch and V. Singhal. Robust latch mapping for combinational equivalence checking. In *ICCAD*, pages 563–569, 1998. 37
- [24] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997. 45, 53
- [25] Wissam C. and Chansu Y. Survey on power management techniques for energy efficient computer systems, retrieved from <http://academic.csuohio.edu/yuc/mcrl/survey-power.pdf>, 2003. 32
- [26] Cadence placement-and-routing (PNR) tool. 61, 72
- [27] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power cmos digital design, 1992. 12

- [28] M. S. Chandrasekhar, J. P. Privitera, and K. W. Conradt. Application of term rewriting techniques to hardware design verification. In *24th ACM/IEEE conference proceedings on Design automation conference*, pages 277–282, 1987. 80
- [29] D. Chaver, L. Piñuel, M. Prieto, F. Tirado, and M. C. Huang. Branch prediction on demand: an energy-efficient solution. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 390–395. ACM Press, 2003. 22, 24, 43, 47
- [30] J. W. Chen, M. Dubois, and P. Stenström. Integrating complete-system and user-level performance/power simulators: The simwattch approach. *Proceedings of International Symposium on Performance Analysis of Systems and Software*, 2003. 45, 53
- [31] P-Y. Chen, C-C. Fang, T. Hwang, and H-P. Ma. Leakage reduction, delay compensation using partition-based tunable body-biasing techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 14(4):53:1–53:22, August 2009. 16, 115
- [32] S-H. Chen and J-Y. Lin. Experiences of low power design implementation and verification. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference, ASP-DAC '08*, pages 742–747, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press. 16
- [33] T. Cioara, I. Salomie, I. Anghel, I. Chira, A. Cocian, E. Henis, and R. Kat. A dynamic power management controller for optimizing servers' energy consumption in service centers. In *Proceedings of the 2010 international conference on Service-oriented computing, ICSOC'10*, pages 158–168, Berlin, Heidelberg, 2011. Springer-Verlag. 16
- [34] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311, 2003. 37

- [35] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program Slicing of Hardware Description Languages. *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999. 47
- [36] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum. Program slicing for vhd. *STTT*, 4(1):125–137, 2002. 47
- [37] O. Coudert. Gate sizing: A general purpose optimization approach. In *Proceedings of the 1996 European conference on Design and Test*, page 214. IEEE Computer Society, 1996. 18
- [38] D. Lampret *et al.* OpenRisc 1000 Architecture Manual. 2003. 45, 55, 130
- [39] N. Dershowitz. A taste of rewrite systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, pages 199–228, London, UK, 1993. Springer-Verlag. 81
- [40] S. Devadas, A. Ghosh, and K. Keutzer. *Logic synthesis*. McGraw-Hill, Inc., 1994. 19
- [41] S. Devadas and S. Malik. A survey of optimization techniques targeting low power vlsi circuits. In *Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 242–247. ACM Press, 1995. 16
- [42] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Proceedings of the 33rd annual international symposium on Computer Architecture, ISCA '06*, pages 78–88, Washington, DC, USA, 2006. IEEE Computer Society. 27
- [43] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th annual*

- IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 7–, Washington, DC, USA, 2003. IEEE Computer Society. 25
- [44] M. Fleischmann. Longrun power management - dynamic power management for cruso processors. Technical report, Transmeta Corp., 2001. 23
- [45] S. Garg and D. Marculescu. System-level mitigation of wide leakage power variability using body-bias islands. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, pages 273–278, New York, NY, USA, 2008. ACM. 18
- [46] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Conference on Automated Deduction*, pages 271–290, 2000. 80
- [47] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In *Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 253–259. IEEE Computer Society Press, 1992. 12
- [48] T. Glokler, J. Baumgartner, D. Shanmugam, R. Seigler, G. Van Huben, B. Ramanandray, H. Mony, and P. Roessler. Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 3–10, Washington, DC, USA, 2006. IEEE Computer Society. 98
- [49] C. Gopalakrishnan. *High level techniques for leakage power estimation and optimization in vlsi asics*. PhD thesis, Tampa, FL, USA, 2003. AAI3133515. 19
- [50] S. Gupta and F. N. Najm. Power macro-models for dsp blocks with application to high-level synthesis. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 103–105, New York, NY, USA, 1999. ACM. xvi, 107, 108

- [51] S. Gurun and C. Krintz. Autodvs: an automatic, general-purpose, dynamic clock scheduling system for hand-held devices. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 218–226, New York, NY, USA, 2005. ACM. 26
- [52] G. D. Hachtel, M. Hermida, A. Pardo, M. Poncino, and F. Somenzi. Re-encoding sequential circuits to reduce power dissipation. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 70–73. IEEE Computer Society Press, 1994. 20
- [53] G. Hamerly, E. Perelman, and B. Calder. How to use simpoint to pick simulation points. *SIGMETRICS Perform. Eval. Rev.*, 31(4):25–30, 2004. 53
- [54] S. Haynal, T. Kam, M. Kishinevsky, E. Shriver, and X. Wang. A system verilog rewriting system for rtl abstraction with pentium case study. In *MEM-OCODE*, pages 79–88, 2008. 80
- [55] K. Hazelwood and D. Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *International Symposium on Low-Power Electronics and Design*, Newport Beach, CA, August 2004. 22
- [56] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Code transformations for energy-efficient device management. In *IEEE Transactions on Computers*, 2004. 31
- [57] J. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *X IFIP International Conference on VLSI (VLSI 99)*, Lisbon, Portugal, November 1999. 80
- [58] C.-H. Hsu, U. Kermer, and M. Hsiao. Compiler directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *ISLPED '01: Proceedings of ISLPED '01*, pages 275–278, 2001. 31

- [59] S. Y. Huang, K. T. Cheng, and K. C. Chen. Verifying sequential equivalence using atpg techniques. *ACM Trans. Des. Autom. Electron. Syst.*, pages 244–275, 2001. 24, 37, 43
- [60] IEEE:1801-2009. IEEE Standard 1801-2009 - IEEE Standard for Design and Verification of Low Power Integrated Circuits <http://standards.ieee.org/findstds/standard/1801-2009.html> . 8
- [61] S. Iman and M. Pedram. Multi-level network optimization for low power. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 372–377. IEEE Computer Society Press, 1994. 19
- [62] Intel Corporation, Microsoft Corporation, Toshiba, and Phoenix. *Advanced Configuration and Power Interface*. 27, 115
- [63] Intel Developer Forum. *Panel Self Refresh*. 127
- [64] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994. 37, 77
- [65] D. Kapur and H. Zhang. An overview of Rewrite Rule Laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995. 77
- [66] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning : An Approach*. Norwell, MA, USA, 2000. Kluwer Academic Publishers. 136
- [67] M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology. 46, 77, 130

- [68] J. Klop. Term Rewriting Systems. In *In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors: Handbook of Logik in Computer Science, Oxford University Press*, volume 2, pages 1–116, 1992. 78, 81
- [69] U. Kremer. Low power/energy compiler optimizations, *Low-Power Electronics Design* (Editor: Christian Piguet), CRC Press. 2005. 31
- [70] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371, 2003. 37
- [71] A. Kuehlmann and C. A. J. van Eijk. Combinational and sequential equivalence checking. pages 343–372, 2002. 37, 97
- [72] W. Kunz and D. K. Pradhan. Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits. In *Proceedings of the IEEE International Test Conference on Discover the New World of Test and Design*, pages 816–825. IEEE Computer Society, 1992. 23
- [73] E. Le Sueur and G. Heiser. Slow down or sleep, that is the question. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC’11, pages 16–16, Berkeley, CA, USA, 2011. USENIX Association. 11
- [74] C. Lemonds and S. S. M. Shetti. A low power 16 by 16 multiplier using transition reduction circuitry. In *Proceedings of the International Workshop on Low-Power Design*, pages 139–142, April 1994. 19
- [75] J. Li and J.F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 77–87, 11-15 Feb. 2006. 27
- [76] D. Liu and C. Svensson. Power consumption estimation in CMOS VLSI chips. *IEEE Journal of Solid-State Circuits*, pages 663–670, June 1994. 21

- [77] Y. Liu and H. Zhu. A survey of the research on power management techniques for high-performance systems. *Softw., Pract. Exper.*, 40(11):943–964, 2010. 32
- [78] M. Lorenz, L. Wehmeyer, and T. Dräger. Energy aware compilation for dsps with simd instructions. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 94–101. ACM Press, 2002. 31
- [79] F. Lu, M. K. Iyer, G. Parthasarathy, Li. . Wang, K. T. Cheng, and K. C. Chen. An efficient sequential sat solver with improved search strategies. In *DATE*, pages 1102–1107, 2005. 37
- [80] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984. 47
- [81] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(11):1061–1079, 1998. 32
- [82] A. J. Martin, M. Nystrom, and P. I. Penzes. Et2: a metric for time and energy efficiency of computation. pages 293–315, 2002. 60
- [83] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *In Proc. ICCAD-02*, pages 721–725, 2002. 25
- [84] T. Matsumoto, H. Saito, and M. Fujita. An equivalence checking method for c descriptions based on symbolic simulation with textual differences. In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, pages 3315–3323, 2005. 38

- [85] Y. Matsunaga. An Efficient Equivalence Checker for Combinational Circuits. In *Proceedings of Design Automation Conference*, pages 629–634, 1996. 24, 43
- [86] A. Mehrotra, S. Qadeer, V. Singhal, R. K. Brayton, A. Aziz, and A. L. Sangiovanni-Vincentelli. Sequential optimisation without state space exploration. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 208–215. IEEE Computer Society, 1997. 23, 37
- [87] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *ISLPED '97: Proceedings of the 1997 International Symposium on Low Power Electronics and Eesign*, pages 72–75, 1997. 31
- [88] Mem-pwr-mgmt. Memory Power Management <http://lwn.net/Articles/446493/>, 2006. 29
- [89] J. Monteiro, S. Devadas, and A. Ghosh. Retiming sequential circuits for low power. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 398–402. IEEE Computer Society Press, 1993. 20
- [90] J. Monteiro, J. Rinderknecht, S. Devadas, and A. Ghosh. Optimization of combinational and sequential logic circuits for low power using precomputation. In *Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*, page 430. IEEE Computer Society, 1995. 21, 24, 43, 49
- [91] Mosis-TSMC 0.18 μ m CLO18 Process. <http://www.mosis.org/products/fab/vendors/tsmc/tsmc018/>. 2004. 55, 69
- [92] M. Mutz. Using the hol prove assistant for proving the correctness of term rewriting rules reducing terms of sequential behavior. In *Proc. Workshop on Computer-Aided Verification*, pages 277–287, 1991. 37, 80

- [93] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and thermal management in the intel core duo processor. *Intel Technology Journal*, 10(2):109–122, 2006. 23, 115
- [94] I. Nedelchev. Power compiler: a gate-level power optimization and synthesis system. In *Proceedings of the 1997 Intl. Conference on Computer Design (ICCD '97)*, ICCD '97, pages 74–, Washington, DC, USA, 1997. IEEE Computer Society. 21, 115
- [95] OPENCORES. <http://www.opencores.org>. 55
- [96] V. Pallipadi and A. Starikovskiy. The ondemand governor: past, present and future. In *Proceedings of Linux Symposium, vol. 2*, pp. 223-238, 2006. 26
- [97] C. Papachristou, M. Spining, and M. Nourani. A multiple clocking scheme for low power rtl design. In *Proceedings of the 1995 international symposium on Low power design*, pages 27–32. ACM Press, 1995. 22, 24, 43
- [98] PASR. Low Power Function of Mobile RAM: Partial Array Self Refresh <http://www.elpida.com/pdfs/E0597E10.pdf>, 2005. 29
- [99] M. Pedram. Power minimization in ic design: principles and applications. *ACM Trans. Des. Autom. Electron. Syst.*, 1(1):3–56, January 1996. 21
- [100] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 244, Washington, DC, USA, 2003. IEEE Computer Society. 53
- [101] C. Pixley, S.-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 620–623, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 97

- [102] J. Pouwelse, K. Langendoen, and H.J. Sips. Application-directed voltage scaling. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(5):812–826, Oct. 2003. 26
- [103] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005. 37
- [104] S.C. Prasad and K. Roy. Circuit optimization for minimization of power consumption under delay constraint. In *Proceedings of the International Workshop on Low-Power Design*, pages 15–20, April 1994. 17, 24, 43
- [105] PUMA: Dual-Issue PowerPC Fixed Point Unit. The reported work contains IP components courtesy of the University of Michigan Intellectual Property Source (UMIPS) and Jayakumaran Sivagnaname, Rahul Rao, and Richard B. Brown of the University of Michigan. 2004. 46, 65
- [106] S. Qadeer, R. K. Brayton, and V. Singhal. Latch redundancy removal without global reset. In *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 432–439. IEEE Computer Society, 1996. 23
- [107] Q. Qiu, Q. Wu, M. Pedram, and C-S. Ding. Cycle-accurate macro-models for rt-level power analysis. In *Proceedings of the 1997 international symposium on Low power electronics and design, ISLPED '97*, pages 125–130, New York, NY, USA, 1997. ACM. 21
- [108] G. Qu. Power management of multicore multiple voltage embedded systems by task scheduling. In *Proceedings of the 2007 Intl. Conf. on Parallel Processing Workshops, ICPPW '07*, Washington, DC, USA, 2007. IEEE Computer Society. 16
- [109] V. Raghunathan, M. B. Srivastava, and R. K. Gupta. A survey of techniques for energy efficient on-chip communication. In *DAC*, pages 900–905. ACM, 2003. 32

- [110] K. Roy and S. Prasad. Syclop: Synthesis of cmos logic for low power applications. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 464–467. IEEE Computer Society, 1992. 20
- [111] A. Sakanaka, S. Fujii, and T. Sato. A leakage-energy-reduction technique for highly-associative caches in embedded systems. *SIG ARCH Comput. Archit. News*, 32(3):50–54, 2004. 24, 46
- [112] S. S. Sapatnekar and W. Chuang. Power-delay optimizations in gate sizing. *ACM Trans. Des. Autom. Electron. Syst.*, 5(1):98–114, 2000. 18
- [113] J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *Proc. 10th International Computer Aided Verification Conference*, pages 135–146, 1998. 77
- [114] R. Sharp and O. Rasmussen. Rewriting with constraints in t-ruby. In *Conference on Correct Hardware Design and Verification Methods*, pages 226–241, 1993. 80
- [115] X. Shen. Design and Verification of Speculative Processors. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998. 80
- [116] Y. Shin, J. Seomun, K-M. Choi, and T. Sakurai. Power gating: Circuits, design methodologies, and best practice for standard-cell vlsi designs. *ACM Trans. Des. Autom. Electron. Syst.*, 15(4):28:1–28:37, October 2010. 16
- [117] G. S. Silveira, A. V. Brito, and E. U. K. Melcher. Functional verification of power gate design in systemc rtl. In *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes, SBCCI '09*, pages 52:1–52:5, New York, NY, USA, 2009. ACM. 40

- [118] T. Simunic, L. Benini, A. Acquaviva, P. W. Glynn, and G. D. Micheli. Dynamic voltage scaling and power management for portable systems. In *DAC*, pages 524–529. ACM, 2001. 26
- [119] D. Singh, J. M. Rabaey, M. Pedram, F. Catthoor, S. Rajgopal, N. Sehgal, and T.J Mozdzen. Power conscious cad tools and methodologies: a perspective. *Proceedings of the IEEE*, 83(4):570–594, apr 1995. 21
- [120] V. Singhal. Design replacement for sequential circuits. In *Ph.D. Thesis, Dept. of Electrical Engineering, University of California Berkeley*, 1999. 23
- [121] V. Singhal, C. Pixley, A. Aziz, S. Qadeer, and R. Brayton. Sequential optimization in the absence of global reset. *ACM Trans. Des. Autom. Electron. Syst.*, 8(2):222–251, 2003. 23
- [122] L. Smria, R. Mehra, B. Pangrle, A.rjuna Ekanayake, A. Seawright, and D. Ng. Rtl c-based methodology for designing and verifying a multi-threaded processor. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 123–128, 2002. 37
- [123] M. Stan and W. Burlison. Limited-weight codes for low-power i/o. In *Proceedings of the International Workshop on Low-Power Design*, pages 209–214, April 1994. 20
- [124] N. Takagi, H. Yasuura, and S. Yajima. High-speed vlsi multiplication algorithm with a redundant binary addition tree. *IEEE Trans. Comput.*, 34(9):789–796, 1985. 101
- [125] C-H. Tan and J. Allen. Minimization of power in vlsi circuits using transistor sizing, input ordering and statistical power estimation. In *Proceedings of the International Workshop on Low-Power Design*, pages 75–80, April 1994. 17, 18
- [126] The VIS Group. VIS: A system for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference*

- on *Computer Aided Verification*, pages 428–432. Springer Lecture Notes in Computer Science #1102, July 1996. 50
- [127] Tickless-Kernel. Tickless Idle <http://www.lesswatts.org/projects/tickless/>, 2010. 29
- [128] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995. 47
- [129] V. Tiwari, S. Malik, and P. Ashar. Guarded evaluation: pushing power management to logic synthesis/design. In *Proceedings of the 1995 international symposium on Low power design*, pages 221–226. ACM Press, 1995. 21, 24, 43, 49
- [130] C. Y. Tsui, M. Pedram, C. Chen, and A. M. Despain. Low power state assignment targeting two-and multi-level logic implementations. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 82–87. IEEE Computer Society Press, 1994. 20
- [131] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. Softsdv: Intel technology journal item q41999 (softsdv.pdf). 41
- [132] K. Usami and M. Horowitz. Clustered voltage scaling technique for low-power design. In *Proceedings of the 1995 international symposium on Low power design*, ISLPED '95, pages 3–8, New York, NY, USA, 1995. ACM. 25
- [133] S. Li V. Pallipadi, A. Belay. cpuidle do nothing, efficiently. In *Proceedings of the Ottawa Linux Symposium*, 2007. 28
- [134] M. Valluri, L. John, and H. Hanson. Exploiting compiler generated schedules for energy savings in high performance processors. In *ISPLED '03: Proceedings of the International Symposium on Low Power Electronics and Design*, pages 414–419, 2003. 31

- [135] C. van Eijk and J. Jess. Detection of equivalent state variables in finite state machine verification, 1995. 37
- [136] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 618–623, Washington, DC, USA, 1998. IEEE Computer Society. 37, 97, 98
- [137] K. Vardarajan, S.K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In *Proceedings of International Symposium on Microarchitecture (MICRO-39)*, 2006. 24
- [138] S. Vasudevan, V. Viswanath, J. A. Abraham, and J. Tu. Automatic decomposition for sequential equivalence checking of system level and rtl descriptions. In *Proceedings of IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2006)*, pages 71–80, 2006. 24, 39, 43, 104
- [139] S. Vasudevan, V. Viswanath, J. A. Abraham, and J. Tu. Sequential equivalence checking of system level and rtl descriptions using effective compare points. *Austin Conference on Integrated Systems & Circuits (ACISC)*, May 2006. 104
- [140] S. Vasudevan, V. Viswanath, J. A. Abraham, and J. Tu. Sequential equivalence checking between system level and rtl descriptions. *Submitted to IEEE Transactions on VLSI*, 2007. 104
- [141] S. Vasudevan, V. Viswanath, J. A. Abraham, and J. Tu. Sequential equivalence checking between system level and rtl descriptions. *Design Automation for Embedded Systems*, 12:377–396, 2008. 10.1007/s10617-008-9033-z. 39

- [142] S. Vasudevan, V. Viswanath, R. W. Summers, and J. A. Abraham. Automatic verification of arithmetic circuits in rtl using stepwise refinement of term rewriting systems. *IEEE Trans. Comput.*, 56(10):1401–1414, 2007. [38](#), [78](#), [80](#), [90](#), [92](#), [93](#), [100](#)
- [143] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *J. Electron. Test.*, 19(2):149–160, 2003. [47](#), [51](#)
- [144] V. Viswanath and J. A. Abraham. Automatic and correct register transfer level annotations for low power microprocessor design. *J. Low Power Electronics*, 8(4):424–439, 2012. [43](#), [130](#)
- [145] V. Viswanath, J. A. Abraham, and W. A. Hunt, Jr. Automatic insertion of low power annotations in rtl for pipelined microprocessors. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 496–501, 2006. [24](#), [43](#), [115](#)
- [146] V. Viswanath, R. Muralidhar, H. Seshadri, and J. A. Abraham. On a rewriting strategy for dynamically managing power constraints and power dissipation in socs. In *ISQED*, pages 128–134, 2013. [128](#)
- [147] V. Viswanath, R. Muralidhar, H. Seshadri, and A. S. Narayan. Power management methods: From specification and modeling, to techniques and verification. *J. Low Power Electronics*, 8(4):353–377, 2012. [17](#)
- [148] V. Viswanath, S. Vasudevan, and J. A. Abraham. Dedicated rewriting: Automatic verification of low power transformations in rtl. *Journal of Low Power Electronics*, 5(3):339–353, 2009. [24](#), [130](#)
- [149] V. Viswanath, S. Vasudevan, and J. A. Abraham. Dedicated rewriting: Automatic verification of low power transformations in rtl. In *VLSID '09: Proceedings of the 2009 22nd International Conference on VLSI Design*, pages 77–82, Washington, DC, USA, 2009. IEEE Computer Society. [24](#), [130](#)

- [150] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Transactions on Information Theory*, pages 260–269, 1967. 79, 107
- [151] P-H. Wang, C-L. Yang, Y-M. Chen, and Y-J. Cheng. Power gating strategies on gpus. *ACM Trans. Archit. Code Optim.*, 8(3):13:1–13:25, October 2011. 16, 115
- [152] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association. 25
- [153] T. W. Williams. The future is low power and test. In *Proceedings of the 2008 13th European Test Symposium*, ETS '08, pages 4–, Washington, DC, USA, 2008. IEEE Computer Society. xiv, 33, 34
- [154] L. C. Yeo, L. C. Ng, N. S. Ho, and C. Konduru. Dynamic power gating implementation on intel embedded media and graphics driver. In *Intel Corporation, White Paper 325293-001*, 2011. 16
- [155] O. Zendra. Memory and compiler optimizations for low power and energy. Technical report, 2006. 30
- [156] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner. The limit of dynamic voltage scaling and insomniac dynamic voltage scaling. *IEEE Trans. Very Large Scale Integr. Syst.*, 13:1239–1252, November 2005. 25, 116
- [157] C. Zhang, F. Vahid, J. Yang, and W. Najjar. A way-halting cache for low-energy high-performance systems. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 126–131. ACM Press, 2004. 24, 46
- [158] Z. Zhou and W. Burleson. Equivalence checking of datapaths based on canonical arithmetic expressions. In *Proceedings of the 32nd ACM/IEEE*

conference on Design automation conference, pages 546–551. ACM Press, 1995. 80

Vita

Vinod Viswanath was born in Mysore, India. He received the Bachelor of Engineering degree in Computer Engineering from Karnataka Regional Engineering College, Suratkal in 1997. He then received the Master of Science in Electrical Engineering and Master of Philosophy in Engineering and Applied Sciences from Yale University in 1999 and 2001 respectively. He started graduate studies at the University of Texas at Austin in Fall 2001. From 2005 until end of 2011, he worked as a Formal Methods Researcher at Intel Corporation in Austin, TX. Since January 2012, he is working as a Sr. Member of Technical Staff at Real Intent, a startup company based out of Sunnyvale, CA.

Permanent address: Mail: 3585 Agate Drive Apt 204, Santa Clara CA 95051
Email: vinod@cerc.utexas.edu
URL: <http://www.cerc.utexas.edu/~vinod>

This dissertation was typeset with \LaTeX^\dagger by the author.

[†] \LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's \TeX Program.